Все, что программист должен знать о мышлении

УМ ПРОГРАММИСТА

КАК ПОНЯТЬ И ОСМЫСЛИТЬ ЛЮБОЙ КОД

Фелин Херманс

bhv®

MANNING

The Programmer's Brain

WHAT EVERY PROGRAMMER NEEDS TO KNOW ABOUT COGNITION

FELIENNE HERMANS
FOREWORD BY JON SKEET

MANNING
SHELTER ISLAND

УМ ПРОГРАММИСТА
КАК ПОНЯТЬ И ОСМЫСЛИТЬ ЛЮБОЙ КОД

Фелин Херманс

Предисловие Джона Скита

Санкт-Петербург

«БХВ-Петербург»

2023

УДК 004.4
ББК 32.973.26-02
Х39

Херманс Ф.

Х39 Ум программиста. Как понять и осмыслить любой код: Пер. с англ. — СПб.: БХВ-Петербург, 2023. — 272 с.: ил.

ISBN 978-5-9775-1176-6

Книга освещает практические основы когнитивистики для программистов. Основные темы: осмысление и развитие чужого и собственного кода, изучение новых языков программирования, мнемонические приемы для программистов, поддержка кода в читаемом состоянии. Объяснено, как снижать когнитивную нагрузку при работе программиста, как делать код логичным и понятным для себя и коллег. Рассмотрены приемы именования функций, классов и переменных, подходы к ведению репозиториев, совместной разработке и доработке кода.

Для программистов и других IT-специалистов

УДК 004.4
ББК 32.973.26-02

Группа подготовки издания:

Руководитель проекта	*Олег Сивченко*
Зав. редакцией	*Людмила Гауль*
Перевод с английского	*Кристины Черниковой*
Компьютерная верстка	*Ольги Сергиенко*
Оформление обложки	*Зои Канторович*

Подписано в печать 02 09 22
Формат 70×100 1/16 Печать офсетная Усл. печ. л. 21,93
Тираж 1000 экз. Заказ № 5329
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-1-61729-867-7 (англ.)
ISBN 978-5-9775-1176-6 (рус.)

Оглавление

Предисловие **13**

От автора **15**

Благодарности **17**

О книге **19**
Структура книги 19
Дискуссионный форум liveBook 20

Об авторе **21**

Об обложке **23**

ЧАСТЬ I. ОБ УЛУЧШЕНИИ НАВЫКОВ ЧТЕНИЯ КОДА **25**

Глава 1. Определение вашего типа замешательства при кодировании **27**
1.1. Разные типы замешательства в коде 28
1.1.1. Первый тип замешательства — недостаток знаний 29
1.1.2. Второй тип замешательства — недостаток информации 29
1.1.3. Третий тип замешательства — недостаток вычислительной мощности 30
1.2. Различные когнитивные процессы, влияющие на процесс кодирования 31
1.2.1. Долговременная память и программирование 31
Программа на APL с точки зрения долговременной памяти 32
1.2.2. Кратковременная память и программирование 32
Программа на Java с точки зрения кратковременной памяти 32
1.2.3. Рабочая память и программирование 33
Программа на BASIC с точки зрения рабочей памяти 33
1.3. Совместная работа когнитивных процессов 34
1.3.1. Краткое описание того, как когнитивные процессы взаимодействуют друг с другом 34
1.3.2. Когнитивные процессы и программирование 35
Выводы 37

Глава 2. Скорочтение кода **39**
2.1. Быстрое чтение кода 40
2.1.1. Что только что происходило в вашем мозге 41
2.1.2. Перепроверка воспроизведенного кода 42
Вторая попытка воспроизведения кода 43

2.1.3. Перепроверка воспроизведенного 44
2.1.4. Почему читать незнакомый код так сложно 44
2.2. Преодоление лимитов памяти 45
2.2.1. Сила чанков 45
Чанки кода 48
2.2.2. Опытные программисты запоминают код лучше начинающих программистов 48
2.3. Вы видите намного больше кода, чем можете прочитать 49
2.3.1. Иконическая память 50
Иконическая память и код 51
2.3.2. Это не то, что вы помните; это то, как вы запоминаете 51
Как написать код, который можно разделить на чанки 52
Используйте паттерны проектирования 53
Пишите комментарии 54
Оставляйте «маячки» 55
2.3.3. Применяйте чанки 58
Выводы 59

Глава 3. Как быстро выучить синтаксис 61
3.1. Советы по запоминанию синтаксиса 62
3.1.1. Отвлечение снижает производительность 62
3.2. Как быстро выучить синтаксис с использованием карточек 63
3.2.1. Когда использовать карточки 64
3.2.2. Расширяем набор карточек 64
3.2.3. Убираем ненужные карточки 64
3.3. Как не забывать информацию 65
3.3.1. Почему мы забываем 65
Иерархия и сеть 66
Кривая забывания 66
3.3.2. Интервальное повторение 67
3.4. Как запомнить синтаксис надолго 69
3.4.1. Два способа запоминания информации 69
Уровень хранения 69
Уровень воспроизведения 69
3.4.2. Просто увидеть недостаточно 70
3.4.3. Воспоминания укрепляют память 70
3.4.4. Укрепление памяти путем активного мышления 71
Схемы 72
Проработка для запоминания концепций программирования 73
Выводы 74

Глава 4. Как читать сложный код 75
4.1. Почему так тяжело понимать сложный код 76
4.1.1. Чем друг от друга отличаются рабочая память и кратковременная память 77
4.1.2. Типы когнитивной нагрузки и как они связаны с программированием 78
Внутренняя когнитивная нагрузка при чтении кода 78
Внешняя когнитивная нагрузка при чтении кода 79
4.2. Способы снижения когнитивной нагрузки 80
4.2.1. Рефакторинг 80

4.2.2. Замена незнакомых языковых конструкций ... 82
Лямбда-функции ... 82
Генератор списков ... 83
Тернарные операторы ... 84
4.2.3. Синонимизация — отличное дополнение к дидактическим карточкам ... 85
4.3. Вспомогательные средства при перегрузке рабочей памяти ... 85
4.3.1. Создание графа зависимостей ... 86
4.3.2. Использование таблицы состояний ... 88
4.3.3. Сочетание графов зависимостей и таблиц состояний ... 91
Выводы ... 93

ЧАСТЬ II. ПРОДОЛЖАЕМ ДУМАТЬ О КОДЕ ... 95

Глава 5. Совершенствуем навыки углубленного понимания кода ... 97
5.1. Роли переменных ... 98
5.1.1. Разные переменные выполняют разные действия ... 98
5.1.2. Одиннадцать ролей, охватывающие почти все переменные ... 99
5.2. Роли и принципы ... 101
5.2.1. Польза ролей ... 102
Практические советы по работе с ролями переменных ... 103
5.2.2. Венгерская нотация ... 104
Системная и прикладная венгерские нотации ... 104
5.3. Углубленное понимание программ ... 106
5.3.1. Понимание текста и понимание плана ... 106
5.3.2. Этапы понимания программы ... 106
Применение этапов углубленного понимания ... 107
5.4. Чтение кода как обычного текста ... 109
5.4.1. Что происходит в мозге при чтении кода ... 110
Поля Бродмана ... 110
Показания фМРТ ... 111
5.4.2. Если вы можете выучить французский, то сможете выучить и Python ... 111
Как люди читают код ... 113
Перед тем как читать код, программисты сканируют его ... 114
Начинающие и опытные программисты читают код по-разному ... 114
5.5. Стратегии понимания текста, которые можно применить к коду ... 115
5.5.1. Активация пассивных знаний ... 116
5.5.2. Наблюдение ... 116
5.5.3. Определение важности разных строк кода ... 117
5.5.4. Предположения о значении имен переменных ... 118
5.5.5. Визуализация ... 119
Таблица операций ... 119
5.5.6. Постановка вопросов ... 120
5.5.7. Резюмирование кода ... 121
Выводы ... 122

Глава 6. Совершенствуем навыки решения задач программирования ... 123
6.1. Использование моделей для размышлений о коде ... 124
6.1.1. Преимущества использования моделей ... 124
Не все модели одинаково полезны ... 125

6.2. Ментальные модели 126
6.2.1. Подробное исследование ментальных моделей 128
6.2.2. Изучение новых ментальных моделей 129
6.2.3. Как эффективно использовать ментальные модели во время размышлений о коде 130
Ментальные модели в рабочей памяти 130
Точные модели работают лучше 131
Создание ментальных моделей исходной программы в рабочей памяти 131
Ментальные модели в долговременной памяти 132
Создание ментальных моделей исходной программы в долговременной памяти 133
Ментальные модели, одновременно хранящиеся в долговременной и рабочей памяти 134
6.3. Условные машины 135
6.3.1. Что такое условная машина 135
6.3.2. Примеры условных машин 136
6.3.3. Разные уровни условных машин 137
6.4. Условные машины и язык 138
6.4.1. Расширяем набор условных машин 139
6.4.2. Разные условные машины могут создать взаимно конфликтующие ментальные модели 140
6.5. Условные машины и схемы 141
6.5.1. Почему схема важна 141
6.5.2. Являются ли условные машины семантическими 142
Выводы 142

Глава 7. Заблуждения 143
7.1. Почему второй язык программирования выучить намного проще, чем первый 144
7.1.1. Как увеличить шанс воспользоваться знаниями по программированию 146
7.1.2. Разные виды трансференции 147
Осознанная и неосознанная трансференция 147
Близкая и дальняя трансференция 147
7.1.3. Знания: добро или зло? 148
7.1.4. Сложности трансференции 149
7.2. Заблуждения. Ошибки в мышлении 150
7.2.1. Исправление заблуждений путем концептуальных замен 152
7.2.2. Подавление заблуждений 152
7.2.3. Заблуждения о языках программирования 153
7.2.4. Предотвращение заблуждений при изучении нового языка программирования 155
7.2.5. Выявление заблуждений в новой базе кода 156
Выводы 157

ЧАСТЬ III. О ХОРОШЕМ КОДЕ 159

Глава 8. Совершенствуем навыки присваивания имен 161
8.1. Почему присваивание имен так важно 162
8.1.1. Почему присваивание имени так важно 162
Имена составляют существенную часть кодовой базы 163

Имена играют роль в обзорах кода ... 163
Имена — это самая удобная форма документации ... 163
Имена могут служить маячками ... 163
8.1.2. Разные точки зрения на присваивание имен ... 163
Хорошее имя можно определить синтаксически ... 164
Имена во всей базе кода должны быть единообразны ... 165
8.1.3. Важно грамотно подбирать имена ... 166
Заключения о практике присваивания имен ... 167
8.2. Когнитивные аспекты присваивания имен ... 167
8.2.1. Форматирование имен поддерживает кратковременную память ... 168
8.2.2. Понятные имена лучше закрепляются в долговременной памяти ... 169
8.2.3. Полезная информация в именах переменных ... 170
8.2.4. Когда стоит оценивать качество имен ... 171
8.3. Какие типы имен проще всего понимать ... 172
8.3.1. Использовать аббревиатуры или нет? ... 172
Однобуквенные имена переменных ... 173
8.3.2. Змеиный или верблюжий регистры? ... 176
8.4. Влияние имен на ошибки кода ... 177
8.4.1. В коде с некачественными именами больше ошибок ... 177
8.5. Как выбирать хорошие имена ... 178
8.5.1. Шаблоны имен ... 178
8.5.2. Трехступенчатая модель Фейтельсона для хороших имен переменных ... 180
Трехступенчатая модель во всех деталях ... 181
Успех трехступенчатой модели Фейтельсона ... 181
Выводы ... 182

Глава 9. Боремся с плохим кодом и когнитивной нагрузкой. Две концепции ... 183
9.1. Почему код с запахами кода создает большую когнитивную нагрузку ... 184
9.1.1. Краткая информации о запахах кода ... 184
Запахи кода на уровне метода ... 186
Запахи кода на уровне класса ... 186
Запахи кода на уровне базы кода ... 187
Влияние запахов кода ... 187
9.1.2. Как запахи кода вредят мышлению ... 188
«Длинный список параметров», сложные «Операторы переключения» — перегрузка рабочей памяти ... 188
«Всемогущий класс», «Длинный метод» — невозможно эффективно разбить код на чанки ... 189
«Клоны кода» — невозможно правильно разбить код на чанки ... 189
9.2. Зависимость когнитивной нагрузки от плохих имен ... 190
9.2.1. Лингвистические антипаттерны проектирования ... 190
9.2.2. Измерение когнитивной нагрузки ... 191
Шкала Пааса для когнитивной нагрузки ... 192
Измерение нагрузки по глазам ... 193
Измерение нагрузки по коже ... 193
Измерение нагрузки по мозгу ... 194
Запись биотоков мозга ... 194
Функциональная fNIRS-томография и программирование ... 195

9.2.3. Лингвистические антипаттерны и когнитивная нагрузка 195
9.2.4. Почему лингвистические антипаттерны вызывают замешательство 196
Выводы 197

Глава 10. Совершенствуем навыки решения сложных задач 199
10.1. Что такое решение задач 200
10.1.1. Элементы решения задач 200
10.1.2. Пространство состояний 200
10.2. Какую роль при решении задач программирования играет долговременная память 201
10.2.1. Решение задачи — это отдельный когнитивный процесс? 201
При решении задач вы используете долговременную память 202
Вашему мозгу проще решить знакомые задачи 202
10.2.2. Как научить долговременную память решать задачи 203
10.2.3. Два вида памяти, наиболее существенные при решении задачи 203
Какие виды памяти играют роль при решении задач 204
Потеря знаний или навыков 205
10.3. Автоматизация: создание имплицитной памяти 206
10.3.1. Имплицитная память с течением времени 207
Когнитивный этап 208
Ассоциативный этап 208
Автономный этап 209
10.3.2. Почему автоматизация помогает программировать быстрее 210
10.3.3. Улучшение имплицитной памяти 211
10.4. Обучение на основе кода и его объяснения 212
10.4.1. Новый вид когнитивной нагрузки: соответствующая нагрузка 213
10.4.2. Примеры с решением на практике 215
Работайте вместе с коллегой 215
Используйте GitHub 215
Читайте книги или блоги об исходном коде 216
Выводы 216

ЧАСТЬ IV. О СОВМЕСТНОЙ РАБОТЕ НАД КОДОМ 217

Глава 11. Процесс написания кода 219
11.1. Различные активности, выполняемые во время программирования 220
11.1.1. Поиск 220
11.1.2. Осмысление 221
11.1.3. Переписывание 221
11.1.4. Наращивание 222
11.1.5. Исследование 222
11.1.6. А как же отладка? 223
11.2. Программист отвлекся 223
11.2.1. Задачи программирования нуждаются в «разогреве» 224
11.2.2. Что происходит после отвлечения 225
11.2.3. Как подготовиться к отвлечению 225
Сохраняйте воображаемую модель 225
Помогите своей проспективной памяти 226
Определитесь с промежуточными целями 228

11.2.4. Когда отвлекать программиста 228
11.2.5. Пара слов о многозадачности 230
Многозадачность и автоматизация 230
Исследования многозадачности 231
Выводы 231

Глава 12. Проектирование и усовершенствование больших систем 233
12.1. Проверка свойств базы кода 234
12.1.1. Когнитивные измерения 234
Подверженность ошибкам 235
Согласованность 236
Размытость 236
Скрытые зависимости 237
Преждевременная фиксация решения 238
Вязкость 238
Поэтапное оценивание 239
Выразительность ролей 239
Близость соответствия 240
Трудность мыслительных операций 241
Вторичные обозначения 242
Градиент абстракции 242
Наглядность 243
12.1.2. Использование когнитивных измерений базы кода для улучшения базы кода 243
12.1.3. Проектные маневры и их плюсы и минусы 244
Подверженность ошибкам и вязкость 244
Преждевременная фиксация решения и поэтапное оценивание против подверженности ошибкам 245
Выразительность ролей и размытость 245
12.2. Измерения и активности 245
12.2.1. Влияние измерений на разные активности 245
Поиск 245
Осмысление 246
Переписывание 246
Наращивание 246
Исследование 247
12.2.2. Изменение базы кода под ожидаемые активности 247
Выводы 247

Глава 13. Как ввести новых программистов в курс дела 249
13.1. Проблемы процесса адаптации 249
13.2. Различия между профессионалами и новичками 251
13.2.1. Поведение новичка более подробно 251
Оригинальная концепция Пиаже 251
Концепция неопиажизма для программирования 252
При изучении новой информации вы можете временно забывать некоторые вещи 255
13.2.2. Разница между вещественным и абстрактным видением концепций 255

13.3. Активности для улучшения процесса адаптации 258
13.3.1. Ограничение заданий до одной активности 258
13.3.2. Поддержка памяти новичка 259
Поддержка долговременной памяти: объяснение релевантной информации 259
Поддержка кратковременной памяти: ставьте небольшие конкретные задачи 260
Поддержка рабочей памяти: используйте диаграммы 261
13.3.3. Совместное чтение кода 261
Активация 262
Определение важности 262
Постановка предположений 262
Наблюдение 262
Визуализация 263
Постановка вопросов 263
Резюмирование 263
Выводы 263

Эпилог. Пара слов перед прощанием 265

Предметный указатель 267

Предисловие

Бóльшую часть своей жизни я размышлял о программировании. Если вы сейчас читаете эту книгу, то готов поспорить, что и вы тоже. Однако я никогда не тратил много времени на размышления о том, как я думаю. Для меня всегда было важно понятие о наших мыслительных процессах и то, как мы взаимодействуем с кодом как люди, однако я не опирался ни на какие научные исследования. Давайте я приведу три примера.

Я — ведущий участник .NET-проекта Noda Time, предоставляющего набор типов даты и времени, альтернативный встроенному в .NET набору. Это была отличная возможность погрузиться в проектирование программных интерфейсов и особенно в придумывание имен! Я увидел множество проблем, связанных с именами: они звучат так, словно меняют уже существующее значение, хотя на самом деле возвращают новое значение! Поэтому я постарался использовать такие имена, чтобы код с ошибками при чтении выглядел неправильно. Например, тип `LocalDate` с методом `PlusDays`, а не `AddDays`. Надеюсь, что большинству C#-разработчиков данный код покажется неправильным:

```
date.PlusDays(1);
```

В то время как следующий код будет более понятным:

```
tomorrow = today.PlusDays(1);
```

Сравните с методом `AddDays` .NET-типа `DateTime`:

```
date.AddDays(1);
```

Кажется, что это всего лишь изменение даты и ошибки тут нет, хотя этот вариант, как и первый, неправильный.

Второй пример, более общего характера, также взят из проекта Noda Time. В то время как многие библиотеки стараются (из лучших побуждений) выполнять всю тяжелую работу за разработчика, мы хотим, чтобы пользователи Noda Time заранее продумали код обработки даты и времени. Мы пытаемся заставить пользователей однозначно сформулировать, чего именно они хотят достичь, а затем помогаем им выразить это в коде.

И наконец, концептуальный пример: какие значения хранятся в переменных Java и C# и что происходит, когда вы передаете аргумент методу. Мне кажется, что бóльшую часть своей жизни я пытаюсь опровергнуть концепцию того, что на Java объекты передаются по ссылке. Похоже, что так и есть: я уже четверть века помогаю другим разработчикам настраивать их мысленные модели.

Получается, мне всегда было важно, как думают другие программисты, но у меня не было никаких знаний в этой области — я довольствовался догадками на своем выстраданном опыте. Эта книга помогает мне это изменить, хотя и не является отправной точкой для меня.

С Фелиной Херманс я познакомился в 2017 году в Осло на конференции NDC, где она выступала с презентацией «Programming Is Writing Is Programming». И моя реакция в Твиттере говорит сама за себя: «Мне понадобится время, чтобы осознать все это! Но я потрясен. Просто потрясен». Я минимум трижды (конечно, в разное время) посещал эту презентацию Фелины и каждый раз узнавал что-то новое. Наконец-то я получил научное объяснение тому, что пытался делать, а также узнал и такие вещи, которые заставили меня изменить свой подход к работе.

Во время чтения этой книги у меня постоянно возникали реакции вроде «Я до этого не додумался!» и «О, теперь я понимаю!». Подозреваю, что помимо очевидной пользы от таких практических советов, как, например, применение дидактических карточек, книга даст толчок, окажет более глубокое влияние. Возможно, вы будете тщательнее обдумывать, куда именно в код нужно вставлять пустую строку. Возможно, измените задачи, предлагаемые новеньким в команде, либо скорректируете сроки выполнения этих задач. Возможно, мы станем по-другому объяснять концепции на платформе Stack Overflow.

Как бы там ни было, Фелина предоставила нам сокровищницу идей, которые мы обдумаем в рабочей памяти, а затем сохраним в долговременной — ведь мысли о мышлении вызывают привыкание!

Джон Скит,
менеджер по персоналу, Google

От автора

Примерно десять лет назад я начала обучать детей программированию. В тот момент я осознала, что практически ничего не знаю о том, как люди используют свой мозг для выполнения различных задач, особенно когда дело касается программирования. И хотя я изучала программирование в университете, ничто из университетского курса не готовило меня к тому, что я буду рассуждать о том, как думают программисты.

Если вы изучали информатику в вузе или постигали программирование самостоятельно, то вы, скорее всего, ничего не знаете о когнитивных функциях мозга. А значит, вы не в курсе, как можно тренировать свой мозг для упрощения процесса чтения и написания кода. Я тоже ничего об этом не ведала, но занятия с детьми погрузили меня в эту тему. Я узнала очень много о том, как мы думаем и как мы учимся. Эта книга — результат многолетних исследований, в ходе которых я прочитала множество книг, общалась с людьми и посещала выступления и конференции, посвященные обучению и мышлению.

Понимание, как работает мозг, представляет собой очень интересную тему, однако это понимание также важно и для программирования. Программирование считается одним из самых сложных когнитивных видов деятельности, ведь программист не только решает проблему абстрактным способом, но и обращается с программой. Все это требует колоссального уровня внимания, которого у большинства людей просто нет. Пропустили пробел? Ошибка. Неверно индексировали массив? Ошибка. Не разобрались в нюансах работы исходного кода? Ошибка.

В процессе программирования вы можете совершить множество ошибок. Из этой книги вы узнаете, что причиной многих ошибок являются когнитивные проблемы. Например, пропуск пробела может означать, что вы недостаточно хорошо овладели синтаксисом языка программирования. Ошибка индексации массива может указывать на то, что вы имеете неправильное представление о коде. Непонимание исходного кода говорит об отсутствии навыков чтения кода.

Цель этой книги проста — помочь вам понять, как мозг обрабатывает программный код. Понимание того, что делает ваш мозг с новой информацией, поможет повысить ваше мастерство, так как профессиональные программисты постоянно сталкиваются с чем-то новым. После того как мы узнаем о том, как код воздействует на мозг, мы рассмотрим методы улучшения навыков обработки кода.

Благодарности

Я очень хорошо понимаю, как мне повезло, что я смогла закончить книгу по любимой теме. Если бы не определенная череда событий, произошедших в определенные моменты, моя жизнь была бы совсем другой и я не написала бы эту книгу. Десятки самых различных встреч с замечательными людьми внесли неоценимый вклад в эту книгу и в мою работу. Имена наиболее значимых я хочу назвать.

Марлиз Альдеверелд (Marlies Aldewereld) впервые познакомила меня с программированием и изучением языков. Мэрилин Смит (Marileen Smit) подтянула меня в психологии, благодаря чему я написала эту книгу. Грег Уилсон (Greg Wilson) вернул в тренды тему обучения программированию. Питер Наббе (Peter Nabbe) и Роб Хогерворд (Rob Hoogerwoord) были для меня примером для подражания в сфере обучения. Штефан Ханенберг (Stefan Hanenberg) дал мне совет, определивший направленность моих исследований. Катя Мордаунт (Katja Mordaunt) открыла первый в мире клуб чтения кода. Размышления Йевеллина Фалко (Llewellyn Falco) о задачах привели в порядок мои мысли об обучении. Рико Хейберс (Rico Huijbers) был моей поддержкой в моменты, когда я совершенно не знала, что делать.

Я также хочу поблагодарить людей из издательства Manning: Марьям Басэ (Marjan Bace), Майка Стивенса (Mike Stephens), Тришу Лаувар (Tricia Louvar), Берта Бейтса (Bert Bates), Михаела Батинича (Mihaela Batinic), Бекки Рейнхарт (Becky Reinhart), Мелиссу Айс (Melissa Ice), Дженнифер Хаул (Jennifer Houle), Пола Уэллса (Paul Wells), Джерри Кюха (Jerry Kuch), Рейчел Хед (Rachel Head), Себастьена Портебуа (Sébastien Portebois), Кэндис Гилхули (Candace Gillhoolley), Криса Кауфмана (Chris Kaufmann), Матко Хрватина (Matko Hrvatin), Ивана Мартиновича (Ivan Martinovic), Бранко Латинчика (Branko Latincic) и Андрея Хофшустера (Andrej Hofšuster) — за то, что они взялись за работу, когда у меня были только наброски, и превратили их во что-то понятное, интересное и легко читаемое.

А также всех рецензентов: Адама Качмарека (Adam Kaczmarek), Адриана Бэйертца (Adriaan Beiertz), Алекса Риоса (Alex Rios), Ариэль Гаминьо (Ariel Gamiño), Бена МакНамара (Ben McNamara), Билла Митчелла (Bill Mitchell), Билли О'Каллагана (Billy O'Callaghan), Бруно Соннино (Bruno Sonnino), Чарльза Лэма (Charles Lam), Клаудию Мадертанер (Claudia Maderthaner), Клиффорда Тербера (Clifford Thurber), Даниэлу Запату Риско (Daniela Zapata Riesco), Эмануэля Оризи (Emanuele Origgi), Джорджа Онофрея (George Onofrei), Джорджа Томаса (George Thomas), Гилберто Такари (Gilberto Taccari), Хэйма Рамана (Haim Raman), Жауме Лопеса (Jaume

Lopez), Джозефа Перения (Joseph Perenia), Кента Спиллнера (Kent Spillner), Кимберли Уинстон-Джексон (Kimberly Winston-Jackson), Мануэля Гонзалеса (Manuel Gonzalez), Марцина Сэка (Marcin Sęk), Марка Харриса (Mark Harris), Мартина Кнудсена (Martin Knudsen), Майка Хьюитсона (Mike Hewitson), Майка Тэйлора (Mike Taylor), Орландо Мендеса Моралеса (Orlando Méndez Morales), Педро Серомено (Pedro Seromenho), Питера Моргана (Peter Morgan), Саманту Берк (Samantha Berk), Себастьяна Феллинга (Sebastian Felling), Себастьена Портебуа (Sébastien Portebois), Саймона Чоке (Simon Tschöke), Стефано Онгарейо (Stefano Ongarello), Томаса Хансена (Thomas Overby Hansen), Тима ван Дерзена (Tim van Deurzen), Туомо Каллиокоски (Tuomo Kalliokoski), Унникришнана Кумара (Unnikrishnan Kumar), Василе Бориса (Vasile Boris), Виктора Бека (Viktor Bek), Закари Бейела (Zachery Beyel) и Чжицзюнь Лю (Zhijun Liu). Ваши предложения помогли мне сделать эту книгу лучше!

О книге

Мозг программиста — это книга для программистов всех уровней, которые хотят разобраться, как работает их мозг и как можно улучшить свои навыки и стиль программирования. В данной книге будут показаны примеры кода на разных языках, включая JavaScript, Python и Java. Если вам удобно читать исходный код на языке, который вы никогда не встречали, то вам не нужны серьезные знания какого-либо из этих языков программирования.

Чтобы получить максимальную пользу от этой книги, желателен опыт работы в группе разработчиков или опыт работы над крупными программными системами, и привлечением людей в команду. Мы будем часто обращаться к подобным ситуациям, так что читатели, которые уже имеют опыт в данной области, получат намного больше ценной информации, чем другие. Человек, который может связать новую информацию с уже имеющимися у него знаниями и опытом, обучается продуктивнее и быстрее.

Несмотря на то что здесь представлены многие темы, связанные с когнитивистикой, эта книга предназначена в основном для программистов. В приведенных примерах мы всегда будем рассматривать работу мозга в контексте результатов исследований программирования и языков программирования.

Структура книги

Книга состоит из 13 глав, разделенных на четыре части. Главы следует читать по порядку, так как все они связаны. Каждая глава содержит прикладные примеры и упражнения, способствующие лучшему усвоению и закреплению полученного материала. В некоторых случаях для выполнения упражнения вам потребуется подыскать кодовую базу — так вы сможете работать с подходящим вам контекстом.

Вы также можете использовать полученные знания в вашей повседневной работе. Я думаю, что эту книгу можно изучать долгое время вот с каким планом: сначала вы изучаете главу, применяете упражнения из главы в своей практике программирования, а затем переходите к изучению следующих глав:

- В *главе 1* рассматриваются три когнитивных процесса, которые играют роль при программировании, а также рассматривается их связь с типами замешательства.
- В *главе 2* вы найдете советы о том, как быстро читать код и понимать его.
- В *главе 3* вы узнаете, как наиболее эффективно изучать синтаксис и концепции программирования.

- В *главе 4* вы узнаете, как читать сложный код.
- В *главе 5* показаны методы, которые помогут глубже понять незнакомый код.
- В *главе 6* рассматриваются методы, которые «прокачают» способность решать задачи программирования.
- В *главе 7* вы найдете советы о том, как избежать ошибок в коде и его понимании.
- В *главе 8* вы узнаете, как выбрать подходящее имя для переменной.
- *Глава 9* посвящена признакам кода «с запахом» и лежащим в их основе когнитивным причинам.
- В *главе 10* рассматриваются более сложные способы решения трудных задач.
- В *главе 11* описывается процесс кодирования и исследуется разнообразие задач программирования.
- В *главе 12* вы узнаете, как улучшить большие кодовые базы.
- В *главе 13* вы узнаете, как сделать процесс адаптации новых сотрудников менее болезненным.

В данной книге вы найдете множество примеров исходного кода: и в пронумерованных листингах, и просто в тексте. В обоих случаях при написании исходного кода используется моноширинный шрифт — так он бросается в глаза в обычном тексте. Иногда код может быть выделен **полужирным шрифтом** — так показаны изменения в коде, использовавшемся в предыдущих шагах в главе (например, при добавлении новой функции к уже существующей строке кода).

Во многих случаях первоначальный исходный код подвергается некоторым изменениям: например, мы добавили переносы или изменили отступы для того, чтобы код уместился на странице книги. Иногда этого было недостаточно, поэтому в листингах есть метка продолжения строки (➥). Кроме того, если исходный код описывается в тексте, то комментарии в исходном коде удаляются. Большинство листингов даются с аннотациями, в которых описаны основные идеи и концепции.

Дискуссионный форум liveBook

Приобретая книгу «Мозг программиста», вы получаете бесплатный доступ к частному веб-форуму Manning Publications, где вы можете оставить свои отзывы и комментарии о книге, а также задать технические вопросы и получить помощь от автора книги и других пользователей. Чтобы получить доступ к форуму, перейдите по ссылке **https://livebook.manning.com/book/the-programmers-brain/discussion**. Больше о форумах Manning и правилах поведения на форуме можно узнать по адресу **https://livebook.manning.com/#!/discussion**.

Нашей обязанностью было создание уголка, где читатели могли бы вести содержательный диалог и друг с другом, и с автором. Вам необязательно ответят, так как участие автора на форуме является добровольным (и не оплачивается). Тем не менее можно задавать сложные вопросы автору!

Об авторе

Доктор Фелина Херманс — доцент Лейденского университета в Нидерландах, где она проводит научные исследования языков программирования и методов обучения программированию. Она читает лекции в академии учителей Амстердамского свободного университета, специализируясь на дидактике компьютерных наук, а также преподает в средней школе Кралингена в Роттердаме.

Фелина является создателем языка программирования Hedy, предназначенного для начинающих программистов, а также ведет подкаст Software Engineering Radio, один из крупнейших интернет-подкастов о программном обеспечении.

Об обложке

Рисунок на обложке данной книги называется «Femme Sauvage du Canada», или «Коренная жительница Канады». Изображение взято из коллекции костюмов разных стран Жака Грассе де Сен-Совера (1757–1810) под названием «Costumes civils actuels de tous les peuples connus», изданной во Франции в 1788 году. Каждая иллюстрация была нарисована и раскрашена вручную. Показанное Грассе де Сен-Совером разнообразие напоминает нам о том, насколько города и регионы мира были культурно обособлены всего лишь 200 лет назад. Изолированные друг от друга, люди разговаривали на разных языках и диалектах. Встретив человека в городе или в деревне, по одной лишь одежде можно было определить, чем он занимается по жизни и где живет.

С тех пор манера одеваться сильно изменилась, а многие характерные признаки регионов просто исчезли. Теперь очень сложно отличить жителей разных континентов, не говоря уже о городах или странах. Может быть, мы променяли культурное разнообразие на разнообразную персональную жизнь — и уж точно на разнообразную и быстро развивающуюся технологическую жизнь.

И когда одну книгу о компьютерах с трудом получается отличить от другой, издательство Manning демонстрирует изобретательность компьютерного производства с помощью обложек книг, которые основаны на богатом разнообразии региональной жизни 200 лет назад, описанном в книге Грассе де Сен-Совера.

Часть I

Об улучшении навыков чтения кода

Чтение кода — основная часть программирования, однако не всегда даже опытные разработчики знают, как правильно это делать. Чтению кода не учат, и это редко практикуется, а незнакомый код сбивает с толку. Первые главы этой книги помогут вам понять, почему чтение кода — это сложно, и что можно сделать, чтобы читать код было проще.

1 Определение вашего типа замешательства при кодировании

В этой главе:

- вы узнаете о том, как можно запутаться при кодировании;
- сравните три когнитивных процесса, играющие роль при кодировании;
- поймете, как разные когнитивные процессы дополняют друг друга.

Замешательство — неотъемлемая часть программирования. Когда вы изучаете новый язык программирования, фреймворк или какую-либо концепцию, новые идеи могут напугать вас. Читая незнакомый код или собственный, но написанный вами много лет назад, вы можете не понять, для чего он нужен или почему он написан именно так. Когда вы приступаете к работе в новой сфере деятельности, то от новых терминов и сленга у вас может получиться мешанина в голове.

Конечно, нет ничего странного в том, что первое время вы будете испытывать замешательство — однако не стоит пребывать в этом состоянии больше необходимого. Эта глава поможет вам распознать и определить ваше замешательство. Возможно, вы никогда не задумывались об этом, но существует несколько способов впасть в замешательство. Замешательство от незнания значения какого-то понятия прикладной области отличается от замешательства при разборе шаг за шагом сложного алгоритма.

Разные типы замешательства относятся к разным когнитивным процессам. На нескольких примерах программного кода мы подробно рассмотрим три типа замешательства и объясним, что происходит в вашей голове.

К концу этой главы вы научитесь распознавать различные способы, которыми код может вызвать замешательство, и определять когнитивный процесс, происходящий в каждом случае в вашем мозге. В следующих главах вы узнаете, как улучшить эти когнитивные процессы.

1.1. Разные типы замешательства в коде

Любой незнакомый код сначала будет сбивать вас с толку, однако разные части одного кода вы будете воспринимать по-разному. Давайте рассмотрим это на трех разных примерах кода. Во всех трех примерах число `N` или `n` преобразуется в двоичное представление. Первая программа написана на APL, вторая — на Java, а третья — на BASIC.

Уделите несколько минут подробному разбору этих программ. На какие знания вы опираетесь? Чем ваши знания отличаются для каждой из программ? Возможно, сейчас вы не сможете объяснить то, что происходит в вашем мозге, но я думаю, что при чтении каждой из программ ваше восприятие отличается. В конце этой главы будут представлены выводы, с помощью которых вы сможете обсуждать различные когнитивные процессы, происходящие в вашей голове при чтении кода.

Пример в листинге 1.1 — это программа на APL, преобразующая число `n` в двоичное представление. Замешательство заключается в том, что вы можете не знать, что такое `T`. Если вы не являетесь математиком из 1960-х, то вы, скорее всего, никогда не работали на языке программирования APL. Этот язык был разработан специально для выполнения математических операций и на сегодняшний день практически не используется.

Листинг 1.1. Двоичное представление на APL

```
2 2 2 2 2 T n
```

Второй пример — это программа на Java, которая преобразует число `n` в двоичное представление (листинг 1.2). Замешательство может быть вызвано незнанием работы метода `toBinaryString()`.

Листинг 1.2. Двоичное представление на Java

```
public class BinaryCalculator {
    public static void main(Integer n) {
        System.out.println(Integer.toBinaryString(n));
    }
}
```

Третий пример — это программа на BASIC, которая преобразует число `N` в двоичное представление (листинг 1.3). Замешательство связано тем, что вы не можете отследить все шаги, выполняющиеся в ходе работы программы.

Листинг 1.3. Двоичное представление на BASIC

```
1 LET N2 = ABS (INT (N))
2 LET B$ = ""
3 FOR N1 = N2 TO 0 STEP 0
```

```
4     LET N2 = INT (N1 / 2)
5     LET B$ = STR$ (N1 - N2 * 2) + B$
6     LET N1 = N2
7 NEXT N1
8 PRINT B$
9 RETURN
```

1.1.1. Первый тип замешательства — недостаток знаний

Теперь давайте разберемся в том, что происходит, когда вы читаете эти программы. Сначала мы обратимся к программе на APL (листинг 1.4). Посмотрите, как программа преобразует число `n` в двоичное представление. Замешательство заключается здесь в том, что вы можете не знать, что означает `T`.

Листинг 1.4. Двоичное представление на APL

```
2 2 2 2 2 T n
```

Я думаю, что бóльшая часть читателей книги не знакомы с APL и не знают значение оператора `T`. Следовательно, замешательство в данном случае заключается в недостатке *знаний*.

1.1.2. Второй тип замешательства — недостаток информации

Во втором примере источник замешательства будет другим. Я предполагаю, что вы знакомы с программированием, и даже если вы не эксперт по Java, вы сможете найти соответствующие части программы. В этом примере показана программа на Java, преобразующая число `n` в двоичное представление (листинг 1.5). В данном случае замешательство может быть вызвано незнанием метода `toBinaryString()`.

Листинг 1.5. Двоичное представление на Java

```
public class BinaryCalculator {
    public static void mian(Integer n) {
        System.out.println(Integer.toBinaryString(n));
    }
}
```

По имени метода можно догадаться о его назначении. Однако, чтобы полностью понять, что делает код, вам нужно перейти к определению `toBinaryString()` в другом месте исходного кода и продолжить чтение с того места. Следовательно, проблемой является недостаток *информации*. Информации о методе `toBinaryString()` нет, ее необходимо найти в коде.

1.1.3. Третий тип замешательства — недостаток вычислительной мощности

В третьей программе, полагаясь на имена переменных и операций, вы можете предположить, что делает код. Но если вы хотите отследить каждый шаг, то ваш мозг не сможет справиться с этой задачей. Дело в том, что выполнение программы на языке BASIC, преобразующей число N в двоичное представление, включает в себя множество мелких шагов, которые вы не можете отследить. Если вам нужно полностью понимать все шаги, то лучше использовать мнемоническое представление информации, например промежуточные значения переменных, показанные на рис. 1.1.

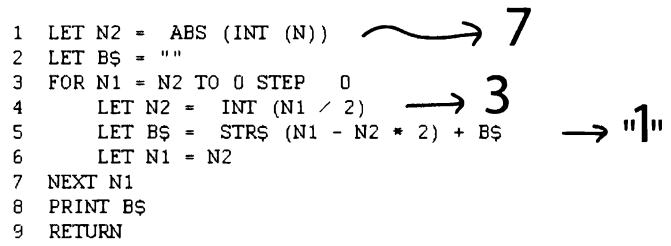


Рис. 1.1. Двоичное представление на BASIC

В данном случае замешательство связано с нехваткой *вычислительной мощности*. По правде говоря, одновременно удерживать в уме промежуточные значения переменных и нужные действия очень сложно. Если вы действительно хотите вычислить, что делает эта программа, то вам нужно взять ручку и листок, на котором вы запишете несколько промежуточных значений. Вы также можете написать промежуточные значения рядом со строками в коде, как это сделано в примере.

На примере этих трех программ мы увидели, что замешательство может иметь три разных источника. Во-первых, замешательство может быть вызвано незнанием языка программирования, алгоритма или предметной области. Однако замешательство может быть вызвано также отсутствием доступа ко всей информации, которая необходима для полного понимания кода. К тому же сейчас код часто использует различные библиотеки и модули, так что для понимания кода вы должны уметь выделять, собирать и усваивать новую информацию, а также запоминать весь алгоритм работы. Наконец, иногда код может быть просто очень сложным, и ваш мозг не сможет обработать его — причиной этого является нехватка вычислительной мощности.

Давайте теперь рассмотрим когнитивные процессы, связанные с каждым из этих трех типов замешательства.

1.2. Различные когнитивные процессы, влияющие на процесс кодирования

Давайте подробнее рассмотрим три когнитивных процесса, которые происходят в вашем мозге при просмотре трех примеров программ. Как указано ранее, различные виды замешательства связаны с различными видами когнитивных процессов. Все эти когнитивные процессы связаны с памятью. Подробную информацию об этом вы можете найти в последующих разделах главы.

Недостаток знаний означает, что в вашей *долговременной памяти*, где хранятся все ваши воспоминания, отсутствуют нужные факты. Недостаток информации представляет собой проблему для *кратковременной памяти*. Информация сначала хранится в кратковременной памяти, однако если вы изучаете несколько источников, то вы можете забыть уже прочитанную информацию. И наконец, если вы обрабатываете большое количество информации, то она отрицательно влияет на *рабочую память*, которая отвечает за мыслительную деятельность.

Итак:

- недостаток знаний = проблемы с долговременной памятью;
- недостаток информации = проблемы с кратковременной памятью;
- недостаток вычислительной мощности = проблемы с рабочей памятью.

Эти три когнитивных процесса играют большую роль не только при написании кода, но и во всей когнитивной деятельности, включая в себя (в случае программирования) написание кода, проектирование архитектуры системы и составление документации.

1.2.1. Долговременная память и программирование

Первый когнитивный процесс, использующийся при программировании, — это *долговременная память*. Долговременная память может хранить информацию в течение длительного времени. Множество людей помнят события, происходившие годы и десятилетия назад. Долговременная память играет большую роль во всем, что вы делаете: при завязывании шнурков активизируется мышечная память, которая автоматически запоминает последовательность действий, а при написании бинарного поиска вы помните буквенный алгоритм, синтаксис языка программирования и то, как печатать на клавиатуре. В *главе 3* подробно рассматривается использование долговременной памяти, включая эти формы запоминания и способы усиления данного когнитивного процесса.

В долговременной памяти хранится несколько видов информации, важной при программировании. Она может хранить воспоминания о том, как вы успешно применили какую-нибудь технику, значение ключевых слов Java, значение слов на английском языке или тот факт, что константа `maxint` в Java содержит значение `2147483647`.

Долговременную память можно сравнить с жестким диском компьютера, способным хранить информацию долгое время.

Программа на APL с точки зрения долговременной памяти

При чтении программы на APL вы больше опираетесь на долговременную память. Если вы знаете значение ключевого слова `т`, то при чтении этой программы вы извлечете его значение из долговременной памяти.

Программы на APL также требуют хорошего знания синтаксиса. Если вы не знаете, что обозначает ключевое слово `т`, то вам будет трудно понять саму программу. С другой стороны, если вы знаете, что оно обозначает *диадическое кодирование* функции, кодирующее значение в другую форму представления, то программу будет очень просто прочитать. Вам не нужно понимать значения слов и не нужно пошагово разбирать работу кода.

1.2.2. Кратковременная память и программирование

Другой когнитивный процесс, играющий большую роль в программировании, — это *кратковременная память*. Ваша кратковременная память используется для хранения поступающей информации в течение непродолжительного времени. Например, когда вам диктуют номер мобильного телефона во время разговора по телефону, то эта информация не сразу откладывается у вас в долговременной памяти. Сначала номер телефона откладывается в вашей кратковременной памяти, емкость которой ограниченна. Оценки емкости кратковременной памяти разнятся, однако все ученые сходятся во мнении, что емкость кратковременной памяти составляет всего несколько объектов. Ученые также сходятся во мнении, что емкость кратковременной памяти ограничена двенадцатью объектами.

Например, при чтении программы все ключевые слова, имена переменных и структуры данных хранятся в кратковременной памяти.

Программа на Java с точки зрения кратковременной памяти

В программе на Java самым важным когнитивным процессом является кратковременная память. Сначала вы просматриваете первую строку (листинг 1.6), из которой узнаете, что входной параметр `n` представлен целым числом. На данный момент вы еще не знаете, что делает данная функция, однако вы можете продолжить просматривать код, зная, что `n` — это число. Информация о том, что `n` представлен целым числом, какое-то время будет храниться в вашей кратковременной памяти. Затем вы переходите ко второй строке, в которой метод `toBinaryString()` определяет, что возвращает функция. Возможно, вы забудете эту функцию через день или даже через час, потому что как только вы поняли функцию, ваша кратковременная память удаляет эту информацию.

Листинг 1.6. Программа на Java, преобразующая число `n` в двоичное представление

```
    public static void mian(Int n) {
        System.out.println(Integer.toBinaryString(n));
    }
}
```

Замешательство может быть вызвано здесь незнанием принципов работы метода `toBinaryString()`.

Несмотря на то что в понимании данной программы бо̀льшую роль играет кратковременная память, пользователь также использует и долговременную память. По правде говоря, мы используем нашу долговременную память каждый раз, когда что-либо делаем. Читая программу на Java, вы также опираетесь и на долговременную память.

Я думаю, что большинство моих читателей знакомы с Java, так что вот вам пример: если вас попросят объяснить, что делает функция, то вы, скорее всего, знаете, что можно не учитывать ключевые слова *public class* и *public static void main*. Готова поспорить, что вы даже не заметили, что на самом деле метод называется mian, а не main.

Ваш мозг нашел короткий путь посредством присвоения имени, характеризующего смешение двух когнитивных процессов. Ваш мозг, опираясь на предыдущий опыт, хранящийся в долгосрочной памяти, решил использовать main вместо имени, которое вы сохранили в краткосрочной памяти. Данный пример опровергает мои слова о том, что эти два когнитивных процесса независимы друг от друга.

Если долговременную память можно сравнить с жестким диском компьютера, то кратковременную память можно сравнить с оперативной памятью компьютера или кешем, который используется для временного хранения информации.

1.2.3. Рабочая память и программирование

Третий когнитивный процесс, играющий роль в программировании, — это *рабочая память*. Кратковременная и долговременная память чаще всего используется для хранения информации. В случае кратковременной памяти информация хранится в течение короткого периода времени после ее прослушивания или прочтения, а в случае долговременной памяти информация хранится в течение длительного периода времени. Однако процесс мышления происходит не в кратковременной и не в долговременной, а в рабочей памяти. Именно здесь формируются мысли, идеи, решения. Если долговременную память можно сравнить с жестким диском, а кратковременную — с оперативной памятью, то рабочую память лучше всего сравнить с процессором мозга.

Программа на BASIC с точки зрения рабочей памяти

При чтении программ на языке BASIC вы используете долговременную память: например, когда вы вспоминаете значение ключевых слов `LET` и `RETURN`. Вы также используете кратковременную память, в которой храните некоторую новую информацию, например что начальное значение `B$` — пустая строка.

Однако, когда вы читаете программу на BASIC, ваш мозг выполняет множество операций. Мысленно вы выполняете код, чтобы понять, для чего он нужен. Этот процесс называется *трассировкой* — мысленной компиляцией и выполнением кода. Часть мозга, использующаяся для трассировки и других сложных когнитивных

задач, называется рабочей памятью. Вы можете сравнить ее с процессором компьютера, выполняющим вычисления.

При трассировке сложных программ у вас может появиться необходимость в записи значений переменных либо в коде программы, либо в отдельной таблице.

Если ваш мозг чувствует необходимость сохранить информацию на внешнем носителе, то это может быть сигналом того, что ваша рабочая память переполнена. В *главе 4* мы расскажем об информационной перегрузке и о том, как ее можно избежать.

1.3. Совместная работа когнитивных процессов

В предыдущих разделах я подробно описала три важных когнитивных процесса, связанных с программированием. Таким образом, долговременная память в течение длительного времени хранит полученную вами информацию, кратковременная память временно хранит информацию, которую вы только что прочитали или услышали, а рабочая память обрабатывает информацию и формирует новые мысли. Хотя я и описала их как отдельные процессы, они тесно связаны друг с другом. Давайте посмотрим, как именно они взаимосвязаны.

1.3.1. Краткое описание того, как когнитивные процессы взаимодействуют друг с другом

Как показано на рис. 1.2, когда вы начинаете думать, в вашем мозге активируются все три когнитивных процесса. Вы могли испытать все три когнитивных процесса при чтении фрагмента кода на Java (листинг 1.2). Какая-то информация сохрани-

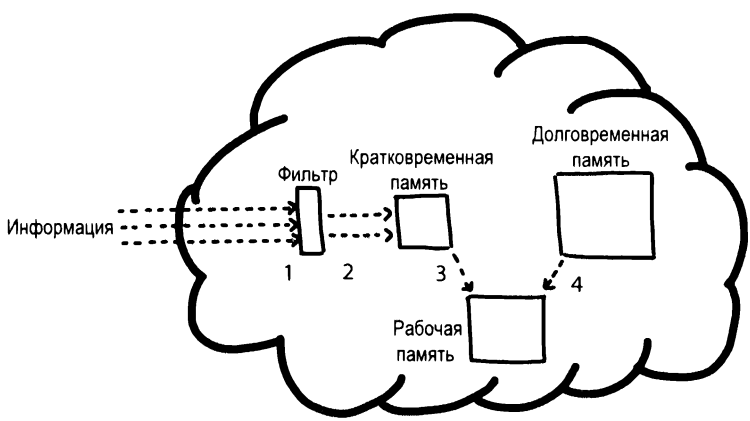


Рис. 1.2. Три когнитивных процесса, рассматриваемые в этой книге кратковременная память, долговременная память и рабочая память Стрелки с цифрой 1 обозначают информацию, поступающую в мозг Стрелки с цифрой 2 обозначают информацию, которая поступает в вашу кратковременную память Стрелка с цифрой 3 обозначает информацию, которая из кратковременной памяти переходит в рабочую, где она объединяется с информацией, поступающей из долговременной памяти (стрелка с цифрой 4) Рабочая память — это место, где обрабатывается информация, пока вы думаете о ней

лась в вашей кратковременной памяти — например то, что число `n` является целым числом. При этом ваш мозг извлек из долговременной памяти понятие целого числа, а рабочая память использовалась для понимания работы программы.

До сих пор мы рассматривали когнитивные процессы только на примере, когда вы знакомитесь с кодом и читаете его. Однако эти когнитивные процессы задействованы во многих других задачах, связанных с программированием.

1.3.2. Когнитивные процессы и программирование

Представьте, что вы читаете присланный клиентом отчет об ошибках. Вам кажется, что ошибка, о которой сообщил клиент, — это ошибка на единицу. Полученная информация поступает в мозг через органы чувств — глаза, если вы зрячий, или через уши, если вы используете экранный диктор. Для исправления ошибки вам нужно снова прочитать код, который вы написали несколько месяцев назад. Пока вы читаете код, ваша кратковременная память сохраняет поступающую информацию. В этот момент ваша долговременная память сигнализирует о том, что при написании кода вы использовали модель акторов. В вашей долговременной памяти также хранится фактическая информация о том, как можно исправить ошибку на единицу. Новая информация из отчетов об ошибке, хранящаяся в кратковременной памяти, а также ваши личные воспоминания из долговременной памяти составляют вашу рабочую память, с помощью которой вы думаете о возникшей проблеме.

УПРАЖНЕНИЕ 1.1. Чтобы попрактиковаться в понимании трех когнитивных процессов, я подготовила три программы. Однако в этот раз я не даю разъяснений, что делают фрагменты кода. Вы должны самостоятельно прочитать листинги программ и решить, для чего они нужны. Программы написаны на APL, Java и BASIC соответственно. Программы друг с другом не связаны и выполняют разные операции, поэтому, когда вы поймете первую программу, не стоит опираться на полученную информацию при чтении следующих программ.

Внимательно прочитайте программы и попытайтесь определить, что они делают. Обратите внимание на то, как вы думаете, и на алгоритм ваших действий. Для проведения самоанализа используйте вопросы из следующей таблицы.

	Листинг 1	Листинг 2	Листинг 3
Вы получаете знания из долговременной памяти?			
Если да, то какая это информация?			
Вы сохраняете информацию в кратковременной памяти?			
Какую именно информацию вы сохраняете?			
Какую информацию вы не запоминаете, считая ее неважной?			

(продолжение)

	Листинг 1	Листинг 2	Листинг 3
Ваша рабочая память может сразу же обработать некоторые части кода?			
Какие части кода сильнее всего загружают вашу рабочую память?			
Вы понимаете, почему эти части кода загружают вашу рабочую память?			

Листинг 1. Программа на APL

```
f • {ω≤1:ω ◊ (∇ ω-1)+∇ ω-2}
```

Что делает данный код? Какие когнитивные процессы были задействованы в процессе чтения кода?

Листинг 2. Программа на Java

```
public class Luhn {
    public static void main(String[] args) {
        System.out.println(luhnTest("49927398716"));
    }

    public static boolean luhnTest(String number){
        int s1 = 0, s2 = 0;
        String reverse = new StringBuffer(number).reverse().toString();
        for(int i = 0 ;i < reverse.length();i++){
            int digit = Character.digit(reverse.charAt(i), 10);
            if(i % 2 == 0){// это для нечетных чисел
                s1 += digit;
            }else{//прибавить цифру 2 * для 0-4, прибавить цифру 2 *- 9 для 5-9
                s2 += 2 * digit;
                if(digit >= 5){
                    s2 -= 9;
                }
            }
        }
        return (s1 + s2) % 10 == 0;
    }
}
```

Что делает данный код? Какие когнитивные процессы были задействованы в процессе чтения кода?

Листинг 3. Программа на BASIC

```
100 INPUT PROMPT "String: ":TX$
120 LET RES$=""
130 FOR I=LEN(TX$) TO 1 STEP-1
140   LET RES$=RES$&TX$(I)
150 NEXT
160 PRINT RES$
```

Что делает данный код? Какие когнитивные процессы были задействованы в процессе чтения кода?

Выводы

- Замешательство во время кодирования может возникнуть по трем причинам: недостаток знаний, отсутствие важной информации и недостаток вычислительной мощности мозга.
- При чтении или написании кода активируются три когнитивных процесса.
- Первый процесс — получение информации из долговременной памяти, где хранится, например, значение ключевых слов.
- Второй процесс — сохранение информации о текущей программе в кратковременной памяти, которая может временно хранить информацию, например имя метода или переменной.
- Третий процесс — это рабочая память. Именно здесь происходит обработка кода, например, когда вы решаете, что индекс слишком мал.
- Пока вы читаете код, все три когнитивных процесса остаются активными и дополняют друг друга. Например, если кратковременная память встречает имя переменной, допустим `n`, то ваш мозг начинает искать программы с такой переменной в долговременной памяти. А когда вы читаете неоднозначное слово, активируется рабочая память — ваш мозг пытается определить правильное значение слова в данном контексте.

2 Скорочтение кода

В этой главе:

- вы узнаете, почему даже опытному программисту сложно быстро читать код;
- узнаете, как мозг выполняет анализ новой информации и разбивает ее на знакомые части;
- узнаете, как долговременная и кратковременная память работает при анализе такой информации, как слова или код;
- узнаете о роли иконической памяти при обработке кода;
- узнаете, как запоминание кода можно использовать как инструмент для (само-)диагностики уровня кодирования;
- попрактикуетесь в написании кода, который другим людям будет просто прочитать.

В *главе 1* были представлены три когнитивных процесса, которые играют важную роль в программировании и чтении кода. Первым рассмотренным нами когнитивным процессом была долговременная память, которую можно сравнить с жестким диском, на который записываются воспоминания и факты. Вторым когнитивным процессом была кратковременная память, которую можно сравнить с оперативной памятью, хранящей информацию, которая поступает на короткое время. И наконец, есть рабочая память, которая является процессором и обрабатывает информацию, поступающую от долговременной и кратковременной памяти.

В этой главе все внимание будет уделено чтению кода. Бóльшую часть времени программист занимается именно чтением кода. Исследования показывают, что примерно 60% времени программистов тратится на *понимание* кода, а не на его *написание*[1]. Следовательно, повышение скорости чтения кода без потери качества поможет вам улучшить ваши навыки программирования.

---

[1] См. «Measuring Program Comprehension: A Large-Scale Field Study with Professionals», Син Ся и др. (2017), **https://ieeexplore.ieee.org/abstract/document/7997917**.

В предыдущей главе вы узнали, что кратковременная память — это первое место, куда попадает вся новая информация. В этой главе мы поможем вам понять, почему же так сложно обрабатывать большой объем информации, которая содержится в коде. Если вы узнаете, что происходит с вашим мозгом при быстром чтении кода, то вам будет проще контролировать то, как и что вы понимаете. После я покажу приемы, которые помогут вам улучшить ваши навыки работы с кодом. Например, вы будете тренироваться быстро читать отдельные фрагменты кода. К концу главы вы узнаете, почему чтение кода вызывает столько трудностей. Вы узнаете, как быстро читать код, а также я расскажу о некоторых приемах, с помощью которых можно улучшить навык чтения кода.

2.1. Быстрое чтение кода

В книге Харольда Абельсона (Harold Abelson), Джеральда Джея Сассмана (Gerald Jay Sussman) и Джулии Сассман (Julie Sussman) «*Структура и интерпретация компьютерных программ*» (MIT Press, 1996) есть известная фраза: «Программы должны писаться для того, чтобы их читали люди, и лишь во вторую очередь для выполнения машиной». В какой-то степени это является правдой, однако на самом деле программисты больше пишут код, а не читают его.

Это начинается еще в самом начале пути каждого программиста. Обучение программированию нацелено на изготовление кода. Я уверена, что, когда вы учились программированию — не важно, в колледже, на работе или в летнем IT-лагере (буткемпе), — больше всего времени вы уделяли именно созданию кода. Целью упражнений было найти и исправить ошибки кода. Могу предположить, что упражнений на чтение кода не было совсем. Из-за отсутствия практики чтения кода программисту намного сложнее работать с кодом. Эта глава поможет вам улучшить навыки чтения и понимания кода.

Вы будете читать код по разным причинам: чтобы добавить функцию, найти ошибку или расширить свое представление обо всей системе. В любом случае вы ищете в коде конкретную информацию, например место, куда будет правильнее всего вставить новую функцию, или место, где появляется ошибка, или место последнего редактирования кода, или то, как применен определенный метод.

Когда вы научитесь быстро находить нужную информацию, вы заметите, что вы стали реже возвращаться к самому коду. Вы также заметите то, что вам не нужно перечитывать весь код для того, чтобы найти необходимую информацию. А время, которое вы сэкономите на чтении кода и поиске в нем информации, можно уделить исправлению ошибок или добавлению новых функций.

В предыдущей главе я просила вас читать программы на трех разных языках программирования — так вы получили представление о работе трех частей мозга. Чтобы глубже понять роль кратковременной памяти, посмотрите следующую программу на Java, которая приводит в действие метод сортировки вставками (листинг 2.1). Времени дается не больше трех минут. По истечении времени закройте код листом бумаги или рукой.

Не открывая код, постарайтесь воспроизвести по памяти код как можно точнее.

Листинг 2.1. Программа на Java, выполняющая сортировку вставками

```
public class InsertionSort {
  public static void main (String [] args) {
    int [] array = {45,12,85,32,89,39,69,44,42,1,6,8};
    int temp;
    for (int i = 1; i < array.length; i++) {
      for (int j = i; j > 0; j--) {
        if (array[j] < array [j - 1]) {
          temp = array[j];
          array[j] = array[j - 1];
          array[j - 1] = temp;
        }
      }
    }
    for (int i = 0; i < array.length; i++) {
      System.out.println(array[i]);
    }
  }
}
```

2.1.1. Что только что происходило в вашем мозге

Когда вы воспроизводили по памяти программу сортировки на Java, вы использовали как кратковременную, так и долговременную память. Подробнее весь процесс показан на рис. 2.1.

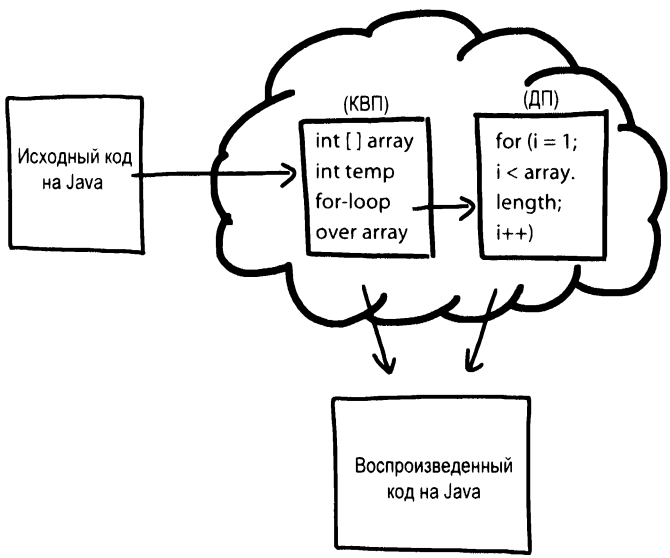


Рис. 2.1. Иллюстрация когнитивных процессов, которые происходят при запоминании кода Часть кода сохраняется в кратковременной памяти (например, имена переменных или значения переменных), а для другой части используются знания, которые хранятся в вашей долговременной памяти (например, синтаксис цикла `for`)

Ваша кратковременная память сохраняла часть информации, которую вы только что прочитали. Долговременная память дополняла полученную информацию двумя способами. Во-первых, при чтении кода вы могли полагаться на знание синтаксиса Java. Наверное, вы вспомнили, что «цикл `for` для массива», что, судя по тому, что у вас хранится в долговременной памяти, эквивалентно `for (int i = 0; i < array.length; i++)`. Возможно, вы также вспомнили «вывод всех элементов массива», что на языке Java соответствует `for (i = 0; i < array.length; i++) {System.out.println(array[i])}`.

Во-вторых, вы могли полагаться на знание того, что код выполняет сортировку вставками. Возможно, это помогло вам заполнить некоторые пробелы, возникшие при запоминании кода. Например, вы могли не запомнить, что два значения массива меняются местами, однако т. к. вы знакомы с сортировкой вставками, то вы знали, что в каком-то фрагменте кода должна произойти замена.

2.1.2. Перепроверка воспроизведенного кода

Прежде чем мы углубимся в понимание когнитивных процессов, давайте еще раз посмотрим на воспроизведенный вами код. Подумайте, какие части кода вы воспроизвели с помощью кратковременной памяти, а какие — с помощью долговременной. Пример воспроизведенного кода с комментариями представлен на рис. 2.2.

```
public class InsertionSort {
  public static void main (String [] args) {
   int [] array = {45,12,…}
   int temp;
   for (int i = 1; i < array.length; i++)   {
    for (int j = i; j > 0; j--)   {
     if (array[j] < array [j - 1])   {
      // swap j with j - 1
      temp = array[j];
      array[j] = array[j - 1];
      array[j - 1] = temp;
     }
    }
   }
   //print array
   for (int i = 0; i < array.length; i++)   {
     System.out.println(array[i]);
   }
  }
}
```

Рис. 2.2. Пример кода из листинга 2 1, воспроизведенного по памяти опытным программистом на языке Java, с пометками, обозначающими когнитивные процессы. Выделенные темным цветом части кода представляют собой информацию, хранящуюся в кратковременной памяти, а части кода, выделенные светлым цветом, представляют собой информацию, поступающую из долговременной памяти. Обратите внимание на то, что воспроизведенный код больше изначального кода из листинга, например добавлены комментарии

Какая информация извлекается из долговременной памяти, зависит, конечно, от того, что в ней хранится. Если человек мало работал с Java, то он извлечет из своей долговременной памяти мало информации, поэтому его воспроизведенный код

будет выглядеть иначе. Также обратите внимание на то, что в примере присутствуют комментарии, которых не было в исходном коде. В ходе тестирования программистов я заметила, что некоторые из них при воспроизведении кода добавляют комментарии, которые облегчают запоминание кода. Например, сначала программист пишет «выводим содержимое массива», а затем уже пишет сам код. А вы воспроизводили код так же?

Чаще всего комментарии используются для описания уже написанного кода, однако в примере вы можете видеть, что комментарии можно использовать и как вспомогательное средство для написания кода. В следующих главах роль комментариев в коде будет рассмотрена более подробно.

Вторая попытка воспроизведения кода

В *главе 1* я объяснила, как при чтении кода взаимодействуют между собой долговременная и кратковременная память. Вы только что более подробно ознакомились с их взаимодействием, воспроизведя части кода программы сортировки вставками на Java с помощью информации, хранящейся в вашей долговременной памяти.

Чтобы лучше понять, как сильно вы полагаетесь на долговременную память, давайте выполним еще одно упражнение. Это упражнение ничем не отличается от предыдущего: посмотрите на код программы не больше трех минут, затем закройте код и попытайтесь воспроизвести код по памяти.

Код на языке Java в листинге 2.2 следует использовать как упражнение для тренировки памяти. Просмотрите код в течение трех минут, а затем воспроизведите его по памяти.

Листинг 2.2. Код на Java для запоминания

```
void execute(int x[]){
    int b = x.length;

    for (int v = b / 2 - 1; v >= 0; v--)
        func(x, b, v);

    // Извлекаем элементы по одному
    for (int l = b-1; l > 0; l--)
    {
        // Перемещаем текущий элемент в конец
        int temp = x[0];
        x[0] = x[l];
        x[l] = temp;
        func (x, l, 0);
    }
}
```

2.1.3. Перепроверка воспроизведенного

Даже ничего не зная о вас или о вашем опыте работы с Java, я могу с уверенностью сказать, что вторую программу было сложнее запомнить. И этому есть несколько причин. Во-первых, вы не знаете, что делает этот код. Из этого следует, что вы не можете опираться на хранящиеся в долговременной памяти знания.

Во-вторых, я намеренно выбрала «странные» имена для переменных, например `b` и `l` для итераторов циклов `for`. Незнакомые имена сбивают с толку, затрудняют распознавание и запоминание конструкций языка. Особенно сбивает с толку имя `l`, т. к. визуально буква l очень похожа на цифру 1.

2.1.4. Почему читать незнакомый код так сложно

Как нам показал предыдущий пример, воспроизвести по памяти код не всегда легко. Почему так сложно запомнить код? Главная причина — ограниченная вместительность кратковременной памяти.

Как бы вы ни старались, вы не сможете сохранить всю информацию из кода в кратковременной памяти. Как вы узнали из *главы 1*, кратковременная память хранит информацию, которую вы слышите или читаете, в течение короткого периода времени. И когда я говорю «короткий», то я и имею в виду короткий! Согласно исследованиям, мы не можем сохранять информацию больше чем на 30 с. Через 30 с вам придется сохранить нужную информацию в долговременной памяти, иначе она будет потеряна навсегда. Представьте, что кто-то по телефону диктует вам номер телефона, а вам некуда его записать. Если в ближайшее время вы не запишите номер телефона, то вы его забудете.

Время, в течение которого вы можете запомнить информацию, является не единственной проблемой кратковременной памяти. Второй проблемой является ее вместимость.

Как и в компьютере, размер вашей долговременной памяти намного больше размера кратковременной памяти. Однако когда мы представляем оперативную память компьютера, то у нас сразу возникают ассоциации с несколькими гигабайтами памяти. К сожалению, кратковременная память намного меньше — она ограничивается несколькими «ячейками». Джордж Миллер, один из самых влиятельных людей в сфере когнитивной науки прошлого века, описал данное явление в 1956 году в статье под названием «Магическое число семь плюс-минус два».

Более поздние исследования показывают, что размер кратковременной памяти еще меньше — она может вместить в себя от двух до шести элементов. Этот диапазон актуален практически для всех людей, и ученые до сих пор не нашли способ увеличить размер кратковременной памяти. И разве не чудо то, что люди не ограничены в своих действиях и могут делать все, что угодно, имея не больше одного байта памяти?

Чтобы восполнить недостаток места, ваша кратковременная память взаимодействует с долговременной — благодаря этому вы можете что-то вспомнить. В следующем разделе подробно описывается то, как кратковременная память взаимодействует с долговременной и обходит ограничения по размеру.

2.2. Преодоление лимитов памяти

В предыдущем разделе вы узнали об ограничениях кратковременной памяти. Вы испытали их на собственном опыте, когда пытались запомнить фрагмент кода. Однако, скорее всего, вы запомнили несколько больше, чем просто шесть элементов из кода. Разве это не противоречит тому, что память может вместить в себя только шесть элементов?

Способность кратковременной памяти запоминать только шесть элементов распространяется на любую когнитивную задачу, а не только на чтение кода. Так как же люди могут вообще что-то делать с такой маленькой памятью? Например, как вы можете читать это предложение?

Согласно теории Миллера, как только человек прочитает шесть букв, он начнет забывать первые буквы. Очевидно, что человек может запомнить больше шести букв, но как это возможно? Чтобы разобраться с тем, почему незнакомый код так тяжело читать, а также узнать чуть больше о кратковременной памяти, давайте рассмотрим один важный шахматный эксперимент.

2.2.1. Сила чанков

Теория чанков впервые была описана нидерландским математиком Адрианом де Гроотом (Adrian de Groot). Он был аспирантом по математике и шахматистом. И его постоянно волновал вопрос: почему один человек может стать великим шахматистом, в то время как другие игроки будут оставаться на среднем уровне всю жизнь? Для исследования данного вопроса де Гроот провел два разных эксперимента.

В первом эксперименте (рис. 2.3) де Гроот на несколько секунд показывал шахматистам расстановку. По истечении определенного времени де Гроот закрывал фигуры, а участники эксперимента должны были восстановить шахматную расстановку по памяти. По сути, это задание очень похоже на упражнение, которое я давала вам в начале этой главы. Де Гроот был заинтересован не только в способности каждого человека запоминать расположение фигур на шахматной доске; он специально разделил всех людей на две группы и сравнивал их. В первую группу входили любители шахмат, а во вторую — профессиональные шахматисты. Сравнивая результаты обычных шахматистов с профессионалами, де Гроот обнаружил, что опытным шахматистам намного проще воссоздать по памяти шахматную расстановку.

После этого эксперимента де Гроот пришел к выводу, что профессионалы превосходят обычных, посредственных игроков, т. к. кратковременная память профессионалов вмещает больше информации. Он также предположил, что причиной большого объема кратковременной памяти профессиональных шахматистов является то, что профессиональные игроки были прежде всего экспертами. Соответственно, они могли запомнить позиции большего количества шахматных фигур, что, собственно, и позволяло им лучше играть в шахматы.

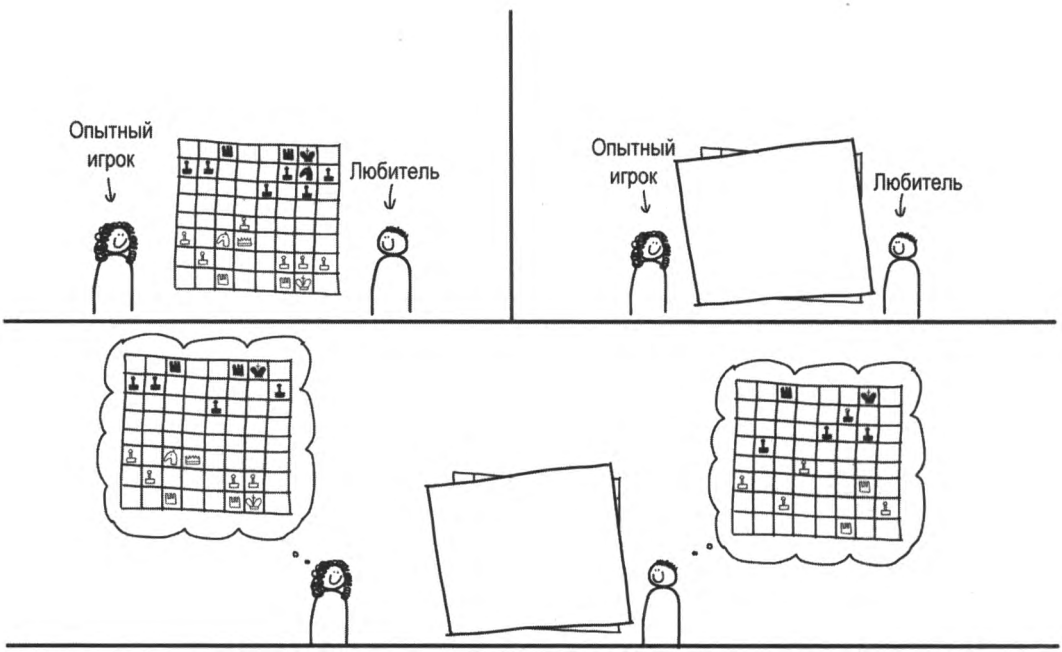


Рис. 2.3. Первый шахматный эксперимент де Гроота, в котором профессионалов и обычных шахматистов попросили воспроизвести шахматную расстановку. Опытные игроки смогли вспомнить позиции большего количества фигур, чем любители

Однако де Гроот был не совсем уверен в своих выводах, поэтому он провел еще один эксперимент. Второй эксперимент был похож на первый: профессиональных и обычных игроков попросили запомнить расположение шахматных фигур после быстрого просмотра шахматной доски. Разница заключалась в шахматных расстановках. Вместо того, чтобы показать участникам настоящую шахматную расстановку, он показал ситуацию, когда фигуры были расставлены в случайном порядке — так, как в реальности они никогда не стояли бы. Затем де Гроот снова сравнил результаты профессиональных и обычных шахматистов. Результаты второго эксперимента отличались от предыдущих: обе группы показали себя одинаково плохо!

Разнящиеся результаты первого и второго экспериментов заставили де Гроота разобраться с тем, как именно обе группы запоминали шахматные расстановки. Оказалось, что в обоих экспериментах обычные шахматисты запоминали расстановку фигуру за фигурой: «ладья на A7, пешка на B5, король на C8» и т. д.

Однако в первом эксперименте профессиональные шахматисты придерживались другой стратегии. Профессионалы полагались на информацию, хранящуюся в их долговременной памяти. Например, они могли запомнить, что это «сицилианская защита, но один конь на две клетки левее». Такое запоминание возможно только в том случае, если вы знаете, какие фигуры используются в сицилианской защите. А эти знания хранятся в долговременной памяти. Профессиональные шахматисты, запоминавшие расстановку фигур, заняли всего четыре ячейки рабочей памяти (сицилианская защита, конь, 2, слева). Как вы знаете, кратковременная память может

хранить от двух до шести элементов, поэтому запоминание четырех позиций вполне возможно.

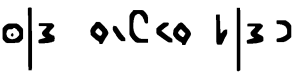
Некоторые профессиональные игроки могли также связать расстановку с личной игрой или играми, которые они видели или о которых читали. Они могли ассоциировать расположение шахмат с «игрой в ту мартовскую дождливую субботу с Бетси, но рокировка была слева». Эта информация также хранится в долговременной памяти. Запоминание расстановки путем вспоминания предыдущего опыта также занимает в кратковременной памяти всего несколько ячеек.

Однако обычные игроки, которые пытались запомнить все шахматы на доске, быстро заполнили кратковременную память. Они не могли сгруппировать все фигуры так, как это делали профессионалы. Это объясняет причину, по которой обычные шахматисты в первом эксперименте показали результаты хуже, чем профессиональные игроки: когда в их кратковременной памяти не осталось места, они перестали запоминать расположение остальных фигур.

Де Гроот назвал группы, в которые люди объединяли информацию, *чанками* (англ. *chunks*). Например, «силицианскую защиту» он принимал за один чанк, который может поместиться в одну ячейку кратковременной памяти. Теория чанков также дает логичное объяснение тому, почему во втором эксперименте обе группы шахматистов показали одинаковые результаты. В хаотичной шахматной расстановке профессиональные шахматисты не могли полагаться на свой опыт и долговременную память, поэтому они не смогли разделить все фигуры на большие группы.

УПРАЖНЕНИЕ 2.1. Возможно, эксперименты де Гроота вас убедили, но лучше всего испытать это на себе!

Посмотрите на это предложение пять секунд и постарайтесь запомнить его:



Как много вы запомнили?

УПРАЖНЕНИЕ 2.2. Теперь попытайтесь запомнить это предложение, посмотрев на него пять секунд:

abk mrtpi qbar

Думаю, второе предложение далось вам легче первого. Все потому, что второе предложение содержит в себе знакомые вам буквы. Трудно поверить, но оба предложения одинаковой длины: они состоят из трех слов, двенадцати букв и девяти неповторяющихся букв.

УПРАЖНЕНИЕ 2.3. Давайте попробуем еще раз. Это предложение также состоит из трех слов и девяти неповторяющихся букв. Посмотрите на предложение пять секунд и попытайтесь его запомнить:

cat loves cake

Третье предложение намного проще предыдущих, да? А все потому, что вы можете разбить буквы на слова, благодаря чему вам просто нужно будет запомнить три чанка: cat, love и cake. Разбив все предложение на три элемента, вы не переполняете свою кратковременную память, в то время как количество элементов в первых двух примерах превышало допустимый лимит вашей кратковременной памяти.

Чанки кода

До сих пор в главе вы видели, что чем больше опыта и информации у вас есть по определенной теме, тем проще вам разделить нужную информацию на чанки. Профессиональные шахматисты помнят множество игровых ситуаций, поэтому им намного проще запомнить расположение фигур. На примере предыдущих упражнений мы выяснили, что слова вспомнить гораздо проще, чем незнакомые символы и отдельные буквы. Слова легче запоминать, потому что вы можете извлечь их значение из долговременной памяти.

Полученные данные о том, что запомнить какую-то информацию намного проще, если в долговременной памяти уже есть знания по этой теме, относятся и к программированию. В оставшейся части главы мы рассмотрим результаты исследований по программированию и чанкам кода. После этого мы более подробно рассмотрим то, как лучше делить код на чанки и как писать код, который легче разделить на чанки.

2.2.2. Опытные программисты запоминают код лучше начинающих программистов

Исследования де Гроота оказали большое влияние на когнитивную науку. Его эксперименты побудили исследователей в области информатики провести похожие эксперименты и сравнить результаты.

Например, в 1981 году исследователь из Bells Labs Кэтрин МакКитен (Katherine McKeithen) попыталась повторить эксперименты де Гроота с программистами[2]. Кэтрин и ее коллеги показали программы на языке Алгол из 30 строк 53 людям: начинающим, средней руки и опытным программистам. Некоторые программы были реальными, как и в первом эксперименте де Гроота, в котором он просил участников запомнить реальную шахматную ситуацию. Код других программ был набором случайных строк, как и во втором эксперименте де Гроота, где шахматные фигуры были расположены в случайных позициях. Участники рассматривали программы на Алголе в течение двух минут, а затем должны были воспроизвести их по памяти.

Результаты эксперимента МакКитен были схожи с результатами де Гроота: лучше всего запоминали реальные программы опытные программисты, а программисты

[2] «Knowledge Organization and Skill Differences in Computer Programmers», Кэтрин МакКитен и др. (1981), **http://mng.bz/YA5a**

средней руки запоминали программы лучше начинающих. Результаты запоминания вымышленной программы между тремя группами практически не отличались.

Главный вывод этих экспериментов заключается в том, что новички могут обработать намного меньше кода (рис. 2.4), чем опытные программисты. Это важно помнить, когда вы принимаете в команду нового сотрудника или когда вы самостоятельно учите язык программирования.

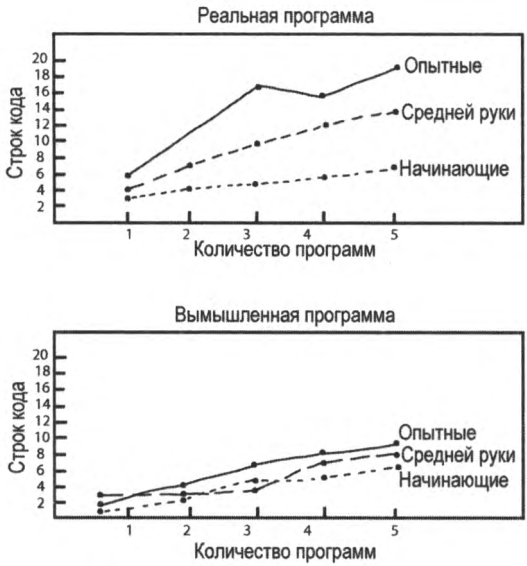


Рис. 2.4. Количество строк кода, которые новички, средней руки и опытные программисты могут запомнить в экспериментах МакКитен и др На верхнем изображении показаны результаты участников в первом эксперименте, где опытные программисты показывают лучшие результаты На нижнем изображении показаны результаты участников во втором эксперименте, где результаты всех трех групп практически одинаковые

Даже самый способный программист, встретив новый язык или предметную область, будет продираться через незнакомые ключевые слова, структуры и концепции, пока они не сохранятся в его долговременной памяти. В следующей главе вы узнаете, как можно быстро сохранить знания в долговременной памяти.

2.3. Вы видите намного больше кода, чем можете прочитать

Прежде чем мы перейдем к глубокому изучению кратковременной памяти, я хочу рассказать о том, что происходит, когда информация попадает в мозг. Существует стадия, через которую проходит вся информация, прежде чем она достигает кратковременной памяти. Эта стадия называется *сенсорной памятью*.

По нашей аналогии сенсорную память можно сравнить с буфером ввода-вывода, который взаимодействует с устройством ввода, например мышью или клавиатурой. Информация, отправляемая с этих устройств, временно хранится в буфере ввода-

вывода. Так же действует и наша сенсорная память: она временно хранит зрительную, слуховую или осязательную информацию. Каждое из пяти чувств — зрение, слух, осязание, обоняние, вкус — имеет свое место в сенсорной памяти. Не все они нам интересны в контексте программирования, поэтому в этой главе мы ограничимся лишь одним, работающим со зрением, — *иконической памятью*.

2.3.1. Иконическая память

Прежде чем информация попадает в кратковременную память, она через органы чувств поступает в сенсорную память. При чтении кода информация поступает через ваши глаза, а после на какое-то время она сохраняется в иконической памяти.

Лучший способ проиллюстрировать иконическую память — это представить, что сейчас канун Нового года, а в руках у вас бенгальский огонь. Если вы быстро начнете двигать рукой с бенгальским огнем, то сможете написать букву. Может, вы никогда об этом не задумывались, как такое возможно, но именно иконическая память позволяет увидеть световые фигуры. Иконическая память на время сохраняет визуальные стимулы, созданные увиденным изображением. Другой пример: закройте глаза. Вы «увидите» форму страницы — это тоже иконическая память.

Перед тем как мы поговорим о том, как можно использовать иконическую память при чтении кода, давайте посмотрим, что мы о ней знаем. Одним из первых исследователей иконической памяти был американский психолог Джордж Сперлинг (George Sperling), занимавшийся исследованием сенсорной памяти в 1960-х годах. В его самом известном исследовании[3] трем участникам показывалась карта размером 3×3 или 3×4 буквы. Эту карту можно сравнить с таблицей для проверки зрения, только на карте Сперлинга все буквы были одинакового размера (рис. 2.5).

Участникам показывали изображение в течение одной двадцатой секунды (0,05 с или 50 мс), после чего их просили повторить случайно выбранную линию, например верхнюю строку или левый столбец.

F	C	M	B
Q	P	V	D
L	X	T	A

Рис. 2.5. Пример карты Сперлинга, которую участники должны были запомнить

За 50 мс невозможно нормально прочитать буквы, потому что время реакции человеческого глаза составляет около 200 мс (или одна пятая секунды). Тем не менее участники данного эксперимента смогли вспомнить все буквы в случайной строке или столбце карты примерно в 75% случаев. Согласно этим результатам, участники запомнили бóльшую часть или все 9 или 12 букв, что намного больше того количества, которое может поместиться в кратковременной памяти.

---

[3] «The Information Available in Brief Visual Presentations», Дж. Сперлинг (1960), **http://mng.bz/O1ao**.

И дело не в том, что у всех участников была идеальная память. Когда участников попросили назвать все буквы, то они показали результаты намного хуже, чем тогда, когда их попросили назвать определенную строку или столбец. Обычно участники запоминали около половины. Когда Стерлинг проводил свои эксперименты, уже было известно, что кратковременная память может вмещать в себя до шести элементов. Однако тот факт, что многие участники запоминали все буквы, показал, что вся карта сохранилась не в кратковременной памяти с ее ограниченным размером, а где-то еще. Это место, куда вся информация поступает из визуального восприятия, Сперлинг назвал иконической памятью. Как показали его эксперименты, не вся информация, хранящаяся в иконической памяти, может быть обработана кратковременной памятью.

Иконическая память и код

Как вы уже узнали, все, что вы читаете, сначала сохраняется в иконической памяти. Однако не все, что там хранится, может быть обработано кратковременной памятью. Поэтому при чтении кода вы должны обращать внимание на то, какую информацию вы можете обработать. Однако этот выбор проходит неосознанно — при чтении вы совершенно случайно можете упустить какие-то детали. Это означает, что чисто теоретически вы можете запомнить намного больше информации о коде, чем можете обработать.

Вы можете использовать эту информацию для того, чтобы улучшить свой навык чтения кода: сначала быстро посмотрите на код, а затем подумайте, что вы увидели и запомнили. Упражнение «код с первого взгляда» поможет вам получить представление о коде.

УПРАЖНЕНИЕ 2.4. Выберите фрагмент кода, который кажется вам знакомым. Вы можете взять фрагмент из собственной кодовой базы или простой фрагмент с GitHub. Язык программирования не имеет значения. Рекомендуемый размер — полстраницы. Рекомендуется распечатать код на бумаге.

Несколько секунд посмотрите на код, затем закройте его и ответьте на следующие вопросы:

- ❒ Какова структура кода?
  - Код сплошной или вложенный?
  - Есть ли выделяющиеся строки?
- ❒ Как пробелы влияют на структуру кода?
  - Есть ли в коде пропуски?
  - Есть ли в коде массивные куски?

2.3.2. Это не то, что вы помните; это то, как вы запоминаете

Когда вы пытаетесь воспроизвести только что прочитанный код, изучение строк кода может стать отличным диагностическим инструментом, который поможет вам

понять собственное (не)понимание. Однако это не только *что* вы можете запомнить — *порядок*, в котором вы запоминаете код, также может быть инструментом для понимания.

Исследователи, повторившие шахматный эксперимент де Гроота с использованием программ на языке Алгол, провели еще один эксперимент, который позволил узнать больше о разделении информации на чанки[4]. Во втором эксперименте начинающих, средней руки и опытных программистов обучали 21 ключевому слову Алгола, например `IF`, `TRUE` и `END`.

Все ключевые слова, которые участники учили в ходе эксперимента, указаны на рис. 2.6. Вы можете попробовать выучить все ключевые слова самостоятельно.

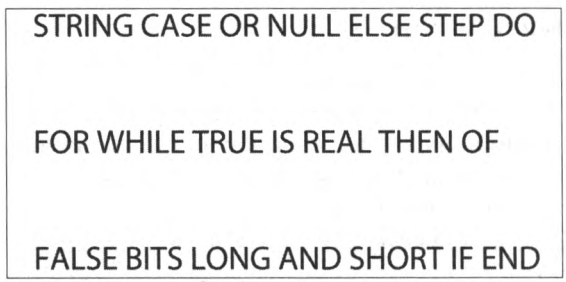


Рис. 2.6. 21 ключевое слово на языке Алгол, которое в своем эксперименте использовала МакКитен Начинающим, средней руки и опытным программистам нужно было выучить все эти слова

Когда участники смогли запомнить и правильно повторить все ключевые слова, исследователи попросили участников составить список всех ключевых слов. Если вы тоже запомнили их, то запишите все ключевые слова и сравните свой результат с результатом участников эксперимента.

Исходя из порядка, в котором участники повторяли ключевые слова, МакКитен смогла понять, как участники связали ключевые слова. Эксперимент показал, что начинающие программисты сгруппировали ключевые слова Алгола не так, как опытные программисты. Например, для запоминания они составляли предложения вроде «TRUE IS REAL THEN FALSE». Опытные же программисты при группировке ключевых слов использовали свои знания программирования. Например, они объединили ключевые слова TRUE и FALSE, а также IF, THEN и ELSE. Этот эксперимент еще раз подтвердил то, что опытные программисты работают с кодом не так, как начинающие.

Как написать код, который можно разделить на чанки

Выполнив несколько раз задание на запоминание и разделение кода на чанки, на интуитивном уровне вы начинаете понимать, код какого типа можно разделить на чанки. Из шахматного эксперимента де Гроота мы знаем, что стандартные или

[4] Результаты представлены в статье «Knowledge Organization and Skill Differences in Computer Programmers». МакКитен и др.

предсказуемые расстановки фигур, как, например, известные дебюты, упрощают разделение на чанки. Итак, если ваша цель — создать шахматную расстановку, которую будет очень просто запомнить, используйте известный дебют. Но что же можно сделать с кодом, чтобы его можно было легко читать? Некоторые исследователи рассматривали способы написания такого кода, который легко разбивается на чанки и соответственно легко обрабатывается.

Используйте паттерны проектирования

Если вы хотите написать код, который легко будет делить на чанки, то используйте паттерны (шаблоны) проектирования — к такому выводу пришел Вальтер Тихи (Walter Tichy), профессор информатики Технологического института Карлсруэ в Германии. Тихи исследовал чанки в программном коде, но вышло это случайно. В основном его интересовал вопрос, помогают ли паттерны проектирования программистам, когда они *сопровождают* код (добавляют новые функции или устраняют ошибки).

Тихи начал с малого: на группе студентов он проверил, помогает ли информация о паттернах проектирования понять код[5]. Он разделил всех студентов на две группы: обе группы получили одинаковый код, однако первой группе также дали дополнительную информацию о коде. Результаты исследования показали, что паттерн проектирования намного упрощал работу с кодом, особенно когда программисты знали, что этот паттерн есть в коде.

Тихи провел такое же исследование с опытными программистами[6]. В данном случае участники сначала выполняли упражнения по изменению кода, а затем проходили курс по паттернам проектирования. После прохождения курса они снова модифицировали код с паттерном проектирования или без него. Результаты данного исследования показаны на рис. 2.7. Следует отметить, что код, с которым участники работали после курса, отличался от кода из предыдущего упражнения. В исследовании использовалось два кода: участники, которые до курса работали с кодом A, после курса работали с кодом B, и наоборот.

На данном рисунке представлены результаты исследования Тихи с диаграммами типа «ящик с усами»[7]. На рисунке видно, что после прохождения курса (на рисунке это область справа, отмеченная как «заключительный тест») времени, требующегося для сопровождения кода с паттернами, нужно намного меньше, чем для сопровождения кода без паттернов. Согласно результатам исследования, наличие знаний о паттернах проектирования улучшает навык деления на чанки, а также помогает

---

[5] См. «Two Controlled Experiments Assessing the Usefulness of Design Pattern Information During Program Maintenance», Луц Прешель, Барбара Унгер и Вальтер Тихи (1998), **http://mng.bz/YA9K**.

[6] «A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns — A Replication in a Real Programming Environment», Марек Вокач, Вальтер Тихи, Даг Сьёберг, Эрик Arisholm, Магне Альдрин (2004), **http://mng.bz/G6oR**.

[7] Ящик представляет собой половину данных, где верхняя линия — это медиана третьего квартиля, нижняя линия — медиана первого квартиля. Линия в ящике — медиана, а «усы» показывают минимальное и максимальное значение.

быстрее обрабатывать код. Также обратите внимание на то, что разные паттерны проектирования имеют разный эффект: на код с паттерном проектирования «Наблюдатель» участникам понадобилось меньше времени, чем на код с паттерном проектирования «Декоратор».

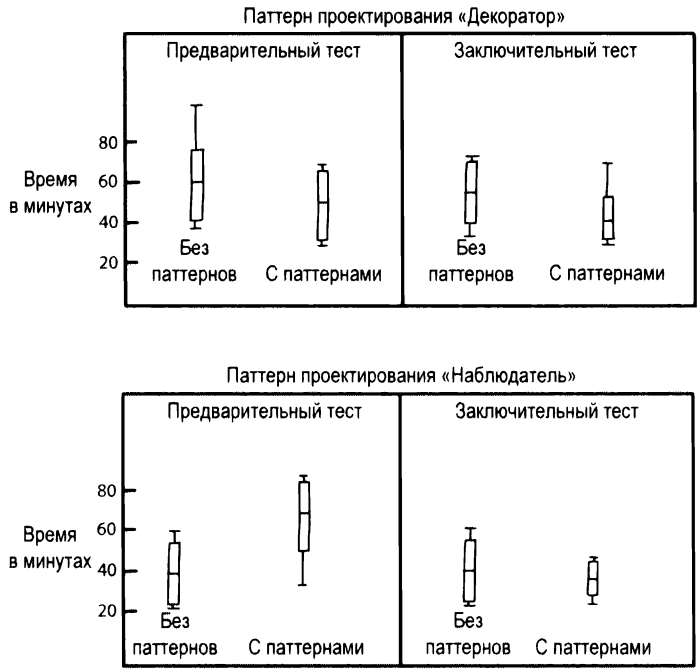


Рис. 2.7. На этих диаграммах показаны результаты исследования паттернов проектирования с участием опытных программистов, проведенного Вальтером Тихи Область «без паттернов» показывает время, которое участники затратили на изменение кода без паттернов проектирования, а область «с паттернами» — время, которое участники затратили на изменение кода с паттернами проектирования. В области «Предварительный тест» вы можете увидеть время, которое участники затратили перед прохождением курса по паттернам проектирования, а «заключительный тест» — время, которое участники затратили после прохождения курса по паттернам проектирования Результаты показывают, что после прохождения курса времени, затрачиваемого на код с паттерном проектирования, нужно меньше, чем на код без паттернов проектирования

Пишите комментарии

Нужно ли вам писать комментарии или код должен быть «самодокументируемым»? Этот вопрос на протяжении долгого времени остается предметом бурных дискуссий среди программистов.

Исследователи также изучили этот вопрос и пришли к некоторым интересным выводам. Исследования показали, что когда в коде есть комментарии, то программисту требуется больше времени, чтобы его прочитать. Вы можете подумать, что это плохо — комментарии замедляют работу и вы тратите на них дополнительное время, — но на самом деле это знак того, что программисты читают комментарии при чтении кода. Это говорит о том, что вы не зря добавляете комментарии в свой код. Марта Элизабет Кросби (Martha Elizabeth Crosby), исследователь из Гавайского

университета, изучала, как программисты читаю код и какую роль при этом играют комментарии[8]. Работа Кросби показывает, что начинающие программисты уделяют комментариям намного больше времени, чем опытные программисты. В *части IV* этой книги мы подробнее рассмотрим процесс включения новых сотрудников в команду, однако результаты Кросби показывают, что добавление в код комментариев помогает новым программистам понять вашу кодовую базу.

Комментарии не только помогают начинающим программистам, но также и играют большую роль в том, как именно программисты делят код на чанки. Написанная в 2010 году диссертация Чжуинь Фэнь (Quiyin Fan) из Университета Мэриленда «The Effects of Beacons, Comments, and Tasks on Program Comprehension Process in Software Maintenance» показала, что разработчики при чтении кода сильно зависят от комментариев. Например, комментарии вроде «Эта функция выводит заданное двоичное дерево по порядку» помогают программистам выделить крупные чанки кода. С другой стороны, комментарии вроде «Инкремент `i` (на единицу)» после строки `i++;` могут лишь запутать программиста.

Оставляйте «маячки»

Последнее, что еще можно сделать для того, чтобы упростить разбиение кода на чанки, — это добавить «маячки». «Маячки» — это части программы, которые помогают программисту понять, что делает код. Вы можете представить «маячок» как строку кода или даже как часть строки кода, на которую вы посмотрите, и у вас возникнет мысль «теперь я понял!».

Обычно «маячки» указывают на то, что фрагмент кода содержит определенные структуры данных, алгоритмы или концепции. В качестве примера давайте посмотрим на следующий код на языке Python, который обходит двоичное дерево (листинг 2.3).

Листинг 2.3. Последовательный обход вершин дерева на Python

```
# Класс, представляющий узел в дереве

class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

# Функция для выполнения последовательного обхода дерева
def print_in_order(root):
    if root:
```

[8] «How Do We Read Algorithms? A Case Study», Марта Э. Кросби и Ян Стеловский, **https://ieeexplore.ieee.org/document/48797**.

```
# Сначала возвращаемся на левом потомке
print_in_order(root.left)

# затем выводим данные по узлу
print(root.val)

# теперь возвращаемся на правом потомке
print_in_order(root.right)
print("Contents of the tree are")
print_in_order(tree)
```

Этот код на Python содержит несколько «маячков», по которым читатель может сделать вывод о том, что в данном коде в качестве структуры данных используется двоичное дерево:

- комментарии со словом дерево;
- переменные с именами `root` и `tree`;
- поля с именами `left` и `right`;
- строка текста с упоминанием деревьев (`Contents of the tree are`).

«Маячки» упрощают процесс понимания кода, т. к. чаще всего они выступают в качестве триггера и подтверждают или опровергают различные догадки программистов. Например, когда вы начинаете читать код на Python из предыдущего листинга, вы, возможно, понятия не имеете, для чего нужен код. Однако, когда вы прочитаете первый комментарий и увидите класс `Node`, вы догадаетесь, что код как-то связан с деревом. Поля `left` и `right` подтверждают вашу догадку и указывают на то, что код работает с двоичным деревом.

Различают два вида «маячков»: *простые маячки* и *составные маячки*.

Простые «маячки» — это понятные без объяснений элементы синтаксиса кода, как, например, значимые имена переменных. В коде на Python (листинг 2.3) `root` и `tree` являются простыми «маячками». В некоторых случаях такие операторы, как `+`, `>` и `&&`, и структурные операторы типа `if`, `else` и т. д. также могут считаться простыми «маячками», т. к. они просты для обработки и помогают программисту понять функциональность кода.

Составные «маячки» — это крупные элементы кода, состоящие из простых «маячков». Составные «маячки» обозначают семантическое значение функций посредством работы нескольких обычных «маячков». В коде на Python из листинга 2.3 «маячки» `self.left` и `self.right` вместе образуют составной «маячок». По отдельности они не дают глубокого понимания кода, а вот вместе дают. Составными «маячками» могут быть также элементы кода. Например, цикл `for` может быть составным «маячком», т. к. он содержит переменную, начальное значение, а также инкремент и граничные значения.

«Маячки» могут принимать разные формы: мы уже видели, что «маячками» могут служить имена переменных и классов; другие идентификаторы, например имена

методов, тоже могут быть «маячками». Кроме имен «маячками» могут быть и определенные конструкции программирования, например инициализация пустого списка или свопинг.

«Маячки» также имеют связь с чанками, однако многие исследователи рассматривают их как разные концепции. Обычно «маячки» представляют собой более мелкие части кода, чем чанки. Кросби, работу которой мы рассмотрели ранее, также изучала роль «маячков». Она обнаружила, что при чтении кода опытные программисты намного чаще начинающих используют «маячки»[9]. Следующее упражнение поможет вам узнавать полезные «маячки».

УПРАЖНЕНИЕ 2.5. Для того чтобы научиться выбирать правильные типы «маячков», может понадобиться практика. Используйте это упражнение для практики использования «маячков» в коде.

Шаг 1. Выберите код

Для данного упражнения выберите незнакомую кодовую базу, но на знакомом языке программирования. Если есть возможность, то было бы хорошо выполнить это упражнение на кодовой базе, которую знает ваш коллега, друг или знакомый, — в таком случае он сможет проверить ваше понимание кода. В кодовой базе выберите один метод или функцию.

Шаг 2. Изучите код

Изучите выбранный код и попробуйте понять общий смысл кода.

Шаг 3. Обращайте внимание на то, какие «маячки» вы используете

Каждый раз, когда наступает момент «ага, вот оно!» и вы понемногу начинаете понимать код, выпишите, что именно привело вас к такому выводу. Это может быть комментарий, имя переменной или метода, промежуточное значение — все это может быть «маячком».

Шаг 4. Рефлексируйте

Когда у вас сложится полное представление о коде, а также будет составлен список «маячков», задайте себе следующие вопросы:

- ❒ Какие «маячки» вы выписали?
- ❒ Это элементы кода или информация на естественном языке?
- ❒ Какую информацию они содержат?
- ❒ Представляют ли они знания о предметной области кода?
- ❒ Представляют ли они информацию о функциональности кода?

Шаг 5. Содействуйте развитию кода (необязательно)

Иногда выбранные вами «маячки» можно улучшить. Или, например, в коде могут понадобиться дополнительные «маячки», которых пока что нет. Это хоро-

[9] См. «The Roles Beacons Play in Comprehension for Novice and Expert Programmers», Марта Э. Кросби, Джин Шульц и Сьюзен Уиденбек.

шая возможность разнообразить код новыми «маячками». Так как до этого упражнения вы не были знакомы с выбранным кодом, то вы прекрасно понимаете чувства человека, который сталкивается с незнакомым кодом.

Шаг 6. Сравните свои результаты с кем-то другим (необязательно)

Если у вас есть друг, коллега или знакомый, который также хочет улучшить свой навык использования «маячков», то вы можете выполнить это упражнение вместе. Различия, которые появятся у вас обоих при воспроизведении кода, станут отличной темой для дискуссии. Так как мы знаем, что между начинающими и опытными программистами есть большие различия, то это упражнение научит вас оценивать ваш уровень знания языка программирования в сравнении с другими людьми.

2.3.3. Применяйте чанки

Описанные в этой главе эксперименты показали, что люди с большим опытом могут запомнить больше шахматных фигур, букв или строк. И хотя знание концепций программирования приходит с опытом, существует несколько вещей, которые помогут вам разбивать код на чанки уже сейчас.

В этой книге вы часто встретите выражение *осознанная практика*. Осознанная практика — это использование упражнения для улучшения определенного навыка. Например, отжимания — это осознанная тренировка мышц рук; тональная лестница — осознанная практика для музыкантов; произношение слов по буквам — осознанная практика для тех, кто только учится читать. В программировании осознанная практика обычно не особо популярна. Многие люди изучали программирование, много программируя, однако это не самый эффективный вариант обучения. А вот целенаправленное деление кода на чанки и активное запоминание кода — это отличные упражнения!

УПРАЖНЕНИЕ 2.6. Данное упражнение поможет вам понять, с какими концепциями вы уже знакомы, а какие концепции даются вам сложнее всего. Как показали исследования, знакомые концепции запоминать проще, чем незнакомые. Вы можете запомнить то, что уже знаете, поэтому данное упражнение можно использовать для (само)диагностики вашего знания кода.

Шаг 1. Выберите код

Выберите кодовую базу, с которой вы немного знакомы, но с которой редко работаете. Также вы можете взять код, который вы недавно написали. Убедитесь в том, что вы имеете базовые знания используемого языка программирования. Вы должны немного понимать, для чего нужен код, но не знать его от начала до конца. Вы должны оказаться в ситуации, похожей на ситуацию шахматистов: они знают доску и фигуры, но не знают расстановку.

В кодовой базе выберите метод или функцию размером примерно в половину страницы и не более 50 строк кода.

Шаг 2. Изучите код

Изучайте выбранный код максимум две минуты. Чтобы было удобнее, поставьте таймер. После того как время таймера закончится, закройте код.

Шаг 3. Воспроизведите код

Возьмите чистый лист бумаги или откройте новый файл и попытайтесь воспроизвести код как можно точнее.

Шаг 4. Рефлексируйте

Если вы уверены, что воспроизвели весь код, то откройте исходный код и сравните его с вашим результатом. Ответьте на следующие вопросы:

- ❒ Что запомнить было проще всего?
- ❒ Есть ли части кода, которые вы воспроизвели частично?
- ❒ Если ли части кода, которые вы не запомнили?
- ❒ Вы понимаете, почему у вас не получилось воспроизвести эти части?
- ❒ Есть ли в пропущенных частях незнакомые вам концепции программирования?
- ❒ Есть ли в пропущенных частях понятия предметной области, с которыми вы не знакомы?

Шаг 5. Сравните с кем-то другим (необязательно)

Если у вас есть друг, коллега или знакомый, который также хочет улучшить свой навык разбиения кода на чанки, то вы можете выполнить это упражнение вместе. Различия, которые появятся у вас обоих при воспроизведении кода, станут отличной темой для дискуссии. Так как мы знаем, что между начинающими и опытными программистами есть большие различия, то это упражнение научит вас оценивать ваш уровень знания языка программирования в сравнении с другими людьми.

Выводы

- ❒ Кратковременная память может вместить в себя от двух до шести элементов.
- ❒ Для того чтобы обойти ограничения кратковременной памяти, при запоминании информации кратковременная память взаимодействует с долговременной.
- ❒ Когда вы читаете новую информацию, мозг пытается разделить ее на знакомые части, называемые чанками.
- ❒ Когда хранящихся в долговременной памяти знаний не хватает, вам приходится полагаться на низкоуровневые элементы кода, такие как буквы и ключевые слова. В таком случае место в вашей кратковременной памяти быстро закончится.
- ❒ Когда в вашей долговременной памяти хранится достаточно релевантной информации, вы можете вспомнить такие концепции, как «цикл `for` в Java» или

«сортировка методом выбора в Python», которые занимают намного меньше места в кратковременной памяти.

- Когда вы читаете код, сначала он хранится в иконической памяти. До кратковременной памяти доходит лишь часть кода.
- Запоминание кода можно использовать как инструмент для (само)диагностики уровня кодирования. Так как всегда легче вспомнить то, что уже знакомо, части кода, которые вы вспомните, могут выявить паттерны проектирования, конструкции программирования или концепции предметной области, которые вы знаете лучше всего.
- Код может содержать в себе элементы, которые значительно упрощают его обработку: например, паттерны проектирования, комментарии и «маячки».

3 Как быстро выучить синтаксис

В этой главе:

- вы узнаете, почему важно хорошо разбираться в синтаксисе;
- рассмотрите методы запоминания синтаксиса;
- узнаете, что вы можете сделать, чтобы предотвратить забывание синтаксиса;
- определите, когда лучше изучать синтаксис и концепции программирования для достижения лучших результатов;
- узнаете, как синтаксис и концепции программирования сохраняются в долговременной памяти;
- выполните упражнения для укрепления памяти и лучшего запоминания концепций программирования.

Эта глава посвящена тому, как люди учатся запоминать. Из этой главы вы узнаете, почему одна информация врезается в память, а другая забывается. Например, в какой-то момент вы узнали, что `System.out.print()` — это метод языка Java, который выводит текст на экран. Однако вы не помните все методы Java. Я уверена, что хоть раз в жизни, но вам приходилось искать определенный синтаксис. Например, чтобы добавить день к модулю `DateTime`, нужно использовать метод `addDays()`, `addTimespan()` или `plusDays`?

Может быть, вы не хотите учить синтаксис наизусть, руководствуясь тем, что в любой момент вы можете посмотреть его в Интернете. Но, как было показано в предыдущей главе, ваши знания влияют на то, насколько эффективно вы читаете и понимаете код. Следовательно, запоминание синтаксиса, концепций программирования и структур данных поможет вам обрабатывать код быстрее.

В этой главе мы рассмотрим четыре метода, которые помогут вам лучше запоминать концепции программирования. Вы будете запоминать концепции программирования на долгое время, что позитивно скажется на вашей работе с кодом. Если вы

когда-нибудь пытались запомнить синтаксис Flexbox в CSS, порядок параметров в `boxplot()` библиотеки matplotlib или синтаксис анонимных функций в JavaScript, то эта глава для вас!

3.1. Советы по запоминанию синтаксиса

В предыдущих главах вы узнали, что запомнить код построчно не так просто. Да и вспомнить, какой синтаксис нужно использовать при написании кода, тоже может быть проблемой. Например, можете ли вы написать код для следующих ситуаций по памяти?

- Чтение файла hello.txt, а также запись всех строк в командную строку.
- Форматирование даты в формате день-месяц-год.
- Регулярное выражение для слов, начинающихся на *s* или на *season*.

Даже опытному программисту при выполнении этих задач могут понадобиться сторонние источники информации. В этой главе мы рассмотрим не только то, почему так сложно запомнить правильный синтаксис, но и узнаем, как можно улучшить запоминание. Однако сначала мы рассмотрим, почему так важно помнить код.

Многие программисты считают, что знать синтаксис необязательно — зачем тратить время на то, что можно просто найти в Интернете? Однако есть две причины, по которым «поиск в Интернете» может быть не самым лучшим решением. Первую причину мы рассмотрели в *главе 2*: то, что вы уже знаете, сильно влияет на вашу эффективность при чтении кода. Чем больше концепций и примеров синтаксиса вы знаете, тем проще вам будет делить код на чанки и, как следствие, проще код запоминать и обрабатывать.

3.1.1. Отвлечение снижает производительность

Вторая причина заключается в том, что отвлечение от работы может нанести намного больше вреда, чем вам кажется. Открыв браузер, вы захотите проверить электронную почту или почитать новости, которые не имеют никакого отношения к причине, по которой вы открыли браузер. Вы также можете отвлечься на другие темы, читая обсуждение за обсуждением на тематических сайтах по программированию.

Крис Парнин (Chris Parnin), профессор Университета штата Северная Каролина, изучил, что происходит, когда программисты отвлекаются от работы[1]. Парнин записал 10 тысяч сеансов программирования с 85 программистами. Он посмотрел, как часто программисты отвлекались на электронные письма или коллег (а отвлекались они часто!), но также и изучил то, что происходит после отвлечения. Парнин также установил, что перерывы негативно влияют на производительность про-

[1] См. «Resumption Strategies for Interrupted Programming Tasks», Крис Парнин и Спенсер Ругабер (2011), **http://mng.bz/0rpl**.

граммиста. Исследование показало, что после отвлечения человеку требуется примерно четверть часа для того, чтобы вернуться к работе с кодом. Только в 10% случаев после отвлечения программисты могли вернуться к работе над редактированием кода менее чем за минуту.

Результаты исследования показали, что после отвлечения программисты часто забывали важную информацию о коде, над которым они работали. Думаю, чувство вроде «Что же я делал?» хоть раз в жизни посещало и вас. Программистам, принявшим участие в исследовании Парнина, чаще всего приходилось прилагать усилия для того, чтобы вспомнить, на чем они остановились. Например, они должны были посмотреть несколько мест в кодовой базе, чтобы вспомнить детали, прежде чем вернуться к работе над кодом.

Теперь, когда вы знаете, почему так важно помнить синтаксис, мы рассмотрим способы, как можно быстро выучить синтаксис.

3.2. Как быстро выучить синтаксис с использованием карточек

Одним из наилучших способов выучить что-то, в том числе и синтаксис, является использование дидактических карточек. Дидактические карточки — это обычные карточки из бумаги или стикеры. На одной стороне вы пишете «подсказку» — например, определение того, что вы хотите выучить. На другой стороне вы пишете соответствующую подсказке информацию.

Если вы решите использовать карточки для программирования, то с одной стороны пишите концепцию программирования, а с другой стороны — соответствующий код. Карточки для понимания генератора выражений в Python могут выглядеть так:

1. Базовое выражение <-> `numbers = [x for x in numbers]`.
2. Выражение с выборкой <-> `odd_numbers = [x for x in numbers if x % 2 == 1]`.
3. Выражение с вычислением <-> `[x*x for x in numbers]`.
4. Выражение с выборкой и вычислением <-> `squares = [x*x for x in numbers if x > 25]`.

При работе с карточками прочитайте сторону, на которой написана подсказка, а затем попытайтесь вспомнить соответствующий синтаксис. Напишите синтаксис на отдельном листе бумаги или введите его в редакторе. Когда напишете, переверните карточку и сравните результат.

Обычно карточки с большой пользой применяются при изучении языков. Но любой язык, например французский, будет сложно выучить только с использованием карточек — во французском языке слишком много слов. В языках программирования, даже таких серьезных, как C++, намного меньше слов, чем в любом естественном языке. Следовательно, изучение основных элементов синтаксиса языка программирования с помощью карточек полностью оправдывает себя и не требует больших усилий.

УПРАЖНЕНИЕ 3.1. Подумайте о 10 концепциях программирования, которые вам трудно запомнить.

Сделайте для каждой концепции набор карточек и опробуйте их. Вы можете заниматься этим в одиночку, а также в группе или команде, где вы можете найти людей, которые также не могут запомнить некоторые концепции программирования.

3.2.1. Когда использовать карточки

Ключ к изучению синтаксиса заключается в частом использовании карточек на практике. Существует большое количество приложений, позволяющих создавать свои собственные цифровые карточки, например: Cerego, Anki или Quizlet. Преимущество таких приложений в том, что они напоминают вам о том, что пришло время снова потренироваться. Если вы будете регулярно просматривать свои бумажные карточки или заниматься в приложении, то в скором времени вы заметите, что ваш словарный запас увеличился. Это значительно сэкономит ваше время, которое вы могли потратить на поиск информации в Google, а также не позволит вам отвлекаться и улучшит ваш навык деления кода на чанки.

3.2.2. Расширяем набор карточек

Есть несколько моментов, когда можно добавить новые карточки в свой набор. Первый момент: когда вы изучаете новый язык программирования, фреймворк или библиотеку, вы можете создавать новую карточку каждый раз, когда вы встречаете новую концепцию. Например, как только вы начали изучать синтаксис составления списка, сразу же создайте соответствующий набор карточек.

Второй момент: вы собираетесь искать концепцию в Интернете. Это сигнал, означающий, что вы не знаете эту концепцию наизусть. Чем не повод создать для нее карточку? С одной стороны карточки напишите концепцию, а с другой стороны напишите код, который вы найдете.

Конечно, не нужно создавать карточку под каждую незнакомую концепцию. Современные библиотеки, концепции и базы кодов настолько огромны, что нет смысла запоминать их все. При встрече со второстепенной концепцией или элементом всегда можно прибегнуть к поиску в Интернете.

3.2.3. Убираем ненужные карточки

Если вы регулярно занимаетесь с карточками, то спустя какое-то время вы заметите, что вы выучили некоторые карточки. Когда это случится, вы можете убрать эти карточки. Чтобы отслеживать свой прогресс, на каждой карточке ведите статистику, сколько раз вы дали верный и неверный ответ, как это показано на рис. 3.1.

Если вы несколько раз подряд правильно вспоминаете карточку, то можете убрать ее. Конечно, если впоследствии вы снова забудете ее, то вы всегда можете вернуть ее обратно в набор. Если при работе с карточками вы используете приложение, то

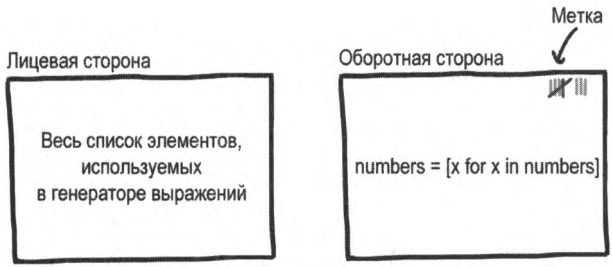


Рис. 3.1. Пример карточки с указанием верных и неверных ответов Так вы можете отслеживать, насколько надежно информация сохранена в вашей долговременной памяти

там есть функция, благодаря которой карточки, которые вы хорошо знаете, не появляются.

3.3. Как не забывать информацию

В предыдущем разделе мы узнали, как с помощью карточек можно быстро и легко выучить синтаксис. Но как долго нужно тренироваться? Когда вы постигнете Java 8 полностью? В этом разделе вы узнаете, как часто нужно «освежать» знания, прежде чем вы запомните информацию на долгое время.

Прежде чем мы перейдем к тому, *как* не забывать, нам нужно понять, *почему* люди вообще забывают. Как вы уже знаете, у кратковременной памяти есть свои недостатки — она не сохраняет много информации, а информация, которая уже хранится там, сохраняется на короткий период времени. У долговременной памяти тоже есть ограничения, но другие.

Одной из главных проблем долговременной памяти является невозможность запоминать информацию на долгое время без постоянного повторения. После того как вы что-то прочитали, услышали или увидели, информация переходит из кратковременной памяти в долговременную. Однако это не значит, что эта информация останется в долговременной памяти навсегда. В этом смысле долговременную память человека нельзя сравнивать с жестким диском компьютера, на котором информация вряд ли сотрется или потеряется.

Конечно, когда мы говорим о долговременной памяти, то она сохраняет информацию не на несколько секунд, как это делает кратковременная память. И все же этого очень мало. Посмотрите на кривую забывания на рис. 3.2. Примерно через час вы забываете половину того, что прочитали. Через два дня остается только 25% от полученной информации. Но это не все: график на рис. 3.2 показывает, сколько в конце концов вы запомните, если не будете «освежать» полученную информацию.

3.3.1. Почему мы забываем

Чтобы понять, почему некоторая информация так быстро забывается, нам нужно подробно рассмотреть работу долговременной памяти. Начнем с того, как мы запоминаем.

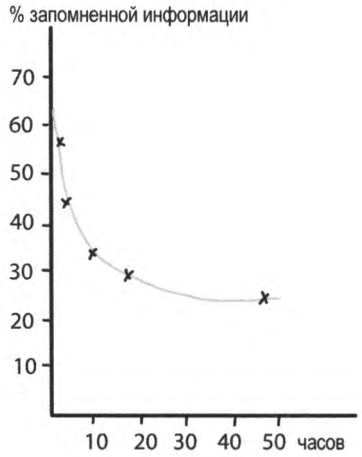


Рис. 3.2. График, показывающий, сколько информации вы запомнили после того, как вы ее узнали Через два дня в вашей долговременной памяти остается только 25% знаний

Мозг не хранит информацию в виде единиц и нулей, однако кое-что общее с хранением информации на диске все же есть — в обоих случаях это называется *кодированием*. Но когда ученые говорят о кодировании, они имеют в виду не передачу мыслей в память, а происходящие в мозге изменения, когда нейроны формируют воспоминания. По правде говоря, ученые до сих пор точно не знают, как происходит кодирование в мозге.

Иерархия и сеть

Мы сравнили долговременную память с жестким диском, однако на самом деле человеческий мозг не работает по принципу иерархии, где все файлы лежат в папках и подпапках. Как показано на рис. 3.3, воспоминания представляют собой сеть. Дело в том, что каждое событие, явление и воспоминание связано со множеством других. Осознание связи между различными явлениями и воспоминаниями важно для понимания того, почему люди забывают.

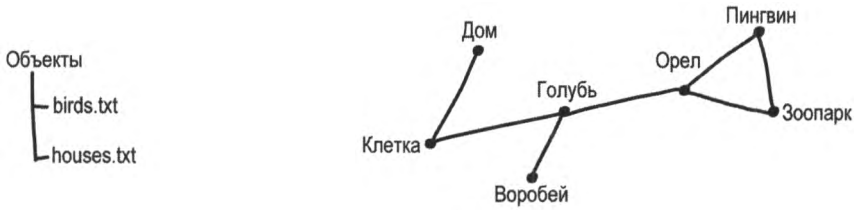


Рис. 3.3. Два способа организации данных слева представлена иерархическая файловая система, а справа — сетевая система

Кривая забывания

Герман Эббингауз (Hermann Ebbinghaus), немецкий философ и психолог, интересовался возможностями человеческого мозга в 1870-х годах. Тогда идея измерения умственных способностей человека была не особо популярна.

Эббингауз хотел понять пределы человеческое памяти, для чего он использовал собственную память. Он быстро осознал, что заучивание известных слов и концепций было бесполезным для экспериментов, потому что все воспоминания в той или иной мере связаны друг с другом. Например, если вы захотите запомнить синтаксис генератора выражений, то знание синтаксиса цикла `for` значительно облегчит вашу задачу.

Для получения более правдивых результатов Эббингауз создал множество коротких бессмысленных слов, например: *wix*, *maf*, *kel* и *jos*. Затем он провел серию экспериментов с самим собой в качестве испытуемого. В течение нескольких лет он учил списки бессмысленных слов, читая все слова вслух под метроном, чтобы не сбиться с темпа, и отслеживал, сколько потребовалось времени для того, чтобы точно запомнить каждый список слов.

Через десять лет, в 1880 году, он произвел подсчеты и получил такой результат: ему потребовалось почти тысяча часов на практику, а в минуту он цитировал 150 слов. Проверяя себя спустя разные периоды времени, Эббингауз смог определить временной интервал для своей памяти. Он описал все свои результаты и выводы в своей публикации 1885 года под названием *Über das Gedächtnis* (*О памяти*). В книге дана формула забывания, показанная на рис. 3.4, которая и лежит в основе концепции кривой забывания.

$$b = 100x\frac{1,84}{(\log_{10} t)^{1.25} + 1,84}$$

Рис. 3.4. Формула Эббингауза для расчета, как долго информация будет храниться в памяти

Недавнее исследование нидерландского профессора Япа Мюррэ (Jaap Murre) из Амстердамского университета показало, что в основном формула Эббингауза верна[2].

3.3.2. Интервальное повторение

Теперь мы знаем, как люди забывают информацию. Однако как это поможет нам не забыть синтаксис `boxplot()` библиотеки matplotlib или генератора выражений в Python? Оказалось, что эксперименты Эббингауза с заучиванием бессмысленных слов не только позволили определить, сколько нужно времени для того, чтобы забыть полученную информацию, но также и помогли понять, как можно предотвратить забывание. Эббингауз заметил, что он смог выучить ряд из 12 бессмысленных слов с 68 повторениями в первый день и с 7 повторениями на следующий день (всего 75 повторений); также он пробовал учить такой же ряд из бессмысленных слов, но с 38 повторениями в течение трех дней, что составляло половину от всего времени учебы.

---

[2] См. «Replication and Analysis of Ebbinghaus' Forgetting Curve», Яп Мюррэ (2015), **https:/journals.plos.org/plosone/article?id=10.1371/journal.pone.0120644**.

Кривую забывания тщательно исследовали: проводились как эксперименты с маленькими детьми, которых обучали базовым математическим операциям, так и со старшеклассниками. Исследование, пролившее свет на оптимальной период времени между повторениями, провел Гарри Бахрик (Harry Bahrick) из Уэслианского университета Огайо. Однако он проводил исследование не только над собой, но и над своей женой и двумя взрослыми детьми, которые тоже были учеными и интересовались этой темой[3].

Все они поставили перед собой цель выучить 300 слов на иностранном языке. Его жена и дочь изучали слова на французском, а сам Бахрик учил слова на немецком. Они разделили все слова на 6 групп по 50 слов и использовали разные интервалы между повторениями для каждой из групп слов. Каждую группу слов учили 13 или 26 раз с интервалом в 2, 4 или 8 недель. Затем они проверяли, сколько они помнят через один, два, три года или пять лет.

Через год после окончания эксперимента Бахрик и его семья обнаружили, что больше всего слов они запомнили из группы, которая была с самыми длинными интервалами между повторениями и с самым большим количеством повторений — 26 раз каждые 8 недель. Через год из этой группы они смогли вспомнить 76% слов, в то время как из группы, где повторения были через каждые две недели, они смогли вспомнить только 56% слов. В последующие годы количество выученных слов уменьшилось, однако все равно количество запомненных слов из группы с самым большим интервалом повторений было больше.

Из этого можно сделать вывод, что чем дольше вы что-то учите, тем больше вы запомните. Однако это не значит, что все время вы должны уделять заучиванию — это означает только то, что вам нужно учить с увеличенными интервалами. Для улучшения вашей памяти и лучшего запоминания информации будет достаточно просмотреть ваши дидактические карточки раз в месяц. Конечно, этот подход не сравнить с «обязательным» образованием, где мы стараемся изучить все-все за один семестр, или учебными лагерями (буткемпами), где людей стараются обучить всему за три месяца. Знания, полученные таким образом, останутся в вашей памяти надолго лишь в том случае, если вы будете часто повторять то, что выучили.

Совет

Главный вывод, который можно сделать из этого раздела, заключается в том, что лучший способ не забыть выученное — это регулярное повторение и практика Каждый раз, когда вы повторяете уже знакомую вам информацию, вы укрепляете свою память. После нескольких повторений в течение определенного периода времени информация должна остаться в вашей долговременной памяти навсегда И если вы когда-нибудь задумывались о том, почему большая часть информации, что вы учили в колледже или университете, забыта, то теперь вы знаете ответ! Если вы не будете возвращаться к полученным знаниям, то вы их быстро забудете

[3] См «Maintenance of Foreign Language Vocabulary and the Spacing Effect», Гарри Бархик и др. (1993), **www.gwern.net/docs/spaced-repetition/1993-bahrick.pdf**.

3.4. Как запомнить синтаксис надолго

Теперь вы понимаете, что учить синтаксис очень важно: это помогает делить код на чанки и экономит время, которое вы могли потратить на поиск информации. Мы также узнали, как часто следует практиковаться: не нужно пытаться запомнить все карточки за один день, занимайтесь в течение длительного периода времени. В оставшейся части главы мы поговорим, как нужно практиковаться. Мы рассмотрим два способа улучшения памяти: повторение (активные попытки вспомнить информацию) и проработку (активное увязывание новой информации с той, что уже хранится в памяти).

Как вы могли заметить, я не предлагала вам просто читать обе стороны карточек. Вместо этого я просила вас прочитать только сторону с подсказкой, что должно было помочь вспомнить соотвествующий синтаксис.

Исследования показали, что активные попытки вспомнить что-то способствуют укреплению памяти. Даже если вы не помните что-то полностью, легче вспомнить то, что вы часто пытались вспомнить раньше. Мы рассмотрим этот процесс подробнее и узнаем, как это можно применить к обучению программированию.

3.4.1. Два способа запоминания информации

Прежде чем мы начнем разбираться с тем, как запоминание укрепляет нашу память, давайте сначала посмотрим на саму проблему. Вы можете думать, что память либо находится в мозге, либо нет, но на самом деле все сложнее. Роберт и Элизабет Бьорк (Robert и Elizabeth Bjork), профессора психологии Калифорнийского университета, выделили два механизма извлечения информации из долговременной памяти: уровень хранения и уровень воспроизведения.

Уровень хранения

Уровень хранения показывает, насколько хорошо сохранилась информация в долговременной памяти. Дело в том, что чем больше вы что-то учите, тем лучше вы это запоминаете. В конце концов вы так запомните материал, что уже никогда его не забудете! Можете ли вы представить себе, что вы забыли, что 3 умножить на 4 это 12? Однако не всю информацию, хранящуюся в мозге, можно вспомнить так же легко, как таблицу умножения.

Уровень воспроизведения

Уровень воспроизведения показывает, насколько легко можно что-то вспомнить. Уверена, у вас было чувство, что вы точно знаете что-то (имя, название песни, номер телефона, синтаксис метода `filter()` в JavaScript), но не можете вспомнить. Если ваш уровень воспроизведения высокий, то в конце концов вы вспомните эту информацию.

Принято считать, что уровень хранения может только увеличиваться. Согласно недавним исследованиям, на самом деле люди никогда не забывают воспомина-

ния[4], — с годами ослабевает уровень воспроизведения. Когда вы постоянно изучаете какую-то новую информацию, вы укрепляете уровень хранения этой информации. Когда вы пытаетесь вспомнить факт, который, как вам кажется, вы знаете, то вы улучшаете уровень воспроизведения.

3.4.2. Просто увидеть недостаточно

При поиске определенного синтаксиса чаще всего проблема заключается в уровне не хранения, а воспроизведения. Например, можете ли вы найти правильный код для обхода списка в обратном порядке на C++ из нескольких вариантов (листинг 3.1)?

Листинг 3.1. Шесть вариантов обхода списка на C++

```
1. rit = s.rbegin(); rit != s.rend(); rit++
2. rit = s.revbegin(); rit != s.end(); rit++
3. rit = s.beginr(); rit != s.endr(); rit++
4. rit = s.beginr(); rit != s.end(); rit++
5. rit = s.rbegin(); rit != s.end(); rit++
6. rit = s.revbeginr(); rit != s.revend(); rit++
```

Все приведенные варианты очень похожи, и даже опытному программисту на C++ может быть сложно вспомнить правильный, хотя он и видел его множество раз. Когда вам говорят правильный ответ, то у вас создается впечатление, что вы всегда его знали: «Конечно же, это ведь `rit = s.rbegin(); rit != s.rend(); rit++`!».

Следовательно, проблема заключается не в прочности, с которой знания хранятся в долговременной памяти (уровень хранения), а в легкости, с которой вы можете найти информацию (уровень воспроизведения). Этот пример показывает, что даже если вы множество раз видели код, это не значит, что вы его запомнили. Нужная информация хранится где-то в вашей долговременной памяти, однако это не значит, что вы всегда сможете ее вспомнить.

3.4.3. Воспоминания укрепляют память

Упражнение из предыдущего раздела показало, что просто хранить информацию в долговременной памяти недостаточно. Вы также должны с легкостью ее вспомнить. Как и многое другое в нашей жизни, вспоминать информацию намного легче, если вы постоянно это практикуете. Так как вы никогда не пытались целенаправленно вспоминать синтаксис, при необходимости вам будет трудно его вспомнить. Мы знаем, что активные попытки вспомнить что-то укрепляют память — эта методика существовала еще во времена Аристотеля.

---

[4] «Recollection, Familiarity, and Cortical Reinstatement: A Multivoxel Pattern Analysis», Джеффри Дж. Джонсон, Сьюзен МакДафф, Майкл Д. Рагг и Кеннет Норман, Neuron, vol. 63, no. 5, September 8, 2009.

Одно из первых исследований данной методики было проведено школьным учителем Филипом Босвудом Баллардом (Philip Boswood Ballard), который в 1913 году опубликовал статью «Obliviscence and Reminiscence» («Забывчивость и припоминание»). В ходе эксперимента он попросил учеников выучить 16 строк из стихотворения «Крушение "Гесперус"», в котором рассказывается история тщеславного шкипера, который стал причиной смерти его дочери. Когда Баллард исследовал способность учеников вспоминать выученное стихотворение, он заметил кое-что интересное. Баллард не сказал ученикам, когда он спросит их в следующий раз, и попросил их рассказать стихотворение через два дня. Так как ученики не знали, что их будут спрашивать еще раз, они больше не учили стихотворение. Когда Баллард просил учеников прочитать поэму во второй раз, то он заметил, что в среднем все ученики запомнили на 10% больше, чем в прошлый раз. Еще больше материала они помнили еще через два дня. Отнесшись к результатам с недоверием, Баллард повторил это исследование несколько раз, однако результаты каждый раз были похожими: когда вы активно пытаетесь вспомнить информацию, при этом не прибегая к дополнительному заучиванию, вы вспоминаете больше материала.

Теперь, когда вы познакомились с кривой забывания и эффектами практики активного воспроизведения информации, вы понимаете яснее, что искать забытый синтаксис каждый раз, когда он вам нужен, не самая лучшая идея. Так как найти синтаксис на сторонних ресурсах не составляет никакого труда, наш мозг начинает считать, что запоминать синтаксис не нужно. Следовательно, уровень воспроизведения синтаксиса языков программирования остается слабым.

Забывание синтаксиса приводит к замкнутому кругу. Мы не помним синтаксис, следовательно, нам нужно его где-то найти. Но если мы будем продолжать постоянно искать его вместо того, чтобы пытаться вспомнить, то наш уровень воспроизведения этих концепций программирования не повысится и мы снова должны будем искать нужную нам информацию, и так до бесконечности.

В следующий раз, когда вы соберетесь что-нибудь загуглить, возможно, стоит сначала вспомнить синтаксис самостоятельно. Даже если у вас не получится вспомнить синтаксис, сама попытка укрепит вашу память, и, может быть, в следующий раз вы сможете вспомнить нужную вам информацию. Однако если это не помогает, то сделайте дидактическую карточку и активно ее учите.

3.4.4. Укрепление памяти путем активного мышления

В предыдущем разделе вы узнали, что активные попытки вспомнить информацию и практика помогают запомнить информацию. Вы также узнали, что лучше всего запоминать информацию в течение длительного периода времени. Однако существует еще один способ укрепления памяти — это активное мышление. Активное обдумывание информации, которую вы только что выучили, называется *проработкой*. Данный способ хорошо подходит для изучения сложных концепций программирования.

Прежде чем мы углубимся в процесс проработки и то, как это можно использовать для запоминания новых концепций программирования, давайте подробнее рассмотрим, как работает память.

Схемы

Как вы уже знаете, все воспоминания представляют собой сеть, в которой они связаны с другими воспоминаниями и фактами. Воспоминания и связи между ними формируют *схему* или *схемы*.

Когда вы изучаете новую информацию, перед запоминанием информации в долговременной памяти в вашем мозге создается схема. Информация, которая больше всего соответствует уже существующим схемам, запомнится лучше. Например, если я попрошу вас запомнить числа 5, 12, 91, 54, 102 и 87, а затем воспроизвести три из них, чтобы получить приз на выбор, то вы столкнетесь с трудностями — числа сложно связать друг с другом. Поэтому числа сохранятся в новой схеме под названием «Числа, которые я запомнила для того, чтобы выиграть классный приз».

Однако, если я попрошу вас запомнить числа 1, 3, 15, 127, 63 и 31, то вы их запомните с большей вероятностью. Если вы посмотрите повнимательнее, то заметите, что все числа, если их преобразовать в двоичную систему, будут состоять только из единиц. Так вы не только запомните числа быстро и легко, но и получите мотивацию запоминать числа — вы понимаете, что это имеет смысл и вы не тратите время зря. Вы понимаете, что если вы знаете верхнюю границу чисел в битах, то это может помочь решить вам некоторые проблемы.

Когда рабочая память обрабатывает информацию, она ищет в долговременной памяти связанные с информацией воспоминания и факты. Если ваши воспоминания связаны друг с другом, то найти их будет намного проще. Другими словами, уровень воспроизведения выше для тех воспоминаний, которые связаны с другими воспоминаниями.

Когда мы запоминаем информацию, она может быть немного изменена для того, чтобы ее можно было связать с существующими схемами. В 1930-х годах британский психолог Фредерик Бартлетт (Frederic Bartlett) провел эксперимент, в котором он попросил группу людей прочитать короткую сказку коренных американцев под названием «The War of the Ghosts» («Война призраков») и вспомнить ее через несколько недель или месяцев[5]. Слушая пересказ сказки спустя определенное время, Бартлетт заметил, что участники частично изменили сказку, чтобы она соответствовала их убеждениям или знаниям. Например, некоторые из участников опустили детали, которые они считали неважными. Другие участники адаптировали сказку и сделали ее более «западной», например заменили лук на ружье. Данный эксперимент показал, что люди не запоминают просто слова или факты, а подстраивают их под уже имеющийся опыт, знания и взгляды.

Тот момент, что при запоминании информации мы можем ее изменить, имеет свои недостатки. Два человека, попавшие в одну и ту же ситуацию, могут вспомнить ее совершенно по-разному: сами мысли людей влияют на то, как они запоминают и хранят информацию. Однако мы также можем опираться на тот факт, что воспоми-

---

[5] См. его книгу «Remembering: A Study in Experimental and Social Psychology» (Cambridge University Press, 1932).

нания могут быть изменены в наших интересах: мы сохраняем основную информацию, при этом добавляя часть от себя.

Проработка для запоминания концепций программирования

Как вы уже заметили ранее в этой главе, воспоминания можно забыть, когда уровень воспроизведения недостаточно высок. Эксперимент Бартлетта показал, что при запоминании информации, когда она сохраняется в долговременной памяти даже впервые, некоторые детали могут быть изменены или забыты.

Например, если я вам скажу, что Джеймс Монро (James Monroe) был пятым президентом США, то вы запомните, что Монро — бывший президент, но забудете тот факт, что он был пятым. Скорее всего вы не запомните номер по нескольким причинам: вам могло показаться это сложным, вы могли отвлечься или посчитать эту информацию неважной. На количество запоминаемых данных влияет много факторов, в том числе и ваше эмоциональное состояние. Например, вы быстрее запомните ошибку, над исправлением которой сидели целую ночь год назад, чем ошибку, которую вы совершили сегодня и исправили за несколько минут.

И хотя вы никак не можете изменить свое эмоциональное состояние, есть много вещей, которые можно сделать для того, чтобы сохранить как можно больше новых воспоминаний. И проработка — это одна из таких вещей.

Проработка означает размышление над той информацией, которую вы хотите запомнить, а также ее связь с существующими воспоминаниями. Проработка могла стать одной из причин, по которой ученики в эксперименте Балларда с каждым разом все лучше и лучше запоминали стихотворение. Неоднократное повторение стихотворения заставляло их вспоминать забытые слова. Возможно, ученики также связали части стихотворения с другими вещами, которые они запомнили раньше.

Если вы хотите лучше запомнить новую информацию, то вам нужно подробно ее проработать. Процесс проработки укрепляет сеть связанных воспоминаний, а когда воспоминание связано со множеством других, его проще вспомнить. Представьте, что вы учите новую концепцию программирования, например генератор списков на Python. Генератор списков — это инструмент для создания списка на основе уже существующего списка. Например, вы хотите создать список квадратов чисел, которые вы уже сохранили в списке `numbers`. Вы можете создать такой список с помощью генератора списка:

```
squares = [x*x for x in numbers]
```

Представьте, что эту концепцию вы изучаете впервые. Если вы хотите помочь себе в запоминании, то воспользуйтесь осознанной проработкой, где вы будете рассматривать связанные концепции. Например, вы можете размышлять о связанных концепциях других языков программирования, об альтернативных концепциях языка Python или других языков или о том, как эта концепция связана с другими подходами.

УПРАЖНЕНИЕ 3.2. Используйте данное упражнение для изучения новой концепции программирования. Ответы на следующие вопросы помогут вам укрепить новые знания:

- ❒ Думая о новой концепции программирования, вы вспоминаете другие концепции. Что это за концепции? Запишите их.
- ❒ Для каждой из записанных концепций ответьте на следующие вопросы:
  - Почему новая концепция заставляет меня думать об этой уже известной мне концепции?
  - У обеих концепций одинаковый синтаксис?
  - Эта концепция может использоваться в данном случае?
  - Является ли новая концепция альтернативой той концепции, которую я уже знаю?
- ❒ Знаете ли вы другие способы написать код, чтобы в итоге он выполнял нужную задачу? Постарайтесь написать как можно больше вариантов такого кода.
- ❒ Есть ли эта концепция в других языках программирования? Можете ли вы записать примеры на других языках? Чем они отличаются от выбранной вами концепции?
- ❒ Соответствует ли новая концепция определенной предметной области, библиотеке или фреймворку?

Выводы

- ❒ Важно знать синтаксис наизусть, т. к. знание синтаксиса облегчит деление кода на чанки. К тому же, если вы решите искать синтаксис в Интернете, вы можете отвлечься от работы.
- ❒ Для практики и запоминания нового синтаксиса используйте дидактические карточки: на одной стороне укажите подсказку, а на другой — сам код.
- ❒ Важно регулярно повторять новую информацию, чтобы не забыть ее.
- ❒ Лучшей практикой будет не поиск информации в Интернете или в документации, а попытка самостоятельно вспомнить нужную вам информацию.
- ❒ Чтобы увеличить объем знаний, которые вы можете вспомнить, практикуйтесь в течение продолжительного времени.
- ❒ Информация в вашей долговременной памяти хранится в виде сети, в которой все явления и факты связаны между собой.
- ❒ Активная проработка новой информации позволяет укрепить сеть воспоминаний и связать с ней новую информацию.

4 Как читать сложный код

В этой главе:

- проанализируем, что происходит, когда рабочая память становится перегруженной из-за сложного кода;
- сравним два разных вида перегрузки рабочей памяти при программировании;
- перепроектируем код для компенсации перегруженной рабочей памяти;
- научимся создавать таблицу состояния и граф зависимостей для поддержки рабочей памяти при чтении сложного кода.

В *главе 1* мы рассмотрели различные способы замешательства при чтении кода. Мы узнали, что замешательство может быть вызвано недостатком информации, которую необходимо хранить в кратковременной памяти, или отсутствием необходимых знаний в долговременной памяти. В этой главе мы рассмотрим третий источник замешательства — недостаток вычислительной мощности мозга.

Иногда код бывает слишком трудным для понимания. Из-за того, что программисты редко практикуют чтение кода, вы могли заметить у себя недостаток стратегий, применимых к чтению кода. Стратегии вроде «прочти это еще раз» и «сдавайся» чаще всего оказываются бесполезными.

В предыдущих главах мы рассмотрели способы, которые можно использовать для улучшения навыка чтения кода. В *главе 2* мы узнали о способах эффективного разбиения кода на чанки, а в *главе 3* получили советы, как можно запомнить большой объем синтаксиса в долговременной памяти и не забыть его. Однако иногда код бывает настолько сложным, что даже умение делить код на чанки и отличное знание синтаксиса ничем не помогут.

В данной главе мы рассмотрим когнитивные процессы, лежащие в основе вычислительной способности мозга, которые мы обычно называем *рабочей памятью*. Мы

узнаем о рабочей памяти чуть больше, а также узнаем, как код может быть настолько запутанным, что он даже может перегрузить рабочую память. После того как мы рассмотрим основы, я покажу вам три основных способа поддержки рабочей памяти, которые помогут вам с легкостью обрабатывать даже самый сложный код.

4.1. Почему так тяжело понимать сложный код

В *главе 1* я показала вам пример программы на BASIC, достаточно сложной, чтобы полностью понять программу, только прочитав код. В подобных случаях, возможно, имеет смысл записать рядом с кодом промежуточные значения, как это показано на рис. 4.1.

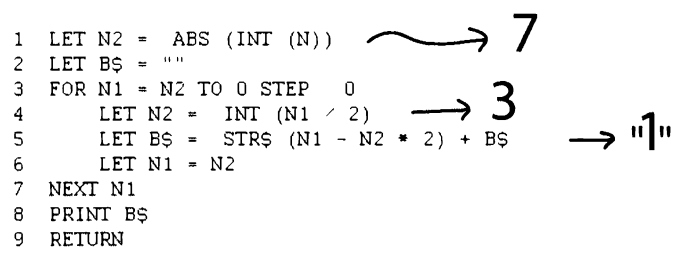


Рис. 4.1. Программа на BASIC, преобразующая число N в двоичное представление. Программа приводит в замешательство из-за того, что вы не можете отследить все совершаемые шаги, поэтому при работе с кодом вы можете обратиться к вспомогательным средствам памяти, например записыванию промежуточных значений переменных

Сам факт того, что вам нужно что-то дополнительно записать, означает то, что у вашего мозга недостаточно ресурсов для обрабатывания кода. Давайте сравним код на языке BASIC со вторым примером из *главы 1*, программой на языке Java, которая вычисляет значение числа n в двоичном представлении. Понимание этого кода может потребовать некоторых умственных усилий, к тому же незнание принципа работы метода `toBinaryString()` может вызвать замешательство. Но, несмотря на все это, вряд ли вам потребуется делать заметки во время чтения этого кода.

Листинг 4.1. Программа на Java, преобразующая n в двоичное представление

```
public class BinaryCalculator {
    public static void main(Integer n) {
       System.out.println(Integer.toBinaryString(n));
    }
}
```

В предыдущих главах мы подробно рассмотрели два когнитивных процесса, которые возникают при чтении сложного кода, — кратковременную и долговременную память. Однако для понимания того, почему иногда требуется «выгружать» ин-

формацию, нам нужно рассмотреть третий когнитивный процесс из *главы 1*. Рабочая память представляет собой способность человеческого мозга думать, формировать мысли, идеи, находить решение проблем. Ранее мы сравнивали кратковременную память с ОЗУ компьютера, долговременную память — с жестким диском. Если следовать этой аналогии, то рабочую память можно сравнить с процессором мозга.

4.1.1. Чем друг от друга отличаются рабочая память и кратковременная память

Некоторые используют термин «рабочая память» в качестве синонима кратковременной памяти, и, быть может, вы видели, что иногда эти два термина взаимно заменяют друг друга. Другие, наоборот, проводят между двумя терминами четкую границу — собственно, в этой книге мы придерживаемся такой же точки зрения. Роль кратковременной памяти — это *запоминание* информации. С другой стороны, роль рабочей памяти — это *обработка* информации. Поэтому данные процессы мы будем рассматривать как отдельные, непохожие друг на друга.

> **Определение**
>
> Определение *рабочей памяти*, которое будет использоваться в оставшейся части книги, — это «кратковременная память, использующаяся для решения проблем».

На рис. 4.2 показана разница между двумя процессами: если вы запоминаете номер телефона, то вы используете кратковременную память; если вы складываете числа, то вы используете рабочую память.

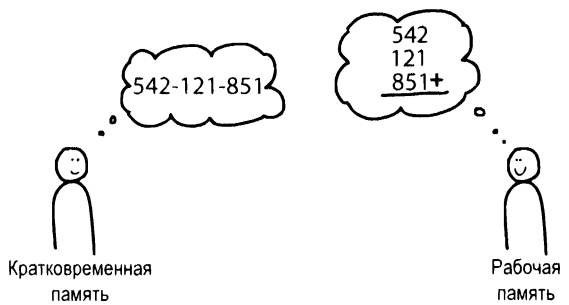


Рис. 4.2. Кратковременная память на короткий период времени сохраняет информацию (например, номер телефона, на рисунке — *слева*), тогда как рабочая память обрабатывает информацию (например, выполняет вычисления, на рисунке — *справа*)

Как вы знаете из *главы 2*, кратковременная память может хранить одновременно от двух до шести элементов. Чтобы обработать больше информации, ее можно разделить на узнаваемые чанки, как это делают со словами, шахматными дебютами и паттернами проектирования. Так как рабочая память — это кратковременная память, использующаяся для решения конкретной задачи, ограничения у нее те же.

Как и кратковременная память, рабочая память способна обрабатывать от двух до шести элементов одновременно. В контексте рабочей памяти эта способность больше известна как *когнитивная нагрузка*. Попытка решить задачу, содержащую

множество отдельных элементов, которые нельзя эффективно разделить на чанки, приведет к «перегрузке» рабочей памяти.

4.1.2. Типы когнитивной нагрузки и как они связаны с программированием

В данной главе мы рассмотрим способы устранения когнитивной нагрузки, однако, прежде чем приступить к этой теме, мы должны рассмотреть существующие виды когнитивной нагрузки. Первым исследователем, предложившим термин «когнитивная нагрузка», был австралийский профессор Джон Свеллер (John Sweller). Он выделил три формы когнитивной нагрузки: внутренняя, внешняя и соответствующая нагрузка. В табл. 4.1 приводится краткое описание каждого типа когнитивной нагрузки.

Таблица 4.1. *Типы когнитивной нагрузки*

Тип нагрузки	Краткое пояснение
Внутренняя нагрузка	Насколько задача сложна сама по себе
Внешняя нагрузка	Какие отвлекающие факторы усугубляют задачу
Соответствующая нагрузка	Когнитивная нагрузка, созданная необходимостью хранить все мысли в долговременной памяти

Мы сфокусируемся на первых двух типах когнитивной нагрузки. Соответствующую нагрузку мы подробно рассмотрим в следующей главе.

Внутренняя когнитивная нагрузка при чтении кода

Внутренняя нагрузка — это когнитивная нагрузка, вызванная некоторыми особенностями задачи, которые заложены в ней изначально. Например, представьте, что вам нужно узнать значение гипотенузы треугольника (рис. 4.3).

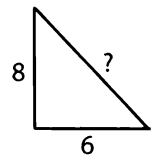


Рис. 4.3. Геометрическая задача, в которой даны значения двух сторон треугольника и нужно вычислить значение третьей стороны. Сложность этой задачи зависит от ваших знаний. В любом случае нельзя эту задачу как-то изменить или пропустить в ней какие-то шаги

Для решения данной задачи вам нужно иметь определенные знания. Например, вам нужно знать теорему Пифагора ($a^2+b^2=c^2$), также вы должны уметь возводить числа в квадрат, а потом извлекать квадратный корень из полученного числа. Так как не существует других способов решить или упростить эту задачу, когнитивная нагрузка является *внутренней*. В программировании мы часто используем термин *внутренняя сложность* для описания внутренних аспектов какой-либо задачи.

А в когнитивистике мы говорим, что эти аспекты вызывают когнитивную нагрузку внутреннего типа.

Внешняя когнитивная нагрузка при чтении кода

Кроме естественной внутренней нагрузки, причиной которой является какая-нибудь задача, существует и другой тип когнитивной нагрузки, которая может *добавиться* к задаче, часто случайно. Снова обратимся к задаче на вычисление длины гипотенузы, но сформулируем ее так, что при решении нам придется мысленно связать две стороны треугольника, длина которых уже известна, с их обозначениями (рис. 4.4). Результатом такой дополнительной работы будет *внешняя когнитивная нагрузка* мозга.

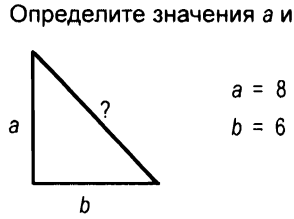


Рис. 4.4. Задача с такой постановкой условий вызывает внешнюю когнитивную нагрузку

На самом деле решение задачи от этого условия никак не изменилось — нам все еще нужно знать теорему Пифагора. Однако это условие сильно отражается на производительности мозга: теперь мозгу нужно соотнести *a* со значением 8 и *b* со значением 6. В программировании мы относимся к такой внешней нагрузке как к ненужной сложности, т. е. к аспектам программы, которые делают поставленную задачу сложнее, чем она есть на самом деле.

Однако то, что создает внешнюю нагрузку у одного программиста, необязательно будет создавать ее у другого. Чем больше у вас опыта, тем меньшую когнитивную нагрузку вы будете испытывать при решении задач. Например, в листинге 4.2 представлены два эквивалентных кода на языке Python.

Листинг 4.2. Два примера кода на Python, который выбирает все числа больше 10

```
above_ten = [a for a in items if a > 10]

above_ten = []
for a in items:
    if a > 10: new_items.append(a)
```

Так как оба фрагмента решают одну и ту же задачу, когнитивная нагрузка на мозг у них одинаковая. Однако внешняя когнитивная нагрузка зависит от уже имеющихся у вас знаний: если вы не знакомы со списками, то внешняя нагрузка в первом примере будет намного больше, чем для тех программистов, кто уже имеет опыт работы со списками.

УПРАЖНЕНИЕ 4.1. В следующий раз, когда вы будете читать незнакомый код, постарайтесь проконтролировать когнитивную нагрузку. Когда вам становится трудно обрабатывать код и вы чувствуете необходимость в заметках или в пошаговом выполнении этой задачи, то, скорее всего, вы испытываете высокую когнитивную нагрузку.

В этот момент стоит понять и определить, какие участки кода вызывают когнитивную нагрузку. Для этого вы можете использовать следующую таблицу:

Строка кода	Внутренняя когнитивная нагрузка	Внешняя когнитивная нагрузка

4.2. Способы снижения когнитивной нагрузки

Теперь, когда вы знаете, какими способами код может перегрузить рабочую память, давайте рассмотрим способы снижения когнитивной нагрузки. В оставшейся части этой главы мы рассмотрим три способа, которые помогут сделать чтение сложного кода легче. Первый способ — это способ, с которым вы, возможно, уже знакомы, хотя и слегка в другом контексте, — рефакторинг.

4.2.1. Рефакторинг

Рефакторинг — это перепроектирование кода с целью улучшения его внутренней структуры, но без изменения его интерфейсов. Например, если один из блоков кода слишком длинный, вы можете разделить его на несколько функций, или если в кодовой базе есть дублирование, то вы можете переписать код, собрав все строки с дублированием в одном месте. Например, в следующем коде на Python (листинг 4.3) повторяющиеся вычисления можно поместить в метод.

Листинг 4.3. Код на Python, который выполняет одно и то же вычисление дважды

```
vat_1 = Vat(waterlevel = 10, radius = 1)
volume_vat_1 = math.pi * vat_1.radius **2 * vat_1. water_level
print(volume_vat_1)

vat_1 = Vat(waterlevel = 25, radius = 3)
volume_vat_2 = math.pi * vat_2. radius **2 * vat_2. water_level
print(volume_vat_2)
```

В большинстве случаев рефакторинг выполняется для того, чтобы упростить сопровождение готового кода. Например, устранив повторяющийся кусок кода, вы сможете вносить в дальнейшем изменения в этот код только в одном месте.

Однако код, который будет легок в сопровождении, не всегда может быть удобен при чтении. Давайте рассмотрим пример кода, в котором есть множество вызовов методов из разных частей файла или даже нескольких файлов. Такая архитектура удобна больше для сопровождения кода, т. к. каждая программная фунция имеет свой собственный метод. Однако такой *делокализованный* код загромоздит рабочую память, потому что вам придется искать описание функций во многих местах.

Поэтому иногда вам захочется сделать рефакторинг кода, чтобы упростить его чтение в данный период времени, а не для того, чтобы упростить сопровождение кода в долгосрочной перспективе. Такой рефакторинг называется *когнитивным*. Когнитивный рефакторинг — это изменение кодовой базы без изменения интерфейса, как и в случае обычного рефакторинга. Однако цель когнитивного рефакторинга заключается в том, чтобы сделать код более читаемым для данного читателя в данный момент времени.

Когнитивный рефакторинг иногда может включать в себя *обратный рефакторинг*, который снижает удобство эксплуатации, например *встраивание кода* — реализация метода и копирование тела функции в место вызова. Некоторые интерактивные среды разработки могут выполнять этот рефакторинг автоматически. Если имя метода неясно, например `calculate()` или `transform()`, то встраивание кода может быть особенно полезным. При вызове метода с непонятным для вас именем вам нужно некоторое время для того, чтобы понять, что делает метод, и, возможно, вам нужно будет поработать с этим методом еще несколько раз, прежде чем он отложится в вашей долговременной памяти.

Встраивание кода снижает внешнюю когнитивную нагрузку и может помочь понять код, вызывающий метод. К тому же изучение кода самого метода может помочь вам понять код. В новом контексте вы можете выбрать подходящее имя для метода.

В качестве альтернативы вы можете изменить порядок методов в коде — например, код будет удобен для чтения, если метод будет располагаться рядом с вызовом метода. Конечно, в наше время многие интерактивные среды уже имеют метки для перехода к определениям методов и функций, однако использование этой функции также занимает определенное место в рабочей памяти. Следовательно, это тоже может вызвать внешнюю когнитивную нагрузку.

Когнитивный рефакторинг чаще всего предназначен для одного человека, потому что опыт и знания разных людей будут отличаться. Во многих случаях когнитивный рефакторинг носит лишь временный характер и предназначен для того, чтобы человек мог понять код. Затем, как только человек начинает понимать его, когнитивный рефакторинг не используется.

И хотя всего лишь несколько лет назад этот процесс мог быть большой проблемой, сейчас системы контроля версий используются для большинства баз кода и включены в большинство интерактивных сред разработки, что позволяет с легкостью создать отдельную «ветвь», в которой вы выполняете изменения, упрощающие код. И если что-то из рефакторинга окажется важным в широком смысле, то это всегда можно объединить с основным кодом.

4.2.2. Замена незнакомых языковых конструкций

В оставшейся части главы мы рассмотрим способы, помогающие бороться с тремя возможными источниками замешательства при чтении кода (недостаток знаний, информации и вычислительной мощности). Если в коде есть концепции программирования, с которыми вы не знакомы, то это недостаток знаний. Мы начнем со способа, который может помочь вам в таком случае.

В некоторых ситуациях незнакомые конструкции могут быть представлены в другом, знакомом вам виде. Например, многие современные языки программирования (такие как Java или C++) поддерживают *анонимные функции*, чаще всего называемые лямбда-функциями. Это функции, которым не требуется имя (отсюда и их название — анонимные). Другой пример — генератор списка языка Python. Лямбда-функции и списки очень помогают сделать код короче и читабельнее, однако многие программисты не знакомы с анонимными функциями и читают такой код с большим трудом.

Если код простой и понятный, то лямбда-функции и списки не будут представлять проблему; однако при работе со сложным кодом такие структуры могут приводить к перегрузке рабочей памяти. Малознакомые языковые конструкции увеличивают внешнюю когнитивную нагрузку на рабочую память, поэтому в сложном коде лучше их не использовать.

И хотя некоторые языковые конструкции, которые вы хотите заменить, зависят лишь от вашего опыта и знаний, обычно есть две причины замены кода: во-первых, эти конструкции приводят вас в замешательство, и во-вторых, у них есть простой и понятный для вас эквивалент. Обе причины применимы к лямбда-функциям и спискам, поэтому они являются хорошими примерами для демонстрации. Иногда полезнее заменить их на циклы `for` или `while` — так вы снизите когнитивную нагрузку и начнете больше понимать, что делает код. Для рефакторинга могут также использоваться тернарные операторы.

Лямбда-функции

Код на Java из листинга 4.4 является примером анонимной функции, которая используется как параметр для функции `filter()`. Если вы знакомы с лямбда-функциями, то этот код не вызовет у вас никаких трудностей.

Листинг 4.4. Функция `filter()` на Java, которая использует в качестве аргумента анонимную функцию

```
Optional<Product> product = productList.stream().
            filter(p -> p.getId() == id).
            findFirst();
```

Если вы не знакомы с лямбда-функциями, то код может вызвать слишком большую внешнюю когнитивную нагрузку. Если вы не можете понять лямбда-функцию, то просто перепишите код с использованием обычной функции, как это показано в следующем примере (листинг 4.5).

Листинг 4.5. Функция `filter()` на Java, которая использует в качестве аргумента обычную функцию

```
public static class Toetsie implements Predicate <Product> {
    private int id;

    Toetsie(int id){
        this.id = id;
    }

       boolean test(Product p){
                  return p.getID() == this.id;
}
}

Optional<Product> product = productList.stream().
       filter(new Toetsie(id)).
       findFirst();
```

Генератор списков

Язык Python поддерживает синтаксическую структуру под названием *генератор списков*, который умеет создавать списки на основе других списков. Например, следующий код создает список имен, используя в качестве основы список клиентов (листинг 4.6).

Листинг 4.6. Генератор списков на Python, который создает список на основе другого списка

```
customer_names = [c.first_name for c in customers]
```

При составлении списка можно также использовать фильтры, которые усложнят список — например, можно создать список имен клиентов старше 50 лет (листинг 4.7).

Листинг 4.7. Генератор списка на Python с использованием фильтра

```
customer_names =
     [c.first_name for c in customers if c.age > 50]
```

И хотя тем, кто давно работает со списками, этот код покажется достаточно легким для чтения, тем программистам, которые никогда не имели дела со списками (или работали с ними в рамках сложного фрагмента кода), он покажется трудным. К тому же такой код может вызвать большую нагрузку на рабочую память. Когда это происходит, для облегчения понимания вы можете преобразовать генератор списка в цикл `for` (листинг 4.8).

Листинг 4.8. Цикл `for` на Python, создающий один список на основе другого с использованием фильтра

```
customer_names = []

for c in customers:
    if c.age > 50:
        customer_names.append(c.first_name)
```

Тернарные операторы

Многие языки программирования поддерживают *тернарные операторы*, которые являются сокращением операторов `if`. Обычно они указывают условие — если условие истинно, то следует один результат; если условие ложно, за оператором следует другой результат. Например, в листинге 4.9 представлена строка кода JavaScript, который с помощью тернарного оператора проверяет, является ли логическая переменная `isMember` истинной. Если значение переменной `True`, то тернарный оператор возвращает 2 доллара США; в противном случае возвращается 10 долларов США.

Листинг 4.9. Код JavaScript, использующий тернарный оператор

```
isMember ? '$2.00' : '$10.00'
```

Языки программирования типа языка Python поддерживают тернарные операторы в ином порядке: сначала они показывают результат, если условие истинно, затем условие, а затем результат, если условие ложно. Следующий пример представляет собой строку кода на Python, которая проверяет значение логической переменной `isMember` с помощью тернарного оператора (листинг 4.10). Как и в примере с JavaScript, если значение переменной `True`, то оператор возвращает 2 доллара США, в противном случае возвращается 10 долларов США.

Листинг 4.10. Код на Python, использующий тернарный оператор

```
'$2.00' if is_member else '$10.00'
```

Тернарный оператор несложно понять, учитывая то, что, если вы профессиональный программист, вы точно знакомы с условным оператором. Однако сам факт того, что операция находится в одной строке и порядок аргументов отличается от привычного программисту оператора `if`, может привести к большой внешней когнитивной нагрузке.

Некоторым людям рефакторинг, описанный в предыдущих разделах, может показаться чем-то странным. Вы можете подумать, что сокращение кода с помощью анонимных операторов или тернарных операторов всегда предпочтительнее, потому что так код становится более читабельным, и можете выступать против рефакторинга кода. Однако, как вы могли увидеть в этой главе, «читабельность» кода

каждый определяет сам для себя. Если вы знакомы с шахматными дебютами, то вам будет намного легче запомнить шахматную расстановку. Так же ситуация обстоит и с тернарными файлами: если вы знакомы с ними, то вам будет легко прочитать код с их содержанием. Будет ли вам легко читать код или нет, зависит от имеющихся у вас знаний, и нет ничего плохого или стыдного в том, что вам нужно слегка упростить код для лучшего понимания.

Однако в зависимости от вашей базы кода вы можете захотеть отменить изменения, внесенные в код в целях повышения читабельности кода, но только при условии, что вы действительно понимаете его. Если вы новичок в команде или единственный, кто не знаком с концепцией генератора списков, то вы можете оставить код с внесенными в него изменениями.

4.2.3. Синонимизация — отличное дополнение к дидактическим карточкам

Хотя нет ничего стыдного во временном упрощении кода, это указывает на ограниченность вашего понимания. В *главе 3* вы узнали о применении дидактических карточек для обучения и запоминания (другое их название — *мнемонические карточки* или *словарные карточки*). С одной стороны карточек вы писали подсказку, а с другой стороны — код. Например, для карточки с циклом `for` на одной стороне вы могли написать «выводить все числа от 0 до 10 на C++», а с другой стороны написать соответствующий код на C++ (как это показано в листинге 4.11).

Листинг 4.11. Код на C++ для вывода чисел от 0 до 10

```
for (int i = 0; i <= 10; i = i + 1) {
  cout << i << "\n";
}
```

Если вам часто бывает непонятен список, то вы можете добавить в свой набор карточек пару карточек с этой конструкцией. Для таких сложных концепций программирования лучше писать код с обеих сторон карточки, а не подсказку: т. е. на одной стороне у вас будет простой код, а с другой стороны — эквивалент, в котором используется тернарный или анонимный оператор.

4.3. Вспомогательные средства при перегрузке рабочей памяти

В предыдущем разделе вы познакомились со способом снижения когнитивной нагрузки — это рефакторинг кода до понятной и знакомой формы. Однако даже после рефакторинга код все еще может перегружать вашу рабочую память, например если структура кода слишком сложная. Код со сложной структурой может перегрузить рабочую память двумя способами.

Во-первых, вы можете не знать, какие части кода вам нужно читать. Следовательно, вы читаете больше кода, чем это необходимо (быть может, даже больше, чем ваша память способна обработать).

Во-вторых, если код сложно организован, то ваш мозг будет пытаться делать две вещи одновременно: понимать отдельные строки кода и понимать структуру кода, чтобы разобраться, где именно нужно продолжить чтение кода. Например, вы натыкаетесь на строку с вызовом метода, функционал которого вы не знаете, — в таком случае вы, скорее всего, пойдете искать информацию о методе и только потом продолжите читать сам код.

Если вам когда-нибудь приходилось читать один и тот же фрагмент кода несколько раз подряд, так и не поняв его, то вы, скорее всего, просто не знали, как и что нужно читать, на чем нужно сосредоточить основное внимание. Возможно, вы могли понять каждую строку кода по отдельности, но код в целом оставался для вас непонятным. Когда ваша рабочая память будет на пределе, вы можете воспользоваться специальными приемами, которые помогут вам сосредоточиться на нужных частях кода.

4.3.1. Создание графа зависимостей

Создание графа зависимостей для вашего кода может помочь вам понять логику и прочитать код, следуя логической последовательности. Для данного способа я посоветовала бы вам распечатать код или сохранить его в PDF-файл и открыть на планшете. Выполните следующие действия, чтобы вашей памяти было проще обрабатывать код:

1. Обведите все переменные.

   Как только у вас будет код в форме, позволяющей оставлять комментарии, найдите все переменные и обведите их, как показано на рис. 4.5.

2. Соедините похожие переменные.

   После того как вы найдете все переменные, соедините линиями все экземпляры каждой переменной, как показано на рис. 4.6. Так вы сможете понять, где в программе используются данные. В зависимости от кода, вы можете соединить друг с другом и похожие переменные (например, такие переменные, как `customers[0]` и `customers[i]`).

   Соединение переменных поможет вам читать код, т. к. вы сможете просто следовать по линии, а не искать все появления переменной. Таким образом вы снизите когнитивную нагрузку и освободите рабочую память, что позволит вам сосредоточиться на работе кода.

3. Обведите все вызовы методов и функций.

   После того как вы найдете все переменные, сосредоточьте внимание на методах и функциях. Обведите их, используя другой цвет.

4. Соедините методы или функции с их определениями.

   Соедините каждое определение метода или функции и место их вызова. Особое внимание уделите методам, которые вызываются только один раз, — такие ме-

```
from itertools import islice

digits = "0123456789abcdefghijklmnopqrstuvwxyz"

def baseN(num,b):
  if num == 0: return "0"
  result = ""
  while num != 0:
    num, d = divmod(num, b)
    result += digits[d]
  return result[::-1] # reverse

def pal2(num):
    if num == 0 or num == 1: return True
    based = bin(num)[2:]
    return based == based[::-1]

def pal_23():
    yield 0
    yield 1
    n = 1
    while True:
      n += 1
      b = baseN(n, 3)
      revb = b[::-1]
      #if len(b) > 12: break
      for trial in ('{0}{1}'.format(b, revb), '{0}0{1}'.format(b, revb),
                    '{0}1{1}'.format(b, revb), '{0}2{1}'.format(b, revb)):
          t = int(trial, 3)
          if pal2(t):
              yield t

for pal23 in islice(pal_23(), 6):
    print(pal23, baseN(pal23, 3), baseN(pal23, 2))
```

Рис. 4.5. Код, в котором для упрощения понимания все переменные обведены

тоды можно будет встроить в код при рефакторинге (подробнее об этом говорилось ранее в этой главе).

5. Обведите все экземпляры классов.

   После того как вы найдете все методы и функции, обратите внимание на классы. Обведите их все еще каким-нибудь цветом.

6. Соедините классы с экземплярами.

   На последнем шаге изучения кода нужно соединить все экземпляры одного класса с его определением, если оно есть в коде. Если определения нет, то вы можете связать друг с другом экземпляры одного класса.

Получившийся «узор» определяет потоки выполнения кода и может использоваться в качестве вспомогательного средства при чтении кода. Теперь структура кода у вас всегда перед глазами, и вам не нужно будет искать, к примеру, определения, а также расшифровывать значение кода. Следовательно, вы не сможете перегрузить

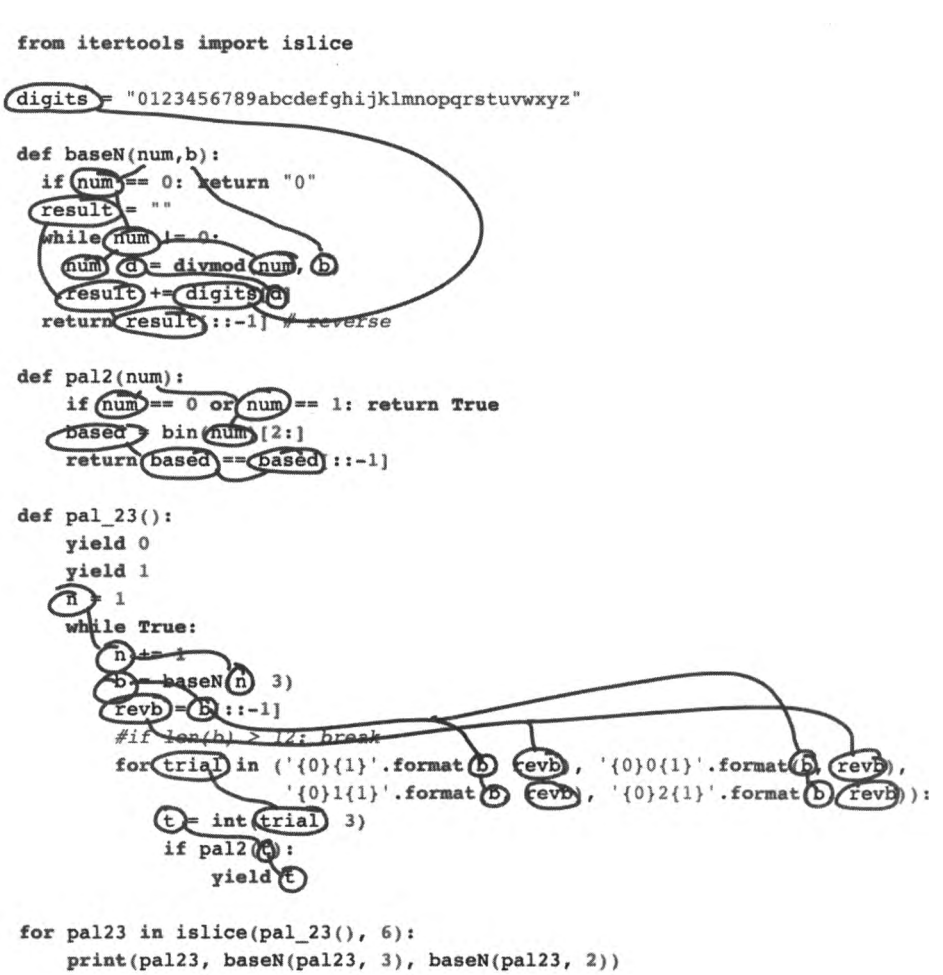


Рис. 4.6. Код, в котором для упрощения понимания все переменные выделены и соединены друг с другом

рабочую память. Вы можете начать с самого начала кода, например метода `main()`, и читать его с этого места. Каждый раз, когда вы увидите ссылку на метод или создание экземпляра класса, вы проследите за линией и продолжите чтение в нужном месте, не тратя время на поиск и чтение всего кода.

4.3.2. Использование таблицы состояний

Даже если код в результате рефакторинга приведен к самой простой форме со всеми отмеченными зависимостями, он все равно может приводить вас в замешательство. Иногда причиной замешательства является не структура кода, а выполняемые в коде вычисления. Это проблема недостатка вычислительной мощности.

Например, давайте обратимся к коду на языке BASIC из *главы 1*, который преобразует число `N` в двоичное представление. В этой программе все переменные влияют друг на друга, так что для понимания кода вам нужно знать значения переменных.

Для этого вы можете использовать вспомогательные средства запоминания: например, граф зависимостей, который будет особенно полезен для кода, который выполняет сложные вычисления. Однако есть еще одно средство, которое может помочь вам с таким кодом, — *таблица состояний.*

Таблица состояний фокусирует ваше внимание на значениях переменных, а не на структуре кода. В таблице есть столбцы для каждой переменной, а также строки, в которых указывается каждый шаг выполнения кода. Еще раз взгляните на пример кода на BASIC (листинг 4.12). Код приводит в замешательство, потому что вы не можете отследить промежуточные шаги и вычисления, а также то, как они влияют на переменные.

Листинг 4.12. Код на BASIC, преобразующий число N в двоичное представление

```
1 LET N2 = ABS (INT (N))
2 LET B$ = ""
3 FOR N1 = N2 TO 0 STEP 0
4 LET N2 = INT (N1 / 2)
5 LET B$ = STR$ (N1 - N2 * 2) + B$
6 LET N1 = N2
7 NEXT N1
8 PRINT B$
9 RETURN
```

Если вам нужно понять подобный код, в котором есть множество связанных между собой вычислений, то используйте таблицу состояний, пример которой показан на рис. 4.7.

	N	N2	B$	N1
Init	7	7	—	7
Loop1		3	1	3
Loop2				

Рис. 4.7. Пример таблицы состояний программы на языке BASIC для определения числа в двоичном представлении

Для создания таблицы состояний выполните следующие действия:

1. Составьте список всех переменных. Если вы уже создали граф зависимостей для этой программы с помощью способа, описанного в предыдущем разделе, вам будет легко составить список переменных, потому что они у вас все обведены одним цветом.
2. Создайте таблицу и выделите для каждой переменной свой столбец. В таблице состояний у каждой переменной должен быть свой столбец, в который будут записываться ее промежуточные значения, как это показано на рис. 4.7.
3. Добавьте одну строку для каждого шага выполнения кода. Скорее всего код, содержащий сложные вычисления, будет иметь сложные зависимости, например цикл, зависящий от вычислений, или составной оператор `if`. Например, как показано на рис. 4.7, строка может представлять собой одну итерацию в цикле,

которой будет предшествовать инициализационный код. В качестве альтернативы строка может представлять ответвление в операторе `if` или группу последовательных строк кода. В очень сложном и коротком коде одна строка в таблице может представлять одну строку кода.

4. Выполните каждую часть кода, а затем запишите измененное значение каждой переменной в нужной строке и столбце.

После того как таблица будет готова, вернитесь к коду. Вычислите значение каждой переменной и запишите его в таблицу. Процесс мысленного выполнения кода называется *трассировкой* или *когнитивной компиляцией*. При работе с кодом и таблицей у вас может возникнуть желание пропустить несколько переменных и заполнить лишь часть таблицы, однако постарайтесь заполнить таблицу полностью. От того, как хорошо вы проработаете код, зависит то, как хорошо вы его поймете. При этом таблица будет поддерживать вашу рабочую память и не допускать ее перегрузки. Когда вы будете читать программу во второй раз и будете использовать таблицу в качестве справочного материала, вы сможете сосредоточиться на самой программе, а не на выполняемых вычислениях.

Приложение для тренировки рабочей памяти

Создание визуализаций для поддержания рабочей памяти вручную имеет большое значение, т. к. при этом вы детально изучаете код. Однако визуализации можно создавать автоматически. Python Tutor — программа, созданная Филиппом Го (Philip Guo), профессором когнитивистики Калифорнийского университета, Сан-Диего. Python Tutor, доступный для многих языков программирования в дополнение к Python, визуализирует выполнение программы. Например, на рис. 4.8 показано, что Python поразному хранит целые числа и списки. Например, значение целого числа сохраняется, в то время как для списка используется система типа указатель.

Исследование использования программы в образовании[1] показало, что студентам нужно время для привыкания к работе с программой, но при этом она особенно полезна при отладке программ.

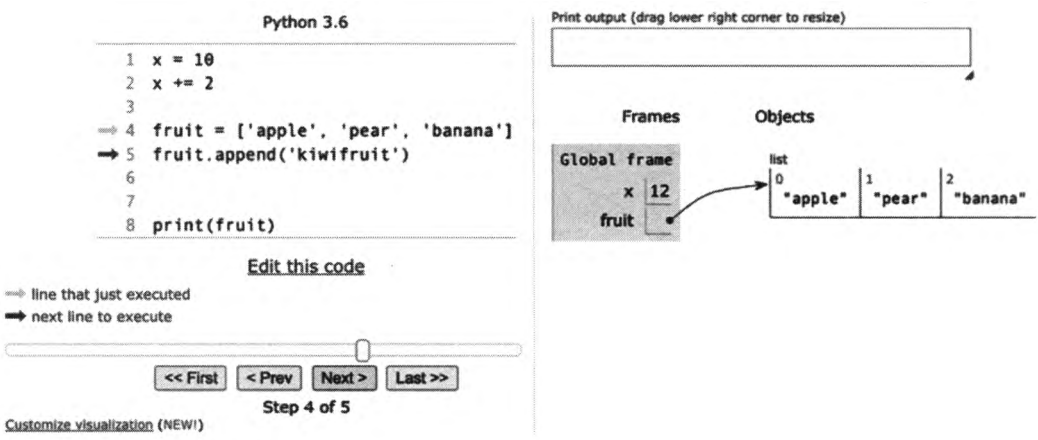


Рис. 4.8. Python Tutor показывает различия между хранением целого числа x и хранением списка `fruit` с указателем

[1] См. «The Use of Python Tutor on Programming Laboratory Session: Student Perspectives», Оскар Карнелим и Мевати Аюб (2017), **https://kinetik.umm.ac.id/index.php/kinetik/article/view/442**.

4.3.3. Сочетание графов зависимостей и таблиц состояний

В *разд. 4.2* и *4.3* данной главы описываются два способа поддержки рабочей памяти при чтении кода путем выписывания информации о коде на бумагу: граф зависимостей и таблица состояний. Эти способы применяются для разных аспектов кода: граф зависимостей обращает ваше внимание на структуру кода, а таблица состояний фиксирует все вычисления.

При чтении незнакомого кода вы можете опираться на два следующих упражнения: таким образом вы получите полное представление о работе кода и попрактикуетесь в использовании вспомогательных средств запоминания.

УПРАЖНЕНИЕ 4.2. Следуя шагам, описанным в предыдущих разделах, создайте граф зависимостей и таблицу состояний для каждой из следующих программ на Java.

Программа 1

```
public class Calculations {
    public static void main(String[] args) {
        char[] chars = {'a', 'b', 'c', 'd'};
        // поиск bba
        calculate(chars, 3, ı -> i[0] == 1 && ı[1] == 1 && i[2] == 0);
    }
    static void calculate(char[] a, int k, Predicate<int[]> decider) {
        int n = a.length;
        if (k < 1 || k > n)
            throw new IllegalArgumentException("Forbidden");

        int[] indexes = new int[n];
        int total = (int) Math.pow(n, k);

        while (total-- > 0) {
            for (int i = 0; i < n - (n - k); i++)
                System.out.print(a[indexes[ı]]);
            System.out.println();

            if (decider.test(indexes))
                break;

            for (int ı = 0; ı < n; i++) {
                if (indexes[i] >= n - 1) {
                    indexes[ı] = 0;
                } else {
                    indexes[ı]++;
                    break;
                }
```

```
            }
        }
    }
}
```

Программа 2

```
public class App {
    private static final int WIDTH = 81;
    private static final int HEIGHT = 5;

    private static char[][] lines;
    static {
        lines = new char[HEIGHT][WIDTH];
        for (int i = 0; i < HEIGHT; i++) {
            for (int j = 0; j < WIDTH; j++) {
                lines[i][j] = '*';
            }
        }
    }

    private static void show(int start, int len, int index) {
        int seg = len / 3;
        if (seg == 0) return;
        for (int i = index; i < HEIGHT; i++) {
            for (int j = start + seg; j < start + seg * 2; j++) {
                lines[i][j] = ' ';
            }
        }
        show(start, seg, index + 1);
        show(start + seg * 2, seg, index + 1);
    }

    public static void main(String[] args) {
        show(0, WIDTH, 1);
        for (int i = 0; i < HEIGHT; i++) {
            for (int j = 0; j < WIDTH; j++) {
                System.out.print(lines[i][j]);
            }
            System.out.println();
        }
    }
}
```

Выводы

- Когнитивная нагрузка — это предел того, что может обработать рабочая память. Если вы испытываете большую когнитивную нагрузку, то вы не сможете правильно обработать код.
- Существуют два типа когнитивной нагрузки, которые можно встретить во время программирования: внутренняя когнитивная нагрузка, возникающая из-за сложной структуры кода, и внешняя когнитивная нагрузка, возникающая либо случайно, либо из-за недостатка знаний и опыта у человека, читающего код.
- Рефакторинг — это способ уменьшить внешнюю когнитивную нагрузку с помощью изменения кода с учетом вашего предыдущего опыта и знаний.
- Создание графа зависимостей помогает понять фрагмент сложного кода, состоящего из нескольких связанных между собой частей.
- Создание таблицы состояний, в которую записываются промежуточные значения переменных, помогает читать код, который требует сложных вычислений.

Часть II

Продолжаем думать о коде

В *части I* книги мы рассмотрели роль долговременной, кратковременной и рабочей памяти в обработке кода. Мы определили, что мы знаем об изучении синтаксиса и концепций программирования, а также узнали способы поддержания нашего разума и мозга при чтении кода.

В *части II* мы сконцентрируем все внимание не на чтении кода, а на размышлении о коде: как добиться глубокого понимания программы и избегать ошибок в мышлении.

5 Совершенствуем навыки углубленного понимания кода

В этой главе:

- вы изучите роли, которые переменные играют в коде;
- сравните поверхностные знания и понимание решений автора кода;
- сравните чтение и изучение естественного языка с чтением и изучением кода;
- обучитесь разным стратегиям углубленного понимания кода.

Ранее мы обсуждали использование дидактических карточек и интервальные повторения как способы изучения синтаксиса кода, а также рассматривали стратегии быстрого ознакомления с кодом — например, выделение переменных и взаимосвязей между ними. Знание синтаксиса и понимание связей между переменными — это важный шаг к понимаю кода, однако существуют и другие проблемы, с которыми сталкиваются программисты при чтении кода.

Когда вы читаете незнакомый фрагмент кода, вам может быть сложно понять, что он делает. Теперь, прочитав *часть I* книги, вы можете сказать, что при чтении незнакомого кода вы испытываете высокую когнитивную нагрузку. Мы уже знаем, что когнитивную нагрузку можно снизить путем изучения синтаксиса, новых концепций программирования, а также с помощью переписывания кода.

Как только вы поймете, что делает код, вы перейдете к следующему шагу — более углубленному пониманию кода. Как именно его написали? Куда можно добавить новую функцию? А можно ли как-то иначе написать этот код?

В предыдущих главах мы рассматривали схемы, или то, как наши воспоминания хранятся в мозге. Все воспоминания связаны между собой. Вы можете воспользоваться этой связью при работе с кодом, т. к. воспоминания, хранящиеся в долговременной памяти, помогают рабочей памяти думать о коде.

Размышление о коде — это главная тема данной главы, в которой мы научимся понимать код углубленно. Мы рассмотрим три стратегии, которые позволят понимать код углубленно, а также способы, которые помогают размышлять об идеях, мыслях и решениях автора кода. Для начала мы рассмотрим структуру, которая поможет вам понять код. Затем мы рассмотрим разные уровни понимания, а также способы чтения кода, основанные на нашем опыте чтения естественного языка. Недавние исследования показывают, что необходимые для чтения кода навыки тесно связаны с навыками, которые мы используем для чтения на естественном языке. Из этого можно сделать вывод, что мы, программисты, можем опираться на свой опыт чтения на естественном языке для углубленного понимания кода.

5.1. Роли переменных

Когда мы думаем о коде, то понимаем, что переменные играют главную роль. Понимание того, какая информация содержится в переменных, позволяет программисту рассуждать о коде и вносить в него изменения. Если вы не знаете, что означает та или иная переменная, то код вызовет у вас определенные сложности. Вот почему понятные имена переменных могут помочь нам углубленно понять код.

Согласно словам профессора Йорма Сайаниеми (Jorma Sajaniemi) из Университета Восточной Финляндии, причина плохого понимания переменных заключается в том, что у многих программистов отсутствует в долговременной памяти схема, связывающая между собой переменные. Сайаниеми утверждает, что мы склонны использовать либо слишком общие термины, как, например, «переменная» или «целое число», либо слишком узконаправленные термины, например конкретное имя переменной `number_of_customers`. Считая, что программистам нужно нечто среднее, он создал структуру *ролей переменных*. Роль переменной показывает, что переменная делает в коде.

5.1.1. Разные переменные выполняют разные действия

Давайте рассмотрим различные роли переменных на примере следующей программы на Python. Функция `prime_factors(n)` возвращает количество простых множителей, на которые можно разложить `n`:

```
upperbound = int(input('Upper bound?'))
max_prime_factors = 0
for counter in range(upperbound):
    factors = prime_factors(counter)
    if factors > max_prime_factors:
        max_prime_factors = factors
```

Этот код содержит четыре переменные: `upperbound`, `counter`, `factors` и `max_prime_factors`. Однако если мы будем описывать этот пример как код с четырьмя переменными, то эта информация никак нам не поможет в понимании кода, потому что формулировка слишком абстрактная. Имена переменных могут немного помочь понять код, однако все равно остаются непонятные места. Например, у пе-

ременной `counter` все еще слишком «общее» имя. Эта переменная отображает постоянное количество объектов или ее значение будет изменяться? Для лучшего понимания кода стоит изучить роли, которые играет каждая из четырех переменных.

В этом коде пользователя просят ввести значение, которое будет храниться в переменной `upperbound`. После этого цикл будет выполняться до тех пор, пока не достигнет верхней границы переменной `counter`. Переменная `factors` содержит в себе текущее значение количества простых множителей для текущего значения переменной `counter`. Наконец, переменная `max_prime_factors` отображает наибольшее число, которое может быть получено при выполнении цикла.

Структура ролей переменных отражает различие переменных в их поведении. Переменная `upperbound` играет роль *держателя последнего значения*, который хранит последнее введенное предельное значение. Переменная `counter` представляет собой *счетчик* и отвечает за повторение цикла. Переменная `max_prime_factors` — это *держатель искомого значения*; эта переменная хранит искомое значение. Переменная `factors` — это держатель последнего значения; переменная хранит в себе последнее значение количества простых множителей. В следующем разделе я подробно объясню эти и другие роли в структуре Сайаниеми.

5.1.2. Одиннадцать ролей, охватывающие почти все переменные

Как мы увидели в предыдущем примере, переменные играют типовые роли. Во многих программах есть переменные-счетчики или переменные-держатели искомого значения. Сайаниеми утверждает, что с помощью всего 11 ролей можно описать практически все переменные:

- *Фиксированное значение* — это переменная, значение которой не меняется. Это может быть как постоянное значение, если выбранный вами язык программирования позволяет фиксировать значения, так и переменная, которая инициализируется только один раз, а затем ее значение остается неизменным. Примерами переменных с фиксированным значением являются математические константы, например число π, а также данные из файла или базы данных.
- *Счетчик* — при итерации цикла всегда есть переменная, которая перебирает список значений. Это счетчик, значение которого можно предсказать сразу после начала последовательности. Это может быть как целое число, например `i`, повторяющееся в цикле `for`, так и более сложный счетчик, например `size = size / 2` в бинарном поиске, где размер искомого массива на каждой итерации уменьшается вдвое.
- *Флаг*[1] — это переменная, которая используется для обозначения того, что что-то произошло или имеет место. Например, переменная `is_set`, `is_available` или

[1] В концепции Сайаниеми данная роль называется «односторонний флаг», однако это понятие мне кажется слишком узким.

`is_error`. Чаще всего флаги содержат логическое значение, однако они могут быть целыми числами или даже строками.

- *Бродяга* — это переменная, которая, как и счетчик, просматривает структуру данных. Разница заключается лишь в способе просмотра. Счетчик всегда выполняет итерацию по уже известному списку значений, например в цикле `for` на Python: `for i in range (0, n)`. Бродяга — переменная, которая просматривает структуру данных путем, который до начала цикла остается неизвестным. В зависимости от языка программирования эта переменная может быть как указателем, так и целым индексом. Такие переменные могут перемещаться по спискам, например бинарному списку, однако чаще всего они находятся в структуре данных, например дереве. Примером может служить переменная, которая просматривает нужный список и ищет место, куда должен быть добавлен новый элемент, или ищет индекс в двоичном дереве.
- *Держатель последнего значения* — это переменная, которая содержит последнее значение, обнаруженное при просмотре всех значений. Например, это может быть последняя прочитанная строка из файла (`line = file.readline ()`) или копия элемента массива, на который в последний раз ссылался счетчик (`element = list [i]`).
- *Держатель искомого значения* — очень часто вы просматриваете список значений для поиска нужного значения. Переменная, содержащая нужное вам значение или очень близкая к нему, называется держателем искомого значения. Одним из классических примеров такой переменной является переменная, хранящая в себе минимальное, максимальное или первое значение, которое подходит под определенное условие.
- *Накопитель* — это переменная, которая собирает данные и затем объединяет их в одно значение. Это может быть переменная с начальным значением 0, которая при каждом повторении цикла добавляет значения, например:

```
sum = 0
for i in range(list):
    sum += list[i]
```

  Однако значение переменной может быть вычислено и на функциональных языках или языках, которые охватывают некоторые функциональные аспекты: `function_total = sum (list)`.
- *Контейнер* — это любая структура данных, содержащая в себе несколько элементов, которые можно добавлять или удалять. Примерами такой переменной являются списки, массивы, стеки и деревья.
- *Последователь* — для некоторых алгоритмов необходимо отслеживание предыдущего и последующего значений. Переменная в этой роли называется последователем, и она всегда связана с другой переменной. Примерами таких переменных являются указатели, указывающие на предыдущий элемент в списке, а также нижний индекс в бинарном поиске.
- *Преобразователь* — иногда для дальнейшей работы переменную надо преобразовать. Например, в некоторых языках программирования вы не сможете полу-

чить доступ к символам в строке до тех пор, пока не преобразуете строку в массив символов. Другой пример — вам нужно сохранить отсортированную версию исходного списка. Все это примеры преобразователей, которые представляют собой переменные, использующиеся лишь для переупорядочивания или хранения значений. Чаще всего такие переменные являются временными.

- *Временная переменная* — это переменная, которая используется в течение короткого времени и часто имеет имя `temp` или `t`. Такие переменные применяются, когда нужно поменять местами данные, а также для хранения результата вычислений, который используется в методе или функции много раз.

На рис. 5.1 представлены все 11 ролей структуры Сайаниеми. Рисунок помогает понять, какую роль может играть переменная.

Рис. 5.1. Вы можете воспользоваться этой схемой для определения роли переменной в фрагменте кода

5.2. Роли и принципы

Роли не ограничиваются определенными принципами программирования, однако они есть во всех принципах. На примере накопителя мы уже узнали, что такая роль может встретиться в функциональном языке. Вы увидите, что переменные играют роли, перечисленные в предыдущем разделе, и в объектно-ориентированном программировании. Давайте рассмотрим следующий класс в Java:

```
public class Dog {
  String name;
  int age;
  public Dog (String n) {
    name = n;
    age = 0;
  }
```

```
  public void birthday () {
    age++;
  }
}
```

У экземпляров класса `Dog` есть два атрибута: `name` и `age`. Значение атрибута `name` после инициализации не меняется, это *фиксированное значение*. Переменная `age` имеет такой же принцип работы, как переменная `counter` в коде на Python, который мы рассматривали ранее: переменная имеет начальное значение 0 и с каждым днем рождения увеличивается, так что у этой переменной роль — *счетчик*.

5.2.1. Польза ролей

Многие опытные программисты в той или иной степени знакомы (возможно, под другими названиями) со структурой ролей Сайаниеми. Основной целью структуры ролей является пополнение словарного запаса программиста для работы с переменными, а не создание новых концепций и подходов. Структура ролей, особенно если ее используют в команде разработчиков, может улучшить понимание кода и обмен идеями о нем.

Новичкам тоже может быть полезно ознакомление с ролями переменных. Исследования показали, что структура ролей помогает студентам мысленно обрабатывать код, а также то, что студенты, использующие роли переменных, добиваются более высоких результатов, чем другие студенты[2]. Одна из причин эффективности заключается в том, что чаще всего группа ролей соотносится с определенным типом кода. Например, программа со счетчиком и держателем искомого значения является программой поиска.

УПРАЖНЕНИЕ 5.1. Самое время попрактиковаться в использовании структуры ролей переменных. Найдите незнакомый код и изучите переменные, записывая для каждой из них следующее:

- ❒ имя переменной;
- ❒ тип переменной;
- ❒ операции, в которых переменная играет роль;
- ❒ роль переменной согласно структуре Сайаниеми.

Заполните таблицу для каждой переменной, которую вы найдете в коде.

Имя переменной	Тип	Операция	Роль

[2] Например, «An Experiment on Using Roles of Variables in Teaching Introductory Programming», Йорм Сайаниеми и Марйа Куиттинен (2007), **www.tandfonline.com/doi/full/10.1080/08993400500056563**.

После того как вы заполните таблицу, подумайте о том, почему вы выбрали для каждой переменной именно эту роль. Что повлияло на ваше решение? Может, это было имя переменной, способ ее работы, комментарий в коде или ваш собственный опыт работы с кодом?

Практические советы по работе с ролями переменных

При чтении незнакомого кода я поняла, что бывает полезным распечатать код на бумаге или сохранить его в PDF-файле, в котором можно делать заметки. Понимаю, что такой подход может показаться странным и, конечно, у него есть некоторые недостатки: например, вы не сможете выполнить поиск по коду, однако возможность делать заметки помогает углубленно понимать код.

Я анализировала код на бумаге совместно со многими опытными программистами, и как только они преодолели некоторые барьеры, то поняли, что это упражнение особенно полезно. Конечно, при работе с большой программой вы не можете распечатать весь код сразу, однако вы можете начать с одного класса или части программы. Если не получается распечатать код полностью из-за его размера или по другим причинам, то многие из описанных здесь способов подходят для работы в интегрированной среде разработки с использованием комментариев.

При выполнении упражнения 5.1 мне удобнее распечатать код и отмечать роль каждой переменной специальным значком (рис. 5.2).

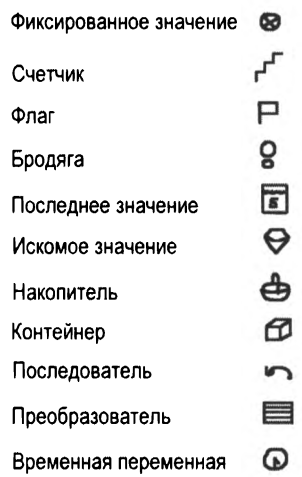


Рис. 5.2. Вы можете создать набор значков для 11 ролей, которые могут играть переменные в соответствии со структурой ролей переменных Сайаниеми Вот значки, которые использую я сама

Запомнив значки, вы получите отличное подспорье для работы с кодом. Чтобы легче было запомнить значки, сделайте набор дидактических карточек.

На рис. 5.3 изображен предыдущий код на Python с обозначенными ролями переменных.

Когда вы пишете код, вы можете указать роль переменной в ее имени, особенно если ваши коллеги знакомы со структурой ролей переменной. И хотя имя перемен-

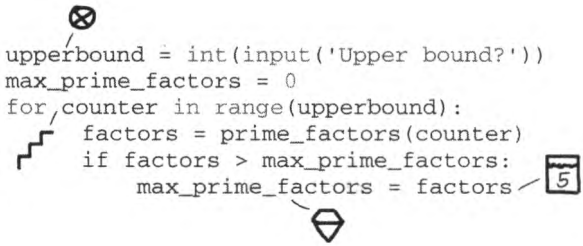


Рис. 5.3. Фрагмент кода на Python со значками, обозначающими роли переменных в коде Переменная `upperbound` — это держатель последнего значения, `counter` является счетчиком, а переменная `max_prime_factors` является держателем искомого значения

ной станет длиннее, вы сможете передать важную информацию, а читателю не нужно будет самостоятельно определять роль переменной.

5.2.2. Венгерская нотация

Структура ролей переменных могла чем-то напомнить вам *венгерскую нотацию*. Идея венгерской нотации состоит в том, чтобы тип переменной был понятен уже по ее имени, например: `strName` — это строка, содержащая какое-то имя, а `lDistance` — это 64-разрядное число, содержащее некое расстояние. Это соглашение изначально было создано для языков, в которых отсутствует система кодирования типов переменных.

Венгерская нотация была описана Чарльзом Симони (Charles Simonyi) в его докторской диссертации 1976 года «Meta-Programming: A Software Production Method», которую даже в наше время стоит почитать. Симони работал в Microsoft, где он руководил разработкой Excel и Word. Его соглашение об именах стало стандартом для программного обеспечения, разработанного Microsoft, а затем и для программного обеспечения, разработанного на языках Microsoft, например на Visual Basic.

Впервые венгерская нотация широко использовалась в 1970-х годах в языке программирования BCPL, который многие считают предком языка C. В то время, когда еще не было интегрированных сред разработки с технологией IntelliSense, было невозможно с помощью редактора увидеть тип переменной. Соответственно добавление типа переменной к ее имени улучшало чтение и понимание кода в целом. Проблема заключалась в том, что из-за этого имена переменных становились длиннее, и поэтому их было трудно читать. А если возникала необходимость в смене типа переменной, то это могло повлиять на многие имена переменных в коде. Так как теперь большинство редакторов могут без труда отобразить тип переменной, венгерская нотация потеряла свое значение. Такое кодирование типа переменной в ее имени теперь редко где-либо применяется, и сегодня использование венгерской нотации нежелательно.

Системная и прикладная венгерские нотации

Простое кодирование типов переменных в их именах было не тем, о чем писал в своей диссертации Симони. По правде говоря, кодирование типов переменных в их имени — это то, что мы теперь называем *системной венгерской нотацией*.

Венгерская нотация Симони была семантической информацией. Теперь это называется *прикладной венгерской нотацией*. В ней у префиксов есть конкретное значение, а не просто тип переменной. Например, в своей диссертации Симони предлагает использовать `cX` для подсчета `X` (так что `cColors` можно использовать для обозначения количества цветов в пользовательском интерфейсе) и `lX` для обозначения длины массива, как в `lCustomers`. Причина, по которой эта нотация называется прикладной, заключается в работе Симони над Word и Excel в Microsoft. В кодовой базе Excel есть много переменных с префиксом `rw` и `col`, и они служат хорошим примером использования венгерской нотации. Значения строк и столбцов — это целые числа, однако для удобства чтения кода лучше иметь возможность определить переменную по ее имени.

По не совсем ясным причинам команда разработчиков Windows приняла эту нотацию, но только для типов данных, а не для семантической информации. Джоэл Спольски (Joel Spolsky), работавший над Excel до основания им платформы Stack Overflow, приписывал неверное толкование венгерской нотации тому факту, что для объяснения роли префикса Симони использует слово «тип» вместо «вид»[3].

Если вы решите посмотреть оригинальную работу Симони, то там он объясняет типы на той же странице, где он объясняет примеры, не относящиеся к типам, например `cX` для счета. Я думаю, что изначально небольшая группа людей или даже один человек неправильно использовали эту систему, и потом она в таком виде распространилась среди других программистов. Как мы увидим далее в *главе 10*, люди часто придерживаются соглашений, если они есть в коде. Как бы то ни было, неправильное понимание венгерской нотации было распространено среди разработчиков Windows — по большей части из-за книги Чарльза Петцольда «*Программирование для Windows*», изданной в 1998 году, — а затем кто-то сказал: «Использование венгерской нотации ущербно», остальное всем известно.

Однако я считаю, что идеи Симони не утратили своей ценности. Некоторые предложения, выдвинутые в прикладной венгерской нотации, очень похожи на структуру ролей переменных Сайамиени. Например, Симони использовал префикс `t` для обозначения временного промежутка, а также предлагал `min` и `max` в качестве префиксов для обозначения минимального и максимального значений в массиве, которые по сути являются типичными примерами держателей искомого значения из структуры ролей переменных Сайамиени. К сожалению, основное преимущество венгерской нотации, состоявшее в том, что для работы с переменными требовалось меньше умственных усилий, было потеряно из-за неверно сложившегося понимания цели соглашения об именах.

---

[3] См. «Making Wrong Code Look Wrong», Джоэл Спольски (11 мая 2005 г.), **www.joelonsoftware.com/2005/05/11/making-wrong-code-look-wrong/**.

5.3. Углубленное понимание программ

В этой главе мы видели, что определение ролей переменных помогает нам понимать код. В *главе 4* я показала еще один способ, позволяющий быстро понять код: это выделение всех переменных и определение отношений между ними. Эти способы чрезвычайно полезны, но они *локальные*: они помогают понимать лишь отдельные фрагменты кода. Сейчас мы сосредоточимся на способах, которые помогают углубленно понимать весь код. Какой была цель создателя этого кода? Чего он пытался достичь, а также какие решения он принял в этом процессе?

5.3.1. Понимание текста и понимание плана

Разделение уровней понимания было целью Нэнси Пеннингтон (Nancy Pennington), профессора психологии Университета штата Колорадо. Она создала модель с двумя разными уровнями, на которых программист может понимать исходный код: *понимание структуры текста* и *понимание плана*.

Согласно модели Пеннингтон, понимание структуры текста связано с пониманием частей программы на поверхностоном уровне, например знание того, для чего нужно то или иное ключевое слово или какова роль переменной. С другой стороны, понимание плана связано с пониманием того, что автор программы планировал и чего хотел достичь. Цели автора кода не только скрыты в переменных и их ролях, но и становятся очевидными при исследовании структуры и внутренних связей кода. В следующих разделах вы узнаете, как понимать цель и смысл кода на углубленном уровне.

5.3.2. Этапы понимания программы

Понимание плана говорит о том, что вы понимаете то, как и почему связаны между собой разные части кода. В этом разделе вы узнаете о теоретических основах детального понимания кода, а также выполните упражнения, которые помогут вам быстро находить поток.

Джонатан Силлито (Jonathan Sillito), профессор Университета Бригама Янга, выделил четыре этапа, на которых программист понимает код[4]. По словам Силлито, наблюдавшего за 25 читающими код программистами, обычно программисты начинают искать в коде *точку фокуса*. Это может быть как начало кода, например метод `main()` в программе на Java или метод `onLoad()` в веб-приложении. Это может быть строка, заинтересовавшая программиста: например, строка с ошибкой или строка, помеченная профайлером как использующая много ресурсов.

С этой точки фокуса программисты начинают анализировать код. Программист может выполнить код и поставить точку останова в этой строке, или проверить код путем выполнения поиска в базе кода всех вхождений некоторых переменных, или

[4] См. «Questions Programmers Ask During Software Evolution Tasks», Джонатан Силлито, Гейл Мерфи и Крис де Волдер (2006), **www.cs.ubc.ca/~murphy/papers/other/asking-answering-fse06.pdf**.

с помощью функций интегрированной среды разработки перейти к другим местам в коде с помощью этой строки.

Понимание программистом кода растет именно отсюда, переходя в понимание более широкой концепции: например, понимание того, что будет результатом работы функции, или знание того, какие поля есть у определенного класса. На заключительном этапе программист имеет полное понимание всей программы, например понимание того, что фокальная линия является частью определенного алгоритма, или понимание всех подклассов класса.

Итак, каждый программист, желая углубленно понять программу, проходит четыре этапа, которые заключаются в следующем:

1. Найти точку фокуса.
2. Анализировать код, начиная от точки фокуса.
3. Понять концепцию из набора связанных объектов.
4. Понять концепции из всего множества объектов.

Точка фокуса кода является важной частью при чтении кода. Проще говоря, вы должны знать, с чего следует начинать читать код. Некоторые техники и методы, например техника внедрения зависимостей, могут фрагментировать точки фокуса, в результате чего точки фокуса оказываются в разных частях кода и их сложно связать друг с другом. Чтобы знать, откуда начинать читать код, вам нужно понять, как именно связаны части кода.

Такая ситуация может вызвать у читателя (да и у автора) сомнение в понимании структуры всей системы, даже если каждая строка кода выглядит понятной. Это пример ситуации, когда у программиста есть *понимание текста*, но нет *понимания плана*. Такая ситуация может расстраивать, потому что вы чувствуете, что должны знать, что именно делает код (он не выглядит сложным), но вы не можете понять его структуру.

Применение этапов углубленного понимания

Теперь, когда вы понимаете разницу между знанием текста и знанием плана, давайте вернется к способу, показанному в *главе 4*, который снижает когнитивную нагрузку при чтении сложного кода:

1. Обведите все переменные.
2. Соедините похожие переменные.
3. Обведите все вызовы методов или функций.
4. Соедините методы или функции с их определениями.
5. Обведите все экземпляры классов.
6. Соедините классы с экземплярами.

Может быть, вы заметили, что эти шесть шагов похожи на четыре этапа, выделенные Силлито. Разница состоит в том, что у шагов в *главе 4* не было точки фокуса и способ применялся ко всем переменным, методам и экземплярам классов. Если

вы хотите получить углубленное понимание определенного фрагмента кода, то выполняйте эти шаги, но для конкретной точки фокуса.

Опять же лучше код распечатать и вручную выделить его части. В качестве альтернативы вы можете работать в интегрированной среде разработки и добавлять комментарии к строкам кода. Давайте рассмотрим четыре этапа понимания плана кода более подробно:

1. Найдите точку фокуса.

   Начните просмотр и изучение кода с определенной точки. Это может быть как метод `main()`, так и определенная часть кода, например строка, помеченная профайлером как использующая много ресурсов.

2. Анализируйте код, начиная от точки фокуса.

   Ищите взаимосвязи. Начните с точки фокуса, затем обведите все необходимые объекты (переменные, методы и классы), которые играют роль. Вы можете соединить между собой похожие переменные, например получение доступа к одному списку — `customers[0]` и `customers[1]`. Также обратите внимание на методы и функции, на что они ссылаются.

   То, что вы сейчас выделяете, называется *срезом* кода. Срез строки кода *X* определяется как все строки кода, которые транзитивно относятся к строке *X*.

   Фокусировка на срезе помогает понять, где в программе используются данные. Например, теперь вы можете спросить себя, есть ли определенная строка или метод, связанные с точкой фокуса. Где они связаны? Это место может стать новой точкой фокуса для более углубленного изучения кода. Какие части кода перегружены из-за вызовов методов? Это тоже может быть хорошим направлением для последующего изучения кода.

3. Поймите концепцию из набора связанных объектов.

   Теперь у нас отмечено несколько строк, относящихся к точке фокуса. Вы можете узнать много нового благодаря *шаблонам вызова* в выбранном фрагменте кода. Например, есть ли такой метод, который вызывается в нескольких местах кода? Возможно, этот метод играет большую роль в коде и может стать точкой фокуса для дальнейшей работы с кодом. На все остальные методы, которые не используются в коде, можно не обращать внимания. Когда вы редактируете код в интегрируемой среде разработки, вы можете захотеть разместить задействованные в коде методы рядом с точкой фокуса, а незадействованные где-нибудь вне поля вашего зрения. Такое решение поможет вам сэкономить время и не увеличивать когнитивную нагрузку при просмотре кода.

   Мы также можем посмотреть, какие части кода в срезе перегружены вызовами методов. Связанные части кода являются ключевыми концепциями, поэтому они могут стать отличной точкой фокуса при дальнейшей работе с кодом. После того как вы изучите важные места в коде, вы можете создать список всех связанным классов. Создайте список и хорошенько подумайте. Помогают ли определенные вами объекты и связи между ними сформировать представление о концепции, лежащей в основе кода?

4. Поймите концепции из всего множества объектов.

   На последнем этапе вам нужно получить полное представление о различных концепциях кода. Например, вы хотите понимать не только структуры данных, которые содержатся в коде, но также операции и ограничения, применяемые к ним. Что вам можно делать, а что запрещено? Например, является ли дерево бинарным или может ли узел дерева иметь произвольное количество дочерних узлов? Есть ли у дерева ограничения? Например, возникнет ли ошибка, если вы добавите третий узел, или это зависит от пользователя?

На последнем этапе вы можете создать список концепций кода, чтобы задокументировать свое понимание кода. Список связанных объектов из шага 3, и этот список концепций можно добавить в код в качестве документации.

УПРАЖНЕНИЕ 5.2. Найдите фрагмент незнакомого кода в вашей кодовой базе. Вы также можете найти код на GitHub. По правде говоря, не важно, какой код вы используете, но он должен быть незнакомым. Для углубленного понимания кода выполните следующие шаги:

1. Найдите в коде точку фокуса. Так как вы не исправляете ошибку и не добавляете функцию, то для вас точкой фокуса будет начало кода, например метод `main()`.
2. Распечатайте код или работайте в интегрированной среде разработки. Определите срез кода, связанный с точкой фокуса. Возможно, вам потребуется рефакторинг кода, чтобы собрать фрагменты из среза кода вместе.
3. Основываясь на шаге 2, запишите, что вы узнали о коде. Какие объекты и концепции есть в коде и как они связаны друг с другом?

5.4. Чтение кода как обычного текста

Несмотря на то что программистам обычно нужно читать много кода — ранее в книге говорилось, что примерно 60% своего времени программист тратит на чтение кода, а не на его написание[5], — мы, разработчики, не так часто читаем код. В своей книге «Кодеры за работой. Размышления о ремесле программиста» (Символ-Плюс, 2011) Питер Сейбел (Peter Seibel) взял интервью у разработчиков об их рабочих привычках, в том числе и о чтении кода. И хотя большинство опрошенных Сейбелом людей сказали, что чтение кода — дело важное и его нужно практиковать как можно чаще, лишь немногие могли вспомнить последний прочитанный код. Дональд Кнут был ярким исключением.

Так как нам не хватает практики, хороших стратегий и методов, мы часто полагаемся на медленную практику чтения кода — построчное чтение или пошаговое выполнение кода с помощью отладочной программы. Это приводит к тому, что люди

[5] См. «Measuring Program Comprehension: A Large-Scale Field Study with Professionals», Син Ся и др. (2017), **https://ieeexplore.ieee.org/abstract/document/7997917**

предпочитают писать свой собственный код, а не использовать и адаптировать уже существующий код, т. к. «самому создать код проще». А что было бы, если читать код было бы так же легко, как и текст на естественном языке? В оставшейся части главы мы рассмотрим сходство чтения кода и текста на естественном языке, а затем способы и методы чтения на естественном языке, которые можно применить для облегчения чтения кода.

5.4.1. Что происходит в мозге при чтении кода

Исследователи пытались понять, что происходит в мозгу программиста, когда он продолжительное время занимается программированием. Ранее в книге мы видели несколько примеров подобных экспериментов, проведенных в 1980-х годах исследовательницей Кэтрин МакКитен. Ее эксперимент мы рассмотрели в *главе 2*. Она просила участников вспомнить программу на языке Алгол, чтобы понять, как люди запоминают[6].

В ранних экспериментах, связанных с программированием и мозгом, очень часто использовались принятые в то время техники: например, участники запоминали слова или ключевые слова. Хотя эти техники до сих пор используются, исследователи также применяют более совершенные современные техники. К ним относится метод нейровизуализации, который помогает определить области мозга и соответствующие им когнитивные процессы, задействованные при программировании.

Поля Бродмана

Несмотря на то что мы многое не знаем о нашем мозге, у нас есть достаточно представлений о том, какие части мозга отвечают за те или иные когнитивные функции. В основном мы знаем об этом благодаря немецкому неврологу Корбиниану Бродману (Korbinian Brodmann). В 1909 году он опубликовал книгу «*Vergleichende Lokalisationslehre der Großhirnrinde*» («Сравнительное учение о локализации областей в больших полушариях головного мозга»), в которой подробно описал расположение 52 областей мозга, сейчас известных как *цитоархитектонические поля Бродмана*. Для каждого поля Бродман подробно описал ментальные функции, располагающиеся в данном поле, например чтение слов или запоминание. Количество деталей на его карте постоянно увеличивается благодаря множеству исследований, проведенных в последующие годы[7].

Благодаря работе Бродмана и другим исследованиям областей мозга теперь мы представляем то, где в нашем мозге «живут» когнитивные функции. Знание того, какие части мозга отвечают за чтение или рабочую память, помогло нам понять смысл крупных задач.

---

[6] «Knowledge Organization and Skill Differences in Computer Programmers», Кэтрин МакКитен и др. (1981), **http://mng.bz/YA5a**.

[7] Если вам интересно, вы можете посмотреть последнюю версию карты Бродмана на сайте **www.cognitiveatlas.org**.

Эти исследования могут быть выполнены с помощью *функциональной магнитно-резонансной томографии (фМРТ)*. По изменениям кровотока в мозге аппарат фМРТ определяет, какие поля Бродмана активны, а какие — нет. В подобных исследованиях участников обычно просят решить сложную задачу, например головоломку. Измеряя увеличение кровотока в определенных полях Бродмана, мы можем определить, какие когнитивные процессы принимают участие в решении этой задачи, например рабочая память. Однако у аппарата фМРТ есть свои недостатки — например, участникам нельзя двигаться, пока аппарат проводит сканирование мозга. Таким образом, выбор задач, которые могут решать программисты, ограничен и не может включать в себя активности, связанные с созданием кода или написанием заметок.

Показания фМРТ

Появление карты Бродмана (и аппаратов фМРТ) заставило ученых задуматься над программированием. Какие поля мозга и когнитивные функции будут активны? В 2014 году немецкий профессор математики Джанет Зигмунд (Janet Siegmund) впервые провела исследование на тему программирования на аппарате фМРТ[8]. Участников попросили прочитать код на Java, в котором были такие известные алгоритмы, как сортировка, поиск по списку и вычисление степени чисел. Важные для кода имена переменных были заменены на непонятные имена, поэтому участники тратили все усилия на понимание потока выполнения программы, а не на мысли о главной цели кода, основанной на именах переменных.

Исследование Зигмунд показало, что понимание кода активирует пять полей Бродмана и все эти поля расположены в левом полушарии мозга: BA6, BA21, BA40, BA44 и BA47.

Тот факт, что при программировании задействованы поля BA6 и BA40, не является чем-то странным. Эти поля связаны с рабочей памятью (процессором мозга) и вниманием. Однако активация полей BA21, BA44 и BA47 может показаться весьма неожиданной, потому что эти поля связаны с обработкой естественного языка. Результат интересен и потому, что Зигмунд использовала непонятные имена переменных.

Это говорит о том, что, хотя имена переменных и были непонятны, участники читали другие элементы кода (например, ключевые слова) и пытались понять их смысл точно так же, как мы это делаем при чтении слов в тексте на естественном языке.

5.4.2. Если вы можете выучить французский, то сможете выучить и Python

Мы узнали, что сканирование мозга с помощью фМРТ показало, что при работе с кодом участвуют поля мозга, связанные с рабочей памятью и обработкой естест-

---

[8] См. «Understanding Programmers' Brains with fMRI», Джанет Зигмунд и др. (2014), **www.frontiersin.org/10.3389/conf.fninf.2014.18.00040/event_abstract**.

венного языка. Значит ли это, что люди с большим объемом рабочей памяти и хорошим знанием естественного языка будут лучшими программистами?

Благодаря недавним исследованиям стало понятнее, какие когнитивные способности важны для программирования. Доцент Шантель Прат (Chantel Prat) из Вашингтонского университета провела исследование связи когнитивных навыков и программирования. В ходе исследования оценивался уровень участников (всего было 36 студентов, прошедших курс языка Python в Code Academy) в нескольких областях, таких как математика, языки и логика, а также их умение программировать[9]. Тесты, которые Прат использовала для проверки навыков, не связанных с программированием, хорошо известны, широко применяются и показывают достоверный результат. Так, для проверки навыков в области математики она использовала примерно такой вопрос: «5 станков за 5 минут делают 5 деталей. Сколько времени понадобится 100 станкам, чтобы произвести 100 деталей?». Тест на логику напоминал тест на определение уровня IQ: в одном из заданий студентам нужно было закончить последовательность абстрактных изображений.

При проверке навыков программирования исследователи рассмотрели три фактора: баллы за тесты в Code Academy, итоговый проект, в котором студенты должны были написать код игры «Камень, ножницы, бумага», а также результаты теста с несколькими вариантами ответа. Для языка Python эксперты разработали экзамен и систему выставления оценок за итоговый проект.

Так как исследователи имели доступ как к баллам за проверку навыков программирования, так и за проверку навыков в других областях для каждого участника, они смогли создать прогностическую модель, в которой отображалось, какие когнитивные способности указывают на способности к программированию. Результат может удивить. Способность к количественному мышлению — знания и навыки, необходимые для математики, — указывала на способности к программированию всего лишь у 2% участников. Языковые способности дали результат чуть лучше — 17%. Это интересно, т. к. обычно мы говорим о важности математических навыков, а многие мои знакомые программисты жалуются, что им с трудом даются иностранные языки. Лучше всего себя показали рабочая память и логическое мышление — 34% участников.

В данном исследовании были измерены не только когнитивные навыки 36 участников, но и активность мозга с помощью электроэнцефалографии (ЭЭГ). В отличие от фМРТ, это довольно простое устройство, позволяющее измерять активность мозга с помощью электродов, прикрепленных к голове исследуемого. Данные ЭЭГ учитывались при рассмотрении результатов трех задач по программированию.

Участники со способностями к языкам прошли курс Code Academy быстрее других. Скорость обучения и другие навыки программирования взаимосвязаны, так что здесь нельзя сказать, что самые шустрые студенты просто «пробежались» по курсу, не вникая в саму суть. Возможно, значительную роль играет тот факт, что студенты, у которых нет проблем с чтением, усваивают материал намного быстрее. В то

---

[9] «Relating Natural Language Aptitude to Individual Differences in Learning Programming Languages», Шантель Прат и др. (2020), **www.nature.com/articles/s41598-020-60661-8**.

же время студенты, у которых чтение вызывает трудности, усваивают материал медленнее и хуже, вне зависимости от области программирования, которую им нужно изучить.

Для правильности программирования, которая проверялась с помощью задания «Камень, ножницы, бумага», особое значение имели общие когнитивные навыки (к ним относятся рабочая память и логическое мышление). Для декларативных знаний, которые измерялись с помощью теста с несколькими вариантами ответа, большое значение имели показатели ЭЭГ. Как показано на рис. 5.4, согласно результатам исследования то, как хорошо вы выучите язык программирования, зависит от ваших способностей к изучению естественных языков.

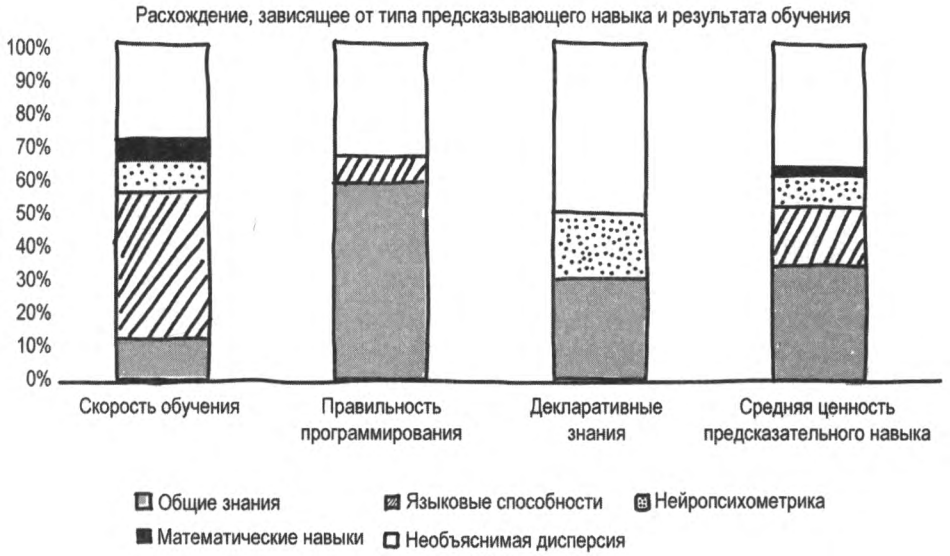


Рис. 5.4. Результаты исследования Прат, показывающие, что математические навыки (на рисунке обозначены черным цветом) являются второстепенным показателем, влияющим на способность к программированию Языковые способности (на рисунке заштрихованы) являются главным показателем, особенно того, как быстро можно выучить язык программирования
Источник Шантель Прат и др. (2020), **www.nature.com/articles/s41598-020-60661-8.pdf**

Для многих программистов результат исследования может показаться неожиданным. Компьютерная наука чаще всего считается как область STEM (НТИМ — наука, технологии, инженерия и математика) и изучается вместе с этими предметами в университете (и в моем тоже). Среди программистов математические навыки считаются не только полезными, но и обязательными. Возможно, результаты исследования подтолкнут вас к пересмотру ваших представлений о том, от чего зависит способность к программированию.

Как люди читают код

Прежде чем мы начнем разбираться с чтением кода, давайте подумаем о том, как мы читаем (научно-популярный) текст, к примеру газету. Что вы делаете, когда читаете статью в газете?

Есть много стратегий, которых придерживаются люди при чтении текста. Например, вы можете пробежаться глазами по тексту перед чтением — так вы поймете, стоит ли вообще тратить время на чтение этого текста. Вы можете рассмотреть изображения, находящиеся в тексте, — так вы немного поймете, о чем текст и что от него можно ожидать. Вы можете делать заметки при чтении, с помощью которых вы потом аннотируете текст, или можете выделять наиболее важные части в тексте. Просмотр текста и изображений — это *стратегии понимания текста*. Многие из этих стратегий преподаются и используются в школе, так что я уверена, что вы используете их, даже не задумываясь над этим.

Скоро мы рассмотрим способы, которые помогут улучшить навыки чтения кода, однако сначала давайте посмотрим, к каким выводам пришли ученые, исследовавшие то, как люди читают код.

УПРАЖНЕНИЕ 5.3. Вспомните, когда вы читали нехудожественный текст последний раз. Какие стратегии вы использовали перед, в момент и после чтения?

Перед тем как читать код, программисты сканируют его

Когда исследователи хотят понять, куда смотрят люди, они используют *устройства отслеживания взгляда*. Это девайс, отслеживающий положение глаз и определяющий, где именно на экране или странице книги человек сосредоточивает внимание. Устройства отслеживания взгляда широко используются в исследованиях маркетинга: они определяют, какая реклама больше всего привлекает внимание людей. Устройства отслеживания взгляда использовались еще в 1920-х годах, и тогда устройство занимало целую комнату. Современные устройства отслеживания взгляда маленького размера. Они могут работать с аппаратурой, например Microsoft Kinect, а также выпускаются в виде программных версий и отслеживают взгляд путем распознавания изображений.

Устройства отслеживания взгляда позволили исследователям лучше понять то, как люди читают код. Например, группа исследователей из Института науки и технологий Нары под руководством профессора Хидетакэ Увано (Hidetake Uwano) обнаружила, что с помощью сканирования программисты узнают, что делает код[10]. Они обнаружили, что за первые 30% потраченного на чтение кода времени программисты просмотрели свыше 70% строк. Быстрое сканирование является обычным явлением при чтении на естественном языке: так человек получает представление о структуре текста. Видимо, люди постепенно начинают применять эту стратегию и при чтении кода.

Начинающие и опытные программисты читают код по-разному

Для того чтобы сравнить, как люди читают код и текст на естественном языке, Тереза Бусьян (Teresa Busjahn), исследователь из Свободного университета Берли-

[10] См. «Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement», Хидетакэ Увано и др. (2006), **www.cs.kent.edu/~jmaletic/cs69995-PC/papers/Uwano06.pdf**.

на, провела исследование, в котором приняли участие 14 начинающих программистов и 6 опытных программистов[11]. Бусьян и ее коллеги изучили разницу между чтением текста на естественном языке и чтением кода. Она обнаружила, что код читается менее последовательно, чем текст на естественном языке: начинающие программисты последовательно читают примерно 80% текста на естественном языке и 75% при чтении кода. Когда программисты не читали последовательно, они просматривали стек вызовов.

Бусьян сравнивала не только чтение кода с чтением текста на естественном языке, но также и опытных программистов с начинающими. Исследование показало, что начинающие и опытные программисты читают по-разному: новички читают более последовательно и смотрят на стек вызовов чаще, чем опытные программисты. Понятно, что только с практикой можно научиться отслеживать стек вызовов при чтении кода.

5.5. Стратегии понимания текста, которые можно применить к коду

Как уже было показано в предыдущем разделе, когнитивные навыки, которые используются при чтении кода, похожи на навыки, которые используются при чтении текста на естественном языке. Это означает, что мы можем применить при чтении кода стратегии, которые используются при чтении текста на естественном языке.

Было проведено множество исследований, где разбирались разные стратегии чтения и способы их освоения. Все стратегии понимания прочитанного материала можно разделить на семь видов[12]:

1. *Активация* — активное размышление о связанных вещах для активации пассивных знаний.
2. *Наблюдение* — отслеживание вашего понимания текста.
3. *Определение важности* — выделение наиболее важных частей текста.
4. *Постановка предположений* — работа с фактами, которые упомянуты в тексте вскользь.
5. *Визуализация* — создание дополнительных схем или рисунков для углубленного понимания текста.
6. *Постановка вопросов* — подготовка вопросов о тексте.
7. *Резюмирование* — краткий пересказ текста.

Так как существует некоторая схожесть между чтением кода и чтением текста, то вполне возможно, что стратегии для чтения на естественном языке будут полезны

---

[11] См. «Eye Movements in Code Reading: Relaxing the Linear Order», Тереза Бусьян и др. (2015), **https://ieeexplore.ieee.org/document/7181454**.

[12] См. «The Seven Habits of Highly Effective Readers», Кэти Энн Миллс (2008), **https://www.researchgate.net/publication/27474121_The_Seven_Habits_of_Highly_Effective_Readers**.

при чтении кода. В данном разделе мы рассмотрит каждую из семи стратегий чтения текста, адаптировав их так, чтобы они подходили для чтения кода.

5.5.1. Активация пассивных знаний

Мы знаем, что перед тем, как читать код, программисты его сканируют. Но чем сканирование кода может быть полезным? Одним из преимуществ является то, что вы получаете представление об используемых концепциях и о синтаксисе кода.

В предыдущих главах мы узнали, что как только вы начнете думать о чем-то, рабочая память будет искать связанные с этим воспоминания в долговременной памяти. Если вы будете активно думать о коде, то это поможет вашей рабочей памяти найти нужную информацию в долговременной памяти. Вы можете выделить около 10 минут на изучение кода — обычно этого хватает для активации уже имеющихся знаний.

УПРАЖНЕНИЕ 5.4. В течение определенного времени (5–10 минут в зависимости от длины кода) просмотрите код. Затем попробуйте ответить на следующие вопросы:

- Какой первый элемент (переменная, класс, концепция) привлек ваше внимание?
- Почему?
- На что вы обратили внимание дальше?
- Почему?
- Связаны ли эти два элемента (переменные, классы, концепции) между собой?
- Какие концепции есть в коде? Знаете ли вы их все?
- Какие синтаксические элементы есть в коде? Знаете ли вы их все?
- Какие понятия предметной области есть в коде? Знаете ли вы их все?

Это задание может подтолкнуть вас к изучению дополнительной информации о неизвестных концепциях программирования или понятий предметной области в коде. Когда вы сталкиваетесь с незнакомым понятием, лучше всего будет попытаться изучить его и только потом снова вернуться к работе с кодом. Изучение нового понятия или новой концепции может вызвать дополнительную когнитивную нагрузку, из-за чего работа с кодом станет менее эффективной.

5.5.2. Наблюдение

При чтении кода важно отмечать то, что вы читаете, и понимаете ли вы прочитанное. Выписывайте не только тот материал, который вы понимаете, но и тот, что непонятен. Для большей пользы вы можете распечатать код на листе и выделять строки, которые вы понимаете и не понимаете. Вы можете делать это с помощью значков, которые вы использовали для обозначения ролей переменных.

На рис. 5.5 показан фрагмент кода на JavaScript, который я комментировала с помощью этого способа. Понятные мне строки я помечала галочкой, а непонятные — знаком вопроса. Подобное наблюдение вашего понимания сильно поможет вам при повторном прочтении кода, потому что тогда вы сможете сосредоточиться на непонятных для вас строках кода.

Комментарии непонятных вам строк могут быть полезны для отслеживания вашего понимания, однако они также помогут найти помощь. Если нужные вам строки уже будут отмечены, то вы сможете попросить автора кода объяснить их. Это будет продуктивнее того, если бы вы сказали что-то вроде: «Я не знаю, что они делают».

```
✓import { handlerCheckTodo } from '../handlers/checkedTask.js';
✓import { handlerDeleteTodo } from '../handlers/deletetask.js';
✓import { restFulMethods } from '../restful/restful.js';

✓export class app {
  ✓state = [];
  ✓nexId = 0;

  ✓renderTodos(todosArray) {
    ✓const Tbody = document.createElement('tbody');

    ✓for (const todo of todosArray) {
      ? const trEl = document.createElement('tr');
      ✓trEl.className = 'today-row';
      ? const DivEl = document.createElement('div');
      ✓DivEl.className = 'row';
      ✓const divElSecond = document.createElement('div');
      ✓divElSecond.className = 'col-1';

      ✓const TdEl = document.createElement('td');
      ✓const checkBoxEl = document.createElement('input');
      ✓checkBoxEl.type = 'checkbox';
      ✓checkBoxEl.addEventListener('click', handlerCheckTodo);

      ✓checkBoxEl.dataset.index = todo.id;
```

Рис. 5.5. Фрагмент кода на JavaScript с пометками Понятные строки отмечены галочкой, а непонятные — знаком вопроса

5.5.3. Определение важности разных строк кода

При чтении кода иногда полезно задуматься над тем, какие строки особенно важны. Вы можете делать это как упражнение. Не имеет значения, сколько строк вы посчитаете важными — это может быть как 10 строк в коротком коде, так и 25 строк в большом коде. Имеет значение лишь то, как вы определяете важность строки.

Если вы работаете с напечатанным кодом, то можете отмечать важные строки восклицательным знаком.

УПРАЖНЕНИЕ 5.5. Выберите фрагмент незнакомого кода. Несколько минут просматривайте код и выделите важные строки. Ответьте на следующие вопросы:

- Почему вы выбрали эти строки?
- Какую роль играют эти строки? Например, это строки, выполняющие инициализацию, ввод-вывод или обработку данных?
- Как эти строки связаны с общей целью, которую должен выполнить код?

Что такое важная строка? У вас может возникнуть вопрос: что такое важная строка? Хороший вопрос! Я часто делала упражнения на выделение важных строк кода с командой разработчиков: в этом упражнении каждый член команды отдельно отмечал строки, которые он считал самыми важными, а затем вы все сравнивали результаты.

Люди, работающие в команде, очень часто расходятся во мнении насчет того, какие строки важны. Некоторые утверждают, что важная строка — это строка, где происходят основные вычисления. Другие в качестве важной строки выбирают оператор импорта для определенной библиотеки или пояснительный комментарий. Люди с опытом работы на разных языках программирования или в разных областях могут иметь разное мнение насчет важности строк кода, и это совершенно нормально. Не думайте об этом как о проблемах, которые нужно решить, а как о возможности узнать что-то новое.

Выполнение этого упражнения вместе с командой хорошо еще и тем, что оно не только помогает понять код, но и позволяет вам узнать как о себе, так и о ваших коллегах (опыт, приоритетные задачи и т. д.).

5.5.4. Предположения о значении имен переменных

Большая часть всего смысла кода лежит в его структуре — например, в операторах условия или в использовании циклов. Смысл есть также в именах таких элементов программы, как переменные, но иногда их смысл придется додумывать. Если имя переменной `shipment`, то полезно будет узнать, что это имя означает в предметной области кода. Транспортировка товара — это то же самое, что и заказ, группа товаров, предназначенная для одного покупателя? Или транспортировка товара — это группа товаров, которые необходимо отправить на производство?

Как вы уже видели, имена переменных могут служить важными *маячками*: с помощью их вы можете понять, о чем в коде идет речь. При чтении кода полезно обращать внимание на имена переменных.

Упражнение заключается в том, что вам надо построчно просмотреть код и создать список всех идентификаторов (классов, переменных, методов, операторов). Вы можете сделать это даже без понимания кода. Конечно, такая работа с кодом может показаться странной, однако чтение идентификаторов может улучшить вашу рабочую память. Как только вы начнете работу с именами переменных, в своей долговременной памяти вы сможете найти подходящую информацию. Найденная информация помогает рабочей памяти с легкостью обрабатывать код.

Создав список идентификаторов, вы можете использовать его для углубленного понимания кода. К примеру, все имена переменных вы можете разделить на две группы: имена переменных, связанных с областью кода (например, `Customer` и `Package`), и имена переменных, связанных с концепциями программирования (например, `Tree` и `List`). Некоторые имена переменных будут входить в обе категории (например, `CustomerList` и `FactorySet`). Не все имена переменных будут понятны без контекста, поэтому вам придется потратить больше времени, чтобы понять их значение — например, вы можете попытаться понять, какую роль играет переменная с помощью структуры Сайаниеми, рассмотренной в начале этой главы.

УПРАЖНЕНИЕ 5.6. Выберите фрагмент исходного кода и создайте список имен всех переменных из кода.

Заполните следующую таблицу для всех переменных:

Имя	Область?	Концепция?	Понятно ли имя переменной без контекста?

Используя эту таблицу, ответьте на следующие вопросы:

- ❒ Какая перед нами область или тематический раздел кода?
- ❒ Какие концепции программирования используются в коде?
- ❒ Что можно узнать из имен переменных?
- ❒ Какие имена переменных связаны друг с другом?
- ❒ Есть ли имена, значение которых без контекста понять нельзя?
- ❒ Какое значение могут иметь такие переменные в этом исходном коде?

5.5.5. Визуализация

В предыдущих главах мы сталкивались с несколькими стратегиями визуализации кода для углубленного понимания: например, создание таблицы состояний и отслеживание потока выполнения кода.

Существуют другие стратегии визуализации, которые можно использовать для углубленного понимания кода. Одной из таких стратегий, которая может помочь при работе со сложным кодом, является перечисление всех операций, в которых участвуют переменные.

Таблица операций

При работе с незнакомым кодом иногда трудно сказать, изменятся ли значения переменных или нет. Когда код тяжело обработать, вы можете создать таблицу операций. Например, следующий фрагмент кода на JavaScript (листинг 5.1) может

быть труден для понимания, особенно если вы не знаете, что оператор `zip` объединяет два списка.

Листинг 5.1. Код на JavaScript, который объединяет списки `as` и `bs`, используя функцию `f`

```
zipWith: function (f, as, bs) {
  var length = Math.min(as.length, bs.length);
  var zs = [];
  for (var i = 0; i < length; i++) {
    zs[i] = f(as[i], bs[i]);
  }
  return zs;
}
```

В этом случае вы можете проверить переменные, методы и операторы и определить, в каких операциях они используются. Например, `f` применяется к `as[i]` и `bs[i]`, так что это функция. При проверке `as` и `bs` мы видим, что они имеют индекс — следовательно, они являются списками или словарями. После того как вы определите типы переменных в сложном коде с помощью операций, вы легко сможете определить их роли.

УПРАЖНЕНИЕ 5.7. Выберите фрагмент незнакомого кода и выпишите имена всех переменных, методов и классов. После перечислите все операции, связанные с каждым из идентификаторов.

Имя идентификатора	Операция(и)

После того как вы заполните таблицу, прочитайте код еще раз. Помогла ли работа с таблицей на углубленном уровне понять роли переменных и смысл всей задачи?

5.5.6. Постановка вопросов

Понять цель и смысл кода могут помочь вопросы, которые вы задаете себе по мере прочтения кода. В предыдущих разделах мы видели множество примеров вопросов, которые можно задать себе. Вот еще несколько важных вопросов:

- Каковы пять ключевых концепций кода? Это может быть имя идентификатора, темы, классы или информация, найденная в комментариях.
- Какие стратегии вы использовали для их определения? Например, вы учли имена методов, имена переменных, документацию или опирались на свой опыт работы с системой?

- ❒ Какие пять ключевых концепций компьютерных наук были использованы в коде? Это могут быть алгоритмы, структуры данных, допущения или стратегии.
- ❒ Как вы оцениваете решения, принятые создателем(ями) кода, например решение использовать определенную версию алгоритма, определенный паттерн проектирования, определенную библиотеку или интерфейс?
- ❒ На каких допущениях основаны ваши решения?
- ❒ Какие преимущества есть у ваших решений?
- ❒ Какие возможные недостатки есть у этих решений?
- ❒ Есть ли у вас предложения альтернативных решений?

Эти вопросы затрагивают не только структуру текста, но и могут помочь вам составить план понимания кода.

5.5.7. Резюмирование кода

Последняя стратегия понимания текста, которую можно использовать в работе с кодом, — резюмирование прочитанного материала. Если вы на естественном языке кратко опишете код, то это поможет вам углубленно понять сам код. Резюмирование можно использовать как дополнительную документацию или даже как основную документацию к коду, если до этого момента ее не было.

Некоторые из рассмотренных ранее стратегий могут помочь сделать резюмирование кода. Например, для начала отлично подойдет выделение важных строк, перечисление всех переменных и связанных с ними операций, а также размышление над решениями, которые приняли авторы кода.

УПРАЖНЕНИЕ 5.8. Резюмируйте фрагмент кода, заполнив следующую таблицу. Естественно, вы можете расширить таблицу и добавить больше информации.

Вопросы	
Цель кода. Чего хочет достичь код?	
Самые важные строки кода	
Основные понятия предметной области	
Основные конструкции программирования	
Решения, принятые при написании кода	

Выводы

- При чтении кода обращайте внимание на то, какую роль играют переменные, — это намного облегчит понимание кода.
- При понимании кода существует большая разница между *пониманием текста*, означающим знание синтаксических концепций, и *пониманием плана*, означающим понимание цели автора кода.
- Между чтением кода и чтением на естественном языке есть много общего, поэтому ваше умение учить естественные языки может помочь вам усовершенствовать навык программирования.
- Стратегии, использующиеся для углубленного понимания текстов на естественном языке, например визуализация и резюмирование, могут использоваться для углубленного понимания кода.

6 Совершенствуем навыки решения задач программирования

В этой главе:

- вы узнаете о применении моделей для эффективного рассуждения о задачах программирования;
- узнаете, как разные способы размышления о задачах влияют на то, как вы их решаете;
- научитесь использовать модели для анализа кода и эффективного решения задач;
- рассмотрите методы, позволяющие решать задачи путем улучшения долговременной памяти;
- попрактикуетесь в использовании моделей для решения задач с помощью рабочей памяти;
- научитесь очерчивать задачи, абстрагируясь от ненужных деталей.

В предыдущих главах мы узнали о разных когнитивных процессах, происходящих в мозге при программировании. Мы узнали, как информация хранится в кратковременной памяти при чтении кода, а также о том, как информация извлекается из нашей долговременной памяти. Мы также немного узнали о рабочей памяти, которая работает с того момента, как мы начинаем думать о коде. В *главе 5* мы рассмотрели стратегии, которые помогают понимать незнакомый код.

В этой главе все внимание мы уделим тому, как решать задачи. Будучи профессиональным программистом, вы часто будете взвешивать различные решения задач. Составить перечень всех клиентов компании в виде простого списка или в виде дерева, отсортированного по филиалам? Использовать структуру на основе микросервисов, или вся логика должна находиться в одном месте?

Когда вы рассматриваете разные варианты решения задач, вы понимаете, что разные варианты решения имеют свою ценность. Принять нужное решение иногда

бывает трудно, т. к. вам нужно учитывать множество факторов. Например, чему вы будете уделять больше внимания: простоте использования или производительности? Собираетесь ли вы изменять код в будущем, или для вас главное — выполнить текущую задачу?

В этой главе мы рассмотрим две схемы, которые помогут вам принимать решения о проектировании программного обеспечения. Для начала мы рассмотрим мысленные представления, которые создает наш мозг при решении задач и программировании. Если вы будете знать, какие представления вы используете тогда, когда вы размышляете о коде, то это поможет вам решать разные виды задач, рассуждать о коде и решать задачи намного быстрее и эффективнее. В этой главе мы рассмотрим два способа с использованием моделей, которые помогут вам укрепить долговременную и рабочую память.

Мы также узнаем, как мы думаем о компьютерах во время решения задач. Занимаясь программированием, мы не всегда учитываем особенности машин, на которых работаем. Бывают моменты, когда мы не замечаем многих деталей: например, вы создаете интерфейс пользователя, и для вас большая часть особенностей операционной системы не важна. Однако при создании приложения для телефона или модели машинного обучения вам необходимо знать особенности аппаратной платформы, на которой будет выполняться код. Вторая структура, которую мы рассмотрим в этой главе, позволяет размышлять о задачах на нужном уровне абстракции.

6.1. Использование моделей для размышлений о коде

Когда люди решают задачи, они почти всегда создают модели. Модель — это упрощенное представление реальности. Основная цель модели заключается в том, чтобы помочь вам в обдумывании и решении задачи. Модели могут быть разной формы и разных уровней формализации. Моделью может считаться как приближенное вычисление на салфетке, так и диаграмма взаимосвязей элементов системы ПО.

6.1.1. Преимущества использования моделей

В предыдущих главах мы создавали разные типы моделей, которые помогают размышлять о коде. Например, мы создавали диаграмму состояний для записи значения переменных (рис. 6.1). Мы также создавали граф зависимостей, который является еще одной разновидностью модели кода.

У явных моделей кода есть два преимущества. Во-первых, такие модели помогают передать информацию о программах другим. Создав диаграмму состояний, я могу показать другому человеку все промежуточные значения переменных и, таким образом, помогу ему понять, как работает код. Это особенно полезно в случае больших систем. Например, если мы вместе посмотрим на структурную диаграмму

```
1  LET N2 =  ABS (INT (N))
2  LET B$ = ""
3  FOR N1 = N2 TO 0 STEP  0
4      LET N2 =  INT (N1 / 2)
5      LET B$ =  STR$ (N1 - N2 * 2) + B$
6      LET N1 = N2
7  NEXT N1
8  PRINT B$
9  RETURN
```

	N	N2	B$	N1
Init	7	7	–	7
Loop1		3	1	3
Loop2				

Рис. 6.1. Код на BASIC, преобразующий число N в двоичное представление
Вы можете воспользоваться вспомогательными средствами памяти, например таблицей промежуточных состояний, чтобы понять, как работает код

кода, то я смогу вам показать все классы и отношения между ними, а также объекты, которые в любой другой ситуации были бы скрыты в коде.

Вторым преимуществом моделей является то, что они могут помочь вам решить задачу. Когда вы близки к предельному количеству элементов, которые можно обработать за один раз, создание модели позволяет снизить когнитивную нагрузку. Как ребенок, складывающий 3 и 5 с помощью числовой прямой (своего рода модель) вместо складывания чисел в уме, программисты могут изобразить архитектуру системы на доске, т. к. очень сложно удерживать все элементы большой кодовой базы в рабочей памяти.

Модели могут быть очень полезны при решении задач, т. к. они помогают долговременной памяти распознать нужные воспоминания. Очень часто модели имеют ограничения: например, диаграмма состояний показывает только значения переменных, а диаграмма взаимосвязей элементов показывает только классы и отношения между ними. Ограничения моделей заставляют вас фокусировать внимание лишь на определенной части задачи, что может склонить вас к выбору неполного решения. Думая о числовой прямой при сложении, вы сосредоточиваете все внимание на счете, а диаграмма взаимосвязей элементов фокусирует ваше внимание на том, из каких элементов или классов состоит ваш код и как они связаны между собой.

Не все модели одинаково полезны

Модели, которые мы используем для размышления о задачах, разные. Как программисты, мы понимаем всю важность представления и его влияние на решение задач. Например, деление числа на два является простой задачей, когда мы преобразовали число в двоичное представление: нам просто надо сдвинуть биты вправо на единицу. И хотя этот пример достаточно простой, существует множество ситуаций, в которых от представления зависит стратегия решения задачи.

Важность представления можно показать на примере задачи с птицей и двумя поездами. Птица сидит на поезде, который отправляется из Кембриджа в Лондон. Как только поезд трогается, из Лондона отходит второй поезд, находящийся в 50 милях. Птица взлетает и летит по направлению ко второму поезду со скоростью 75 миль/ч. Оба поезда едут с одинаковой скоростью в 50 миль/ч. Когда птица долетает до второго поезда, она разворачивается и летит к первому поезду. Она делает это до тех

пор, пока два поезда не встретятся. Как далеко будет птица в момент встречи поездов?

Сначала люди представляет два поезда и летающую между ними птицу (рис. 6.2).

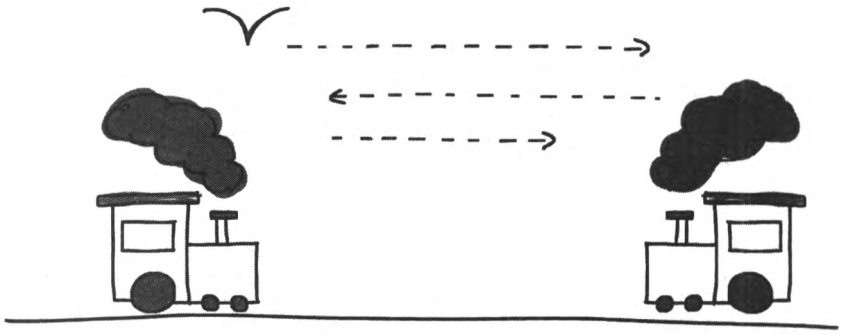


Рис. 6.2. Модель расстояния, которое пролетает птица между двумя поездами с точки зрения птицы Это правильное, но весьма сложное решение, в котором нужно рассчитывать положение обоих поездов

Моделирование траектории полета птицы — это правильное решение, однако оно включает в себя сложную систему уравнений, которые множество людей не хотели бы решать. В простом решении все внимание уделено птице, а решение выглядит следующим образом. Поезда встретятся через 30 мин между Кембриджем и Лондоном. К этому времени оба поезда прошли расстояние в 25 миль. Так как птица летит со скоростью 75 миль/ч, то за полчаса она пролетит 37,5 миль. Этот пример хорошо иллюстрирует ситуацию, что то, как вы думаете о задаче, влияет на то, как вы ее решаете.

В программировании мы также сталкиваемся с разными представлениями задач. Некоторые языки программирования лимитируют количество возможных представлений, что может быть как полезным, так и вредным для решения задач. Например, язык APL идеально подходит для создания модели решения, включающего в себя матрицы. Однако на этом языке бывает трудно решить задачи, требующие другого представления. С другой стороны, на языке Java вы можете создавать классы для представления любых задач, так что этот язык может использоваться для решения задач с матрицами, однако вам придется создавать класс матрицы. Скорее всего, в данном случае вы выберете решение, включающее в себя создание двух вложенных циклов `for`, т. к. они встроены в язык и широко применяются.

6.2. Ментальные модели

До сих пор мы рассматривали модели, которые были созданы вне нашего мозга. Таблицы состояний, графы зависимостей или диаграммы взаимосвязей элементов являются моделями, которые строятся на бумаге или любой другой физической поверхности. Вы можете создать такую модель для того, чтобы показать ее другим людям или углубленно понять задачу. Однако существуют также модели, которые создаются в уме. Такие модели называются *ментальными моделями*.

В предыдущем разделе мы увидели, что представление, которое вы используете для решения задачи, влияет на то, как вы размышляете о задаче. Это относится и к ментальным моделям: одни помогают обдумывать задачу, а другие — нет. В этом разделе мы узнаем, что такое ментальные модели и как их можно использовать для решения задач.

Примером ментальной модели, которую мы используем для работы с кодом, является то, как мы думаем об обходе дерева. Конечно, ни в коде, ни в компьютере нет настоящего дерева, которое мы обходим; просто в памяти есть информация, о которой мы думаем как о древовидной структуре. Эта модель помогает нам рассуждать о коде. Намного легче думать о «дочерних узлах», чем об «элементах, на которые ссылается этот элемент».

Впервые ментальные модели были описаны шотландским философом Кеннетом Крейком (Kenneth Craik) в работе «*Природа объяснения*» в 1943 году. Крейк описал ментальные модели как «модели действительности в уменьшенном масштабе». Согласно Крейку, люди используют ментальные модели для изучения и объяснения окружающей действительности, а также предвидения будущих событий.

Лично мне больше всего нравится такое определение ментальной модели: ментальная модель создает в вашей рабочей памяти абстракцию, с помощью которой вы можете рассуждать о задаче.

Работая с компьютером, мы создаем большое количество ментальных моделей. Например, при мысли о файловой системе вы представляете папку с файлами. Если вы подумаете чуть больше, то поймете, что на вашем жестком диске нет файлов и папок; жесткий диск содержит только нули и единицы. Однако, когда мы думаем о нулях и единицах, мы представляем их организованными в такую структуру.

Когда мы размышляем о коде, мы также используем ментальные модели. Примером ментальной модели, которую мы создаем при размышлении о программировании, является идея о выполнении каждой конкретной строки кода. Мы думаем так даже при работе с компилируемым языком, хотя, конечно же, выполняется не строка Java или C, а сгенерированный байт-код, соответствующий этой строке. Несмотря на то, что наше представление не соответствует реальному механизму выполнения программы, оно может стать отличной моделью для рассуждения о программах.

Но модели могут также и навредить: например, когда вы работаете с высокооптимизированным кодом в отладчике и понимаете, что оптимизирующий компилятор изменил код настолько, что отладчик двигается по коду не тем образом, каким вы представляли выполнение исходного кода.

УПРАЖНЕНИЕ 6.1. Рассмотрим фрагмент кода, с которым вы работали в последние дни. Какие ментальные модели вы использовали при программировании? Эти модели связаны с компьютером, выполнением кода или другими аспектами программирования?

6.2.1. Подробное исследование ментальных моделей

У ментальных моделей и моделей, которые выражаются за пределами нашего мозга, есть одна похожая характеристика: модель правильно представляет задачу, однако в более простом и абстрактном варианте. Ментальные модели обладают и другими важными характеристиками, перечисленными в табл. 6.1.

Таблица 6.1. *Основные характеристики ментальных моделей с примерами из программирования*

Характеристика	Пример
Ментальные модели неполные. Ментальная модель не должна быть полной моделью целевой системы, она похожа на масштабную модель, которая в некоторой степени упрощает моделируемый физический объект. Неполная ментальная модель может быть полезна, если она не включает в себя ненужные детали	Не стоит представлять переменную как коробочку со значением, т к такая модель не помогает вам думать о переназначении Сможет ли второе значение поместиться с первым в одной коробочке? Или оно вытеснит первое значение?
Ментальные модели непостоянны Не все модели навсегда остаются без изменений, модели очень часто изменяются Например, вы создаете ментальную модель электричества, сравнивая его с потоком воды Сначала вы представляете прямую реку, но когда вы узнаете, как течет ток, то ваша река становится то широкой, то узкой Человек может забыть некоторые части ментальной модели, если он долгое время не будет ею пользоваться	Когда мы учимся программировать, то мы думаем о переменной как о коробочке со значением внутри Однако спустя время мы понимаем, что переменная не может иметь больше одного значения, так что потом лучшей аналогией становится метка с именем
Одновременно могут существовать несколько ментальных моделей, даже если они конфликтуют друг с другом У начинающих очень часто есть «локально связанные, но глобально несовместимые» ментальные модели, которые привязаны к деталям конкретного случая, который они рассматривают[1]	Вы можете представлять переменную как коробочку со значением Вы также можете представлять переменную как метку с именем, которую вы прикрепляете к значению Обе ментальные модели могут существовать в одно и то же время и иметь свои преимущества в определенных ситуациях
Ментальные модели могут выглядеть «сомнительно» и даже казаться суевериями Очень часто люди верят в бессмысленные вещи	Вы когда-нибудь говорили компьютеру что-то вроде «Пожалуйста, заработай в этот раз»? Даже зная, что компьютер не является разумным существом и не слышит вас, вы все равно создаете ментальную модель компьютера в виде живого существа, которое может решить оказать вам услугу
Люди воздержанно относятся к использованию ментальных моделей Так как мозг потребляет много энергии, люди стараются экономить умственные ресурсы и решать проблемы физически	При отладке многие программисты вносят небольшие изменения в код и запускают его снова, проверяя, исправлена ли ошибка, а не расходуют энергию для того, чтобы создать хорошую ментальную модель задачи

[1] Гентнер, Дерде (2002). Психология ментальных моделей. См. «International Encyclopedia of the Social and Behavioral Sciences», Н. Дж. Смелсер и П. Б. Бейтс (стр. 9683–9687).

6.2.2. Изучение новых ментальных моделей

Как показано в табл. 6.1, люди могут одновременно работать с двумя конфликтующими ментальными моделями. Вы можете думать о файле, находящемся «в папке», в то же время понимая, что файл — это ссылка на место на жестком диске, где хранится информация.

При изучении программирования люди часто учат новые ментальные модели постепенно. Например, сначала вы думаете о документе на жестком диске как о настоящем, физическом листе бумаги с написанными на нем словами, хранящемся в каком-нибудь месте; только потом вы узнаете, что на жестком диске хранятся только нули и единицы. Другой пример: сначала вы думаете о переменной и ее значении как об имени и номере телефона в адресной книге, но затем вы обновляете свою модель, т. к. узнаете, как устроена рабочая память компьютера. Вы можете подумать, что как только вы узнаете, как на самом деле что-то работает, то «неправильная» модель стирается из вашей памяти и заменяется «правильной» моделью. Однако из предыдущих глав мы узнали, что из долговременной памяти информация почти никогда не удаляется полностью. А это означает, что всегда есть вероятность того, что вы вновь воспользуетесь устаревшими и неверными ментальными моделями, заученными ранее. Бывают ситуации, когда несколько ментальных моделей могут быть одновременно активны, и границы между ними не всегда четкие. В результате этого из-за высокой когнитивной нагрузки вы внезапно можете прибегнуть к использованию старой ментальной модели.

В качестве примера конфликтующих ментальных моделей давайте рассмотрим загадку: что произойдет со снеговиком, если на него надеть теплый свитер? Скорость таяния снеговика увеличится или останется прежней?

Первой вашей мыслью может быть то, что скорость таяния снеговика увеличится, т. к. ваш мозг получает ментальную модель свитера, который дает тепло. Однако после некоторых рассуждений вы понимаете, что свитер не дает, а удерживает тепло нашего тела. Так как свитер удерживает что-то, то он будет удерживать холод, который снеговик теряет, — следовательно, скорость таяния снеговика не увеличится, а уменьшится.

Аналогично при чтении сложного кода вы можете неожиданно прибегнуть к простым ментальным моделям. Например, при чтении кода с большим количеством указателей вы можете запутаться в значениях и адресах памяти, путая между собой ментальные модели переменных и указателей. Или при отладке сложного кода с асинхронными вызовами вы можете использовать старую, неполную ментальную модель синхронного кода.

УПРАЖНЕНИЕ 6.2. Подумайте о двух известных вам ментальных моделях для одной концепции программирования, например для переменных, циклов, файлового хранилища или управления распределением памяти. Какие у этих двух моделей есть сходства и различия?

6.2.3. Как эффективно использовать ментальные модели во время размышлений о коде

В предыдущих главах мы рассматривали происходящие в мозге когнитивные процессы. Мы говорили о долговременной памяти, в которой хранятся не только важные воспоминания, но и абстрактные представления знаний — схемы. Мы также говорили о рабочей памяти, в которой происходит процесс мышления.

У вас может возникнуть вопрос: с какими когнитивными процессами связаны ментальные модели? Эти модели хранятся в долговременной памяти и при необходимости передаются рабочей памяти? Или они формируются в рабочей памяти при работе над кодом? Понимание того, как обрабатываются ментальные модели, очень важно, т. к. это помогает нам эффективно использовать эти модели. Если они хранятся в долговременной памяти, то мы можем выучить их с помощью дидактических карточек, а если они формируются в рабочей памяти, то мы можем использовать визуализацию для поддержки когнитивного процесса, отвечающего за использование ментальных моделей.

Интересный факт: ментальные модели рассматривались в первой книге Кеннета Крейка, а затем эта тема не поднималась около 40 лет. В 1983 году были опубликованы две книги под одинаковым названием «*Mental Models*», но разных авторов. Авторы книг придерживались разных точек зрения насчет того, как наш мозг обрабатывает ментальные модели. Мы рассмотрим эти точки зрения в последующих разделах.

Ментальные модели в рабочей памяти

Первая книга о ментальных моделях вышла в 1983 году под авторством Филиппа Джонсона-Лэрда (Philip Johnson-Laird), профессора психологии из Принстонского университета. Джонсон-Лэрд утверждал, что ментальные модели активируются во время мышления и располагаются в рабочей памяти. В своей книге он описывает исследование, в котором он и его коллеги изучали эти модели. Участникам давались наборы предложений, в которых была описана сервировка стола, например «Ложка лежит справа от вилки» и «Тарелка стоит справа от ножа». Затем участников попросили выполнить несколько несвязанных друг с другом задач. Затем им представили четыре варианта описания сервировки стола и попросили определить, какой из вариантов больше всего подходит под описание, которое им дали в самом начале.

Предоставленные участникам варианты были такими: два варианта были полностью неверными, один — верным, а также было предоставлено описание, из которого можно было понять расположение предметов. Например, из описаний «Нож лежит слева от вилки» и «Вилка лежит слева от тарелки» можно понять, что тарелка стоит справа от ножа. Затем участников попросили расставить варианты в порядке от самого подходящего под описание до самого неподходящего.

Как правило, на первое место участники ставили верный вариант и вариант с описанием, из чего исследователи сделали вывод, что участники создали ментальную

модель сервировки стола, которую они использовали во время выбора правильного варианта ответа.

Что полезное о ментальных моделях и программировании мы можем узнать из книги Джонсона-Лэрда? Он писал, что очень полезно создавать абстрактную модель, т. к. она позволяет рассматривать саму модель, а не полагаться на доработку кода, что было бы менее эффективно при работе с кодом.

Точные модели работают лучше

Далее мы рассмотрим, как можно сознательно создавать ментальные модели при рассуждении о коде. Однако сначала давайте рассмотрим один из аспектов исследования Джонсона-Лэрда. В его исследовании был один интересный поворот!

Участники получили разные виды описаний. В некоторых случаях описания, которые им давали, соответствовали только одной сервировке стола. Иногда даваемые описания могли соответствовать нескольким реальным сервировкам. Например, утверждения «Вилка лежит слева от ложки» и «Ложка лежит справа от вилки» подходят для обеих сервировок (рис. 6.3). С другой стороны, утверждение «Тарелка стоит между ложкой и вилкой» походит только для сервировки, расположенной в левой половине рисунка.

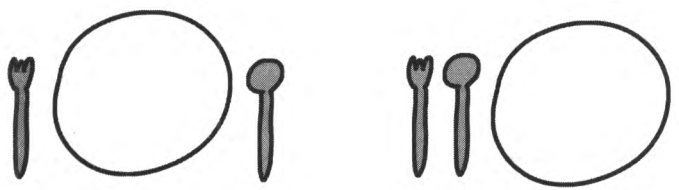


Рис. 6.3. Примеры рисунков, которые участники должны были сопоставить с полученным описанием. Здесь утверждение «Вилка лежит слева от ложки» правдиво для обоих вариантов

Когда Джонсон-Лэрд сравнил результаты участников по обоим видам описания, то он обнаружил, что участники с определенными описаниями выбирали правильные ответы чаще, чем те участники, которым давали описания, соответствующие нескольким вариантам сервировки стола. Разница была большой: первая группа дала 88% правильных ответов против второй группы, где было дано 58% правильных ответов.

Применительно к программированию это означает, что чем больше деталей в ментальной модели, тем легче рассуждать о коде и правильно отвечать на вопросы о коде.

Создание ментальных моделей исходной программы в рабочей памяти

Ранее в этой главе мы узнали, что точные и детальные ментальные модели могут поддерживать наши размышления о сложных кодах или системах. И у вас может возникнуть вопрос: как создать такую ментальную модель? Когда код простой, то у вас не возникнет трудностей при создании ментальной модели. Однако если

структура кода сложная или у вас недостаточно знаний о кодовой базе или предметной области, то на создание ментальной модели придется потратить много усилий. Но на создание модели стоит потратить время и усилия, т. к. она может быть очень полезной.

Выполните эти три шага, чтобы сформировать ментальную модель сложного кода у вас в рабочей памяти:

1. Начните с создания локальных моделей. В предыдущих главах вы узнали об использовании моделей для поддержки рабочей памяти: таблицы состояний и графа зависимостей. И хотя эти способы поддержания памяти являются локальными и отвечают лишь за фрагмент кода, они могут помочь вам создать ментальную модель большого фрагмента кода двумя способами. Во-первых, локальные модели поддерживают рабочую память и снижают когнитивную нагрузку, так что вы можете полностью сосредоточиться на создании большой ментальной модели. Во-вторых, локальные модели могут стать «кирпичиками» для большой ментальной модели. Например, на графе зависимостей отображаются тесно связанные между собой строки кода, которые могут играть важную роль при создании ментальной модели.
2. Составьте список всех элементов, используемых в кодовой базе, и отношений между ними. При создании ментальной модели вы можете порассуждать об элементах, которые будут в этой модели. Например, ментальная модель кода, создающего накладную, может содержать в себе ограничение, согласно которому у одного человека может быть несколько накладных, но при этом накладная принадлежит только одному человеку. Для того чтобы понять связь между разными элементами кода, составьте список элементов и отметьте связи между ними. Это поможет вам составить четкое представление обо всей системе.
3. Ответьте на вопросы о системе и используйте ответы для улучшения своей модели. Теперь вы можете ответить на вопросы о системе, с которой вы работаете, опираясь на созданную модель. Правильность ответа на вопрос будет зависеть от системы, которую вы используете, но тут есть и несколько общих вопросов, которые актуальны для всех систем:
   - Какие элементы (классы, объекты, страницы) являются самыми важными в системе? Они есть в модели?
   - Каковы отношения между этими элементами?
   - Какова основная цель кода?
   - Как цель связана с важными элементами и отношениями между ними?
   - Как обычно используется код? Отражено ли это в модели?

Ментальные модели в долговременной памяти

До сих пор мы рассматривали ментальные модели как мышление, которое, согласно словам Джонсона-Лэрда, располагается в рабочей памяти. Однако существует другой подход к рассмотрению ментальных моделей, согласно которому модели располагаются в долговременной памяти.

Вторая книга, вышедшая в 1983 году, была написана Дедре Гентнером (Dedre Gentner) и Альбертом Стивенсом (Albert Stevens), исследователями из компании Bolt, Beranek, and Newman Inc. (BBN). В отличие от Джонсона-Лэрда они утверждали, что общие ментальные модели располагаются и хранятся в долговременной памяти, откуда их можно переместить в рабочую память.

Например, человек может помнить модель того, как течет жидкость, на примере наливания молока в стакан. Так как это общая ментальная модель, то люди понимают, что при наливании в миску теста для блинов жидкость будет вести себя чуть иначе, но сам процесс будет соответствовать запомненной ментальной модели.

Как это можно применить к программированию? Вы можете запоминать абстрактные представления о том, как работает обход дерева: вот корень, с которого вы начинаете работу, затем у вас есть два варианта: вы можете сделать обход по уровням, где вы исследуете все дочерние узлы, либо сделать обход в глубину, где вы исследуете один дочерний узел и все его дочерние узлы. Когда вы сталкиваетесь с программой, работающей с деревьями, вы можете вспомнить общую ментальную модель деревьев.

Описание ментальных моделей Гентнера и Стивенса похоже на схемы в долговременной памяти: ментальные модели помогают систематизировать данные и могут применяться для работы в новых ситуациях, похожих на те, с которыми вы сталкивались раньше. Например, вы, скорее всего, сможете понять, как выполняется обход дерева на незнакомом для вас языке программирования, если у вас уже есть усвоенная ментальная модель.

Создание ментальных моделей исходной программы в долговременной памяти

Такой взгляд на ментальные модели позволяет по-новому ими пользоваться. Вместо того чтобы создавать конкретные ментальные модели под определенную ситуацию, считают Гентнер и Стивенс, для более эффективного использования ментальных моделей нужно расширять запас их потенциальных разновидностей.

В предыдущих главах вы узнали о способе увеличения объема информации, хранимой в долговременной памяти. Одним из таких способов является использование дидактических карточек. На карточках, которые мы обсуждали в *главе 3*, с одной стороны была написала концепция, а с другой — соответствующий концепции код. Если вы хотите запомнить больше ментальных моделей, то вы можете воспользоваться данным способом. Однако теперь карточки будут содержать другую информацию. Теперь главная цель данного способа состоит не в расширении ваших знаний о синтаксисе кода, а в расширении ваших запасов ментальных моделей и способов размышления о коде. На одной стороне напишите название ментальной модели (подсказка), а на другой — краткое объяснение или визуализацию ментальной модели.

То, какие ментальные модели вы будете использовать при размышлении о коде, отчасти зависит от предметной области, языка программирования и архитектуры кода. Однако вот несколько вещей, которые стоит использовать:

- структуры данных, например ориентированные и неориентированные графы, а также различные формы списков;
- паттерны проектирования, например паттерн «Наблюдатель»;
- архитектурный паттерн, например «Модель — Представление — Поведение»;
- диаграммы, например диаграмма взаимосвязей или диаграмма последовательности действий;
- средства моделирования, например диаграмма состояний или сети Петри.

Существуют два способа использования набора карточек ментальных моделей. Вы можете использовать их для проверки знаний так же, как и синтаксические карточки: вы читаете подсказку, а затем пытаетесь вспомнить материал на обратной стороне. Как уже объяснялось ранее, если вы сталкиваетесь с незнакомой информацией, то вы можете добавить новую карточку к существующему набору карточек ментальных моделей.

Также вы можете использовать карточки ментальных моделей при чтении кода, с которым вы работаете. Просмотрите свои карточки и решите, можно ли каждую из моделей использовать при работе с этим кодом.

Например, вы выбираете карточку с деревом и спрашиваете себя: «А могу ли я представить этот код в виде дерева?». Если ваш ответ «да», то вы можете создавать ментальную модель на основе этой карточки. При работе с деревом это будет означать то, какие узлы, листья и ветви будут присутствовать в модели и что они могут представлять.

> **УПРАЖНЕНИЕ 6.3.** Создайте набор дидактических карточек с ментальными моделями, которые будут полезны при работе с имеющимся у вас кодом. На одной стороне карточки напишите название ментальной модели, а на другой — ее определение. К определению добавьте вопросы, которые нужно задавать во время применения этой модели к коду. Например, для кода с древовидной структурой вы будете моделировать узлы, листья и ветви, так что первый вопрос может звучать так: «Какие фрагменты кода могут быть представлены в виде листьев?». При использовании ментальной модели таблицы состояний вам нужно будет создавать список переменных, поэтому первый вопрос может звучать так: «Какие именно переменные?».
>
> Вы также можете выполнять это упражнение в команде и учиться на моделях коллег. Наличие общего запаса ментальных моделей может облегчить общение и работу над кодом всей команде.

Ментальные модели, одновременно хранящиеся в долговременной и рабочей памяти

Оба взгляда на ментальные модели до сих пор широко распространены. И хотя мнения о них могут показаться противоречащими друг другу, как мы уже увидели в этой главе, обе теории имеют свои преимущества и отлично дополняют друг друга. Проведенные в 1990-х годах исследования показали, что обе теории в какой-то сте-

пени верны: ментальные модели, хранящиеся в долговременной памяти, могут влиять на механизм создания ментальных моделей в рабочей памяти[2].

6.3. Условные машины

В предыдущем разделе мы рассмотрели ментальные модели: представления, которые формируются в нашем мозгу при рассуждении о задачах. Ментальные модели универсальны и могут использоваться во многих областях. В исследованиях языков программирования используется концепция *условной машины*. Ментальная модель может представлять собой модель всех вещей в мире, в то время как условная машина — это модель, которую мы используем для рассуждения о том, как именно компьютер выполняет код. А если точнее, условная машина — это абстрактное представление компьютера, на котором мы работаем и обрабатываем код.

Когда мы пытаемся понять, как работает программа или язык программирования, нас не интересует, как именно работает прибор под названием «компьютер». Нам все равно, как с помощью электричества хранятся биты. Вместо этого нам интересно влияние языка программирования на высоком концептуальном уровне, например нам интересен свопинг двух значений или поиск самого большого элемента в списке. Для обозначения различий между физическим компьютером и тем, что он делает на абстрактном уровне, мы используем термин «условная машина».

Например, условная машина для Java или Python может включать в себя концепцию ссылок, но может опустить адреса памяти. Адреса памяти могут рассматриваться как детали реализации, которые необязательно знать при работе на Java или Python.

Условная машина — это правильная абстракция выполнения языка программирования. Она может быть неполной, как мы только что увидели на примере. Таким образом, условные машины отличаются от ментальных моделей, которые могут быть ошибочными или противоречивыми.

Я нашла один способ, с помощью которого можно хорошо понять разницу между условной машиной и ментальной моделью. Суть способа состоит в том, что условная машина — это объяснение того, как именно работает компьютер. Когда вы поняли то, как работает условная машина и можете спокойно пользоваться ею, она становится ментальной моделью. Чем больше вы знаете об используемом языке программировании, тем более правильной и похожей на условную машину будет ваша ментальная модель.

6.3.1. Что такое условная машина

Перед тем как мы начнем рассматривать примеры условных машин, давайте разберемся с этим загадочным термином «условная машина». Первое, на что стоит обра-

---

[2] См. «Toward a Unified Theory of Reasoning», Джонсона-Лэрд и Khemlani, **https://www.sciencedirect.com/science/article/pii/B9780124071872000010**.

тить внимание, — это то, что условная машина представляет собой *машину*, т. е. объект, с которым можно взаимодействовать. Это важная характеристика, которая отличает условные машины от ментальных моделей, которые мы создаем, например, при занятии физикой или химией. И хотя мы можем проводить эксперименты для понимания окружающего мира, существует множество вещей, которые нельзя понять путем эксперимента (или это просто небезопасно). Например, при создании ментальной модели поведения электронов или проникающей радиации вы вряд ли найдете дома или на работе безопасную экспериментальную установку. В программировании же есть устройство, с которым мы можем взаимодействовать в любое удобное для нас время. Условные машины создаются для корректного понимания машины, выполняющей код.

Другая часть термина — это *условная* (notional). Согласно Оксфордскому словарю, это слово обозначает «основанное на предположении, оценке или теории; не существует в реальном мире». Когда мы рассматриваем метод работы компьютера, мы не хотим знать все до малейших подробностей. Нас интересует теоретическая модель работы компьютера. Например, когда переменной `x` присваивается значение 12, мы не думаем об адресе памяти, где хранится значение переменной, и указателе, который связывает переменную `x` с адресом памяти. Можно просто представить объект `x` с данным значением, который где-то живет. Условная машина представляет собой абстракцию, которую мы используем при рассуждении о работе компьютера на том уровне абстракции, который необходим в данное время.

6.3.2. Примеры условных машин

Условные машины придумал Бен дю Булай (Ben du Boulay), профессор Сассекского университета, когда он занимался разработкой Logo в 1970-х годах. Logo — это образовательный язык программирования, разработанный Сеймуром Пейпертом (Seymour Papert) и Синтией Соломон (Cynthia Solomon). Это был первый язык, в котором использовалась *черепашья графика* — черепаха, передвигающаяся по экрану, рисующая линии и управляемая кодом. Название происходит от греческого слова *logos*, означающего «слово» и «понятие».

Впервые дю Булай использовал термин «условная машина» для описания стратегии обучения детей и учителей языку Logo. Он описал этот термин как «идеализированную модель компьютера, содержащую в себе конструкции языка программирования». Хотя в объяснениях дю Булая использовались сделанные вручную визуализации, в основном использовались аналогии.

Например, при создании аналогии для языковой модели исполнения дю Булай использовал рабочего на заводе. Рабочий может выполнять команды и функции. У него есть уши, с помощью которых он может услышать значения параметров, у него есть рот, с помощью которого он может произнести выходное число, а также руки, которые выполняют действия, указанные в коде. Изначально представление концепций программирования было простым, однако со временем создавались объяснения всего языка Logo, включая встроенные команды, пользовательские процедуры и функции, подпрограммные вызовы и рекурсию.

Как вы уже видели, условные машины предназначены для объяснения метода работы устройства, которое выполняет код, поэтому они имеют несколько общих характеристик с этими устройствами. Например, как у физического компьютера, так и у условной машины есть понятие «состояния». Когда мы думаем о переменной как о коробочке, то эта виртуальная коробочка может быть пустой или иметь что-то внутри.

Существуют другие формы условных машин, которые не так зависимы от устройства реального компьютера. Скорее всего, вы используете абстрактные представления устройства, которое работает в тот момент, когда вы читаете или пишете код.

Например, размышляя о том, как именно работает вычисление на языках программирования, мы можем сравнивать работу компьютера с работой математика. В качестве примера давайте рассмотрим следующее выражение на Java:

```
double celsius = 10;
double fahrenheit = (9.0/5.0) * celsius + 32;
```

Когда вам нужно узнать значение `fahrenheit`, вы, скорее всего, воспользуетесь методом подстановки: замените переменную `celsius` во второй строке на значение 10. Затем вы можете мысленно добавить скобки, указывая приоритет оператора:

```
double fahrenheit = ((9.0/5.0) * 10) + 32;
```

Основанные на преобразовании вычисления в уме являются подходящей моделью того, что можно вычислить, но она не показывает вычисления, которые совершает машина. Конечно, работа физического компьютера отличается. Скорее всего, компьютер будет использовать стек для проведения оценки. Компьютер преобразует выражение в обратную польскую нотацию и поместит в стек результат выражения `9.0/5.0`, а затем извлечет его, умножит на 10 и вернет результат для дальнейших вычислений. Это хороший пример условной машины, которая немного ошибается, но все еще остается полезной. Мы можем назвать это «условной машиной подстановки», и она будет ближе к ментальной модели большинства программистов, а не к стековой ментальной модели.

6.3.3. Разные уровни условных машин

Вы увидели несколько примеров условных машин. Некоторые из них работают на уровне языка программирования и абстрагируются от всех деталей базовой машины, как, например, условная машина подстановки.

Другие условные машины, например представление стека как стопки листов бумаги, намного лучше отражают то, как физический компьютер выполняет код. При использовании условных машин как средств для объяснения и обучения концепциям программирования полезно думать о том, какие именно детали раскрывает и упускает условная машина. На рис. 6.4 показан обзор четырех уровней абстракции, на которых может работать условная машина. Для каждого уровня приведен пример. Например, «переменные как коробочки» играют роль на уровне языка программирования и на уровне компилятора/интерпретатора, однако они абстрагируют детали о скомпилированном коде или операционной системе.

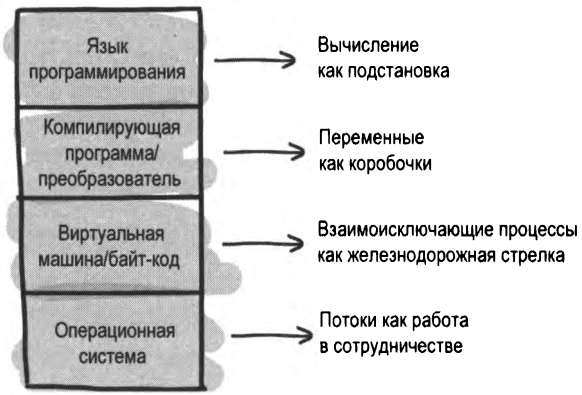


Рис. 6.4. Уровни, на которых условная машина может использовать абстракции. Например, при «вычисление как подстановка» условная машина игнорирует всё, кроме языка программирования, а при «потоки как работа в сотрудничестве» игнорирует всё, кроме операционной системы

Иногда важно знать, от каких деталей вы абстрагируетесь при работе с кодом. Хотя это и хороший способ, который помогает достигнуть высокого уровня понимания кода, некоторые из форм размышления о коде могут игнорировать важные детали.

УПРАЖНЕНИЕ 6.4. Приведите три примера условных машин и уровни абстракции, на которых они работают. В следующую таблицу запишите ваши примеры условных машин и уровни абстракции.

Условная машина	Язык программирования	Компилятор/ интерпретатор	Виртуальная машина/байт-код	Операционная система

6.4. Условные машины и язык

Мы часто используем условные машины не только для того, чтобы понимать принцип работы компьютера, но и для того, чтобы обсуждать сам код. Мы говорим, что переменная «хранит» значение, даже если это не физический объект, в котором что-то хранится. Эти слова указывают на ментальную модель переменной, которая похожа на коробочку с содержимым.

В программировании существует много примеров выражений, которые подразумевают условные машины и приводят к определенным ментальным моделям. Например, мы говорим, что файл «открыт» или «закрыт», технически имея в виду, что нам разрешено или запрещено читать файл. Мы используем слово «указатель» и говорим, что он «указывает» на какое-то значение; мы говорим, что функция «возвращает» значение, когда она помещает значение в стек, после чего «вызов» (тоже ментальная модель) может использовать это значение.

Условные машины, использующиеся для объяснения того, как все работает, закрепляются в языке, на котором мы говорим о коде, и даже в самих языках программирования. Например, концепция указателя представлена во многих языках программирования, и многие среды разработки позволяют узнать точное место, где «вызывается» та или иная функция.

УПРАЖНЕНИЕ 6.5. Приведите в качестве примера еще три выражения из области программирования, которые указывают на использование какой-либо условной машины и ментальной модели.

6.4.1. Расширяем набор условных машин

Ранее я использовала термин «условная машина», словно в каждом определенном моменте существует только одна условная машина. На самом деле языки программирования имеют скорее набор условных машин, а не одну общую условную машину. Например, при изучении простых типов данных вы могли услышать, что переменная — это коробочка со значением внутри. В дальнейшем вы узнали о составных типах, которые рассматриваются как стопка коробочек с простым значением внутри каждой. На рис. 6.5 показано, как две условные машины связаны друг с другом.

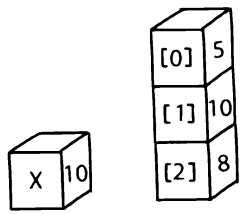


Рис. 6.5. Две условные машины, которые можно совместить друг с другом. Слева переменная представлена как коробочка, а справа массив представлен как стопка коробочек

Есть множество других примеров расширения набора абстракций, которые мы используем для понимания концепций языка программирования. Давайте рассмотрим условную машину, которая используется для размышлений о передаче параметра в языке, который поддерживает функции. Сначала вы можете думать о функции без

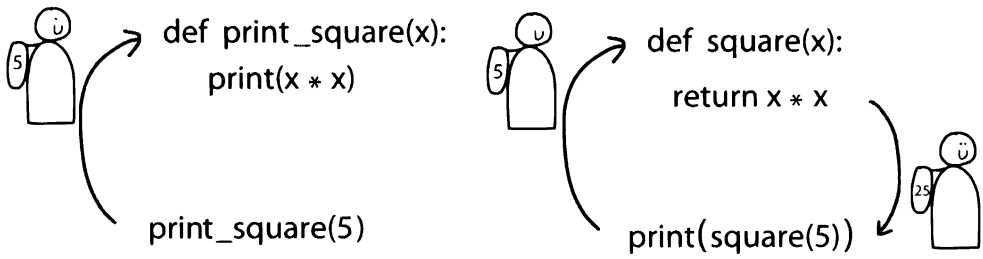


Рис. 6.6. Две условные машины для операторов. Слева расположена условная машина, поддерживающая входные параметры, а справа — модель, которая содержит входные и выходные параметры

параметров как о контейнере для нескольких строк кода. Когда появляются входные параметры и мы переходим от процедуры к функции, то функцию можно рассматривать как путешественника, который упаковывает вещи в рюкзак и переносит значения в место вызова. Когда появляются еще и выходные параметры, то путешественник приносит в своем рюкзаке значение обратно, как это показано на рис. 6.6.

6.4.2. Разные условные машины могут создать взаимно конфликтующие ментальные модели

В предыдущем разделе мы узнали, что некоторые условные машины могут быть совмещенными: например, условная машина переменной в виде коробочки, которая вместе с условной машиной массива представляет собой стопку коробочек. Однако условные машину могут создать ментальные модели, которые конфликтуют друг с другом.

Например, условная машина, которая представляет переменную как коробочку, отличается от условной машины, представляющей переменную как метку с именем. Эти две условные машины не могут быть объединены в одну совмещенную ментальную модель; мы представляем переменную либо как одну, либо как другую. Каждая из этих условных машин имеет свои плюсы и минусы. Например, переменная в виде коробочки говорит о том, что переменная может содержать несколько значений точно так же, как в коробочке может быть несколько монет или конфет. Такой (ошибочный) образ мышления маловероятен, если вы думаете о переменной как о метке с именем или стикере. Стикер можно наклеить только на один предмет, так что переменная может использоваться для описания одного значения. В 2017 году моя исследовательская группа провела исследование в научном музее NEMO в Амстердаме, где мы изучали данную концепцию[3]. В исследовании приняли участие 496 человек, которые не имели никакого опыта программирования. Они прошли вводный урок программирования на языке Scratch. Scratch — это блочный язык программирования, созданный для детей командой программистов из Массачусетского технологического института. Хотя язык предназначен для начинающих, которые только делают первые шаги в программировании, на нем могут работать и опытные программисты: например, на Scratch можно работать с переменными. Вы можете создать переменную нажатием кнопки и вводом имени переменной, а установить значение переменной можно с помощью блока программирования, изображенного на рис. 6.7.

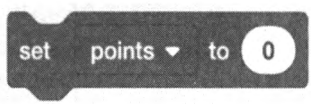


Рис. 6.7. Присвоение переменной `points` значения `0` на Scratch

[3] «Thinking Out of the Box: Comparing Variables in Programming Education» (2018), **https://dl.acm.org/doi/10.1145/ 3265757.3265765**.

В нашем исследовании все участники познакомились с концепцией переменной, но участники получили разный материал. Одна половина участников, группа «меток», прошла вводный урок программирования, на котором мы объясняли переменную как метку, например температуру или возраст человека. Другая половина, группа «коробочек», прошла урок, на котором мы объясняли переменную как коробочку, например копилку или обувную коробку. На обоих уроках мы использовали одну и ту же метафору: например, мы использовали выражение «x содержит 5?» для группы «коробочек» и «x это 5?» для группы «меток».

После прохождения вводного урока участников проверили на понимание программирования. Им задавались простые вопросы о переменных с одним значением, однако были и такие вопросы, где одной переменной было дважды присвоено значение. Мы задавали такие вопросы для того, чтобы выяснить, усвоили ли участники то, что переменная может содержать только одно значение.

Результаты исследования показали, что обе метафоры имеют свои плюсы и минусы. Группа «коробочек» лучше справилась с простыми вопросами, когда переменным присваивали только одно значение. Мы думаем, что группа «коробочек» ответила на эти вопросы лучше из-за того, что люди все время кладут свои вещи в коробки. Следовательно, визуализация концепции коробочки для хранения значения переменной помогает участникам понять саму суть. Однако, когда мы проанализировали результаты того, сколько участников были уверены в том, что переменной может быть присвоено два значения, мы увидели, что участники из группы «коробочек» чаще давали неверные ответы.

Важнейшим выводом нашего исследования является то, что нужно быть осторожным при описывании концепции программирования и принципа работы компьютера в терминах объектов и процессов из реального мира. И хотя эти метафоры могут быть полезными, они также могут создать заблуждение, особенно из-за того, что устаревшие ментальные модели все еще хранятся в долговременной памяти и могут переходить в рабочую память.

УПРАЖНЕНИЕ 6.6. Представьте условную машину, которую вы обычно используете при рассуждении о коде или при объяснении кода. Какие недостатки или ограничения ментальных моделей создаются этой условной машиной?

6.5. Условные машины и схемы

У использования условных машин могут быть свои недостатки, но они очень полезны в качестве средства размышления о программировании. Условные машины связывают концепции программирования с вещами из реального мира, для которых люди уже создали схемы.

6.5.1. Почему схема важна

Схемы — это способ, с помощью которого информация хранится в долговременной памяти. Например, у людей есть прочные ассоциации с идеей коробочки. Склады-

вание и вытаскивание вещей, открытие и закрытие коробочки — это то, с чем знакомы все люди. Следовательно, представление переменной как коробочки не вызывает дополнительной когнитивной нагрузки. Если вместо этого мы бы сказали: «Переменная похожа на одноколесный велосипед», то данная ассоциация была бы не такой полезной именно из-за того, что у большинства людей нет ментальной модели операций, которые связаны с одноколесным велосипедом.

То, что узнают люди, не имеет привязки к месту или времени. Так что при объяснении концепции необходимо выбрать сравнение, с которым человек точно будет знаком. Например, при объяснении детям из сельских районов Индии функциональных возможностей компьютера многие учителя в качестве примера компьютеров использовали слонов, а в качестве программистов — их хозяев, т. к. дети знакомы с данной концепцией.

6.5.2. Являются ли условные машины семантическими

Мое определение условных машин может напомнить вам определение *семантики* компьютерной программы. Семантика — это подраздел информатики, который занимается изучением *смысла* программы, а не ее внешнего вида, того, что обычно называется *синтаксисом*. У вас может возникнуть вопрос: указывает ли условная машина языка программирования на его семантику? Однако цель семантики — формализовать работу компьютера в виде математических уравнений и с математической точностью. Их целью является не игнорирование деталей, а их точное и полное определение. Иначе говоря, условные машины — это не просто семантика.

Выводы

- Ваше представление задачи может сильно повлиять на то, как вы будете о ней размышлять. Например, представление клиентов в виде списка, а не коллекции может повлиять на то, как вы храните и обрабатываете объекты клиента.
- Ментальные модели — это ментальные представления, которые мы создаем при размышлении о какой-либо задаче. Люди могут иметь несколько ментальных моделей, которые могут конфликтовать друг с другом.
- Условные машины — это абстрактные версии того, как работает физический компьютер. Обычно условные машины используются для объяснения концепций программирования и рассуждения о программировании.
- Условные машины помогают нам понимать программирование, т. к. с помощью условных машин мы применяем к программированию уже существующие схемы.
- Условные машины иногда отлично дополняют друг друга, но в то же время они могут создать конфликтующие ментальные модели.

7 Заблуждения

В этой главе вы узнаете:

- как знание одного языка программирования может помочь выучить новый язык;
- как избежать проблем при изучении второго языка программирования;
- как мозг сохраняет неверные представления и как они приводят к ошибкам в коде;
- как избежать заблуждений и предотвратить ошибки.

В последних нескольких главах мы рассмотрели методы размышления о коде, например визуализацию, использование фреймворков для поддержания рабочей памяти, а также использование ментальных моделей для решения проблем с кодом. Какими бы полезными ни были методы, которые мы используем для поддержки нашего мышления, мы все равно будем совершать ошибки, размышляя о коде.

В этой главе мы подробно рассмотрим ошибки кода. Ошибки могут быть результатом небрежности, например когда вы забыли закрыть файл или неправильно написали имя файла. Однако чаще ошибки в коде возникают из-за ошибок в мышлении. Вы можете не знать, что как только вы заканчиваете работать с файлом, его надо закрыть, или вы можете думать, что язык программирования автоматически закрывает файл за вас.

Сначала мы рассмотрим тему изучения нескольких языков программирования. Существует много причин ошибочных предположений при изучении нового языка; одной из них является то, что разные языки программирования имеют разные условные обозначения для всевозможных концепций. Например, Python закроет файл после фрагмента, начинающегося с `open()`, и для этого не надо писать оператор `file.close()`. Однако если вы будете писать код на C, то вам всегда нужно будет использовать оператор `fclose()`. В первой части главы я покажу вам, как можно

использовать имеющиеся знания для изучения новых языков программирования, а также как избежать ошибок, которые возникают из-за различий в изучаемых языках.

Во второй части главы мы рассмотрим ошибочные предположения о коде. Мы рассмотрим связанные с кодом заблуждения, а также то, откуда они взялись. Понимание того, какие заблуждения влияют на код, поможет вам предсказать и предотвратить многие ошибки.

7.1. Почему второй язык программирования выучить намного проще, чем первый

В предыдущих главах вы узнали, что ключевые слова и ментальные модели, которые хранятся в вашей долговременной памяти, помогают вам понять код. Иногда приобретенные знания могут быть полезны в нескольких областях. Это называется *трансференцией (переносом)*. Перенос происходит в том случае, когда информация, которую вы уже знаете, помогает вам изучать что-то новое. Например, если вы умеете играть в шашки, то вам будет легче научиться играть в шахматы, т. к. некоторые правила в этих играх похожи. Это касается и программирования: если вы знаете язык Java, то учить Python будет проще, т. к. вы уже знакомы с такими главными концепциями программирования, как переменные, циклы, классы и методы. К тому же навыки, которые вы приобрели во время программирования, могут помочь вам при изучении второго языка программирования.

Есть два способа, с помощью которых знания, хранящиеся в долговременной памяти, могут помочь выучить новые концепции программирования. Во-первых, если вы уже много чего знаете о программировании (или любой другой области), то вы с легкостью будете узнавать что-то новое. Сама информация, хранящаяся в долговременной памяти, помогает вам узнавать новое. Это называется *трансференцией во время обучения*.

В *главе 2* вы узнали, что вся новая информация через сенсорную память переходит в кратковременную память, а затем переходит в рабочую память. Этот процесс показан на рис. 7.1. Когда вы активируете рабочую память путем обдумывания новой концепции программирования, в этот момент активируется долговременная память, которая начинает поиск актуальной информации.

Как показано на рис. 7.1, долговременная память может найти воспоминания, связанные с новой информацией. Если это происходит, то эти воспоминания переходят в рабочую память. Эти воспоминания могут представлять собой процедурную память, схемы, планы и событийную память.

Например, если вы уже знаете Java и решили выучить методы на Python, то вы можете вспомнить методы на Java. Это поможет вам быстро выучить методы на Python, даже если работа методов на двух языках немного отличается.

В *главе 3* мы рассматривали использование проработки при изучении новой концепции программирования. Проработка — это привязка новой информации к знаниям, которые вы имеете. Польза данного способа заключается в том, что активный

поиск дополнительной информации в долговременной памяти помогает выучить новый материал легче и быстрее. Следовательно, проработка помогает улучшить трансференцию.

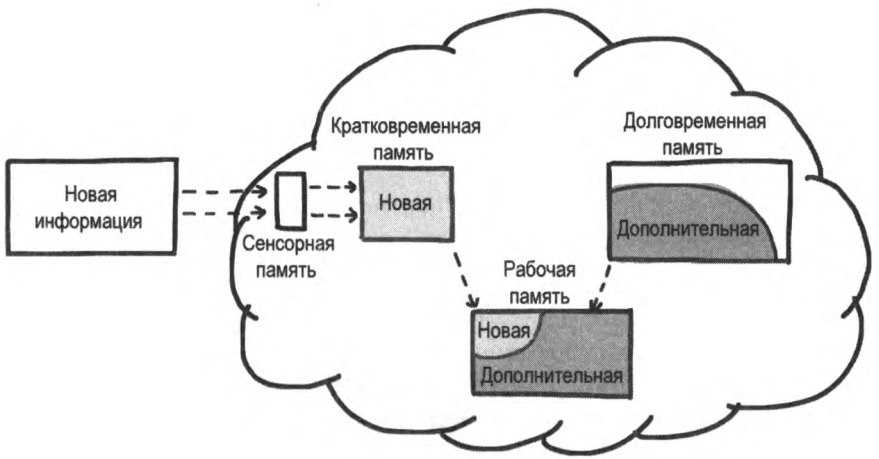


Рис. 7.1. Когда вы узнаете новую информацию, сначала она обрабатывается сенсорной памятью, а затем передается в кратковременную память. В это время долговременная память начинает поиск дополнительной информации. Если ДП находит дополнительную информацию, то эта информация тоже передается в рабочую память для более эффективного и активного обдумывания новой информации

УПРАЖНЕНИЕ 7.1. Подумайте о недавно изученной концепции программирования или библиотеке. Знание каких концептов помогло вам выучить новый концепт?

Вторым способом, которым хранящиеся в долговременной памяти знания помогают учить что-то новое, является *трансференция обучения (перенос модели обучения)*. Трансференция обучения происходит тогда, когда вы применяете накопленные знания и опыт для решения незнакомых задач. Когда люди говорят о когнитивистике и используют термин «трансференция», они практически всегда говорят о трансференции обучения.

Иногда трансференция обучения происходит, даже когда вы об этом не думаете. Например, вы покупаете новую пару брюк, но, примеряя их, вы даже не задумываетесь о том, как нужно застегивать пуговицу. Вы просто понимаете, что вам нужно делать, даже если эта модель брюк или пуговица вам незнакомы. То же самое происходит, когда вы покупаете ноутбук: вы знаете, как работает его клавиатура, даже если никогда раньше не работали на данной модели ноутбука. Трансференция обучения может происходить и сознательно, например когда вы изучаете новый язык программирования. Если вы уже знаете язык Python и решите учить JavaScript, то у вас возникнет мысль: «Я знаю, что в теле цикла на Python надо делать отступ. А на JavaScript так же?»

Трансференция обучения похожа на трансференцию во время обучения, т. к. в обоих случаях долговременная память ищет нужную стратегию, которую можно применить в той или иной ситуации.

7.1.1. Как увеличить шанс воспользоваться знаниями по программированию

Будучи профессиональным программистом, я уверена, что вы хоть раз попадали в ситуацию, когда можно было провести трансференцию знаний, но вы не могли этого сделать. Возможно, вы не поняли, как работает та или иная функция в библиотеке, а затем узнали, что совершенно так же, как и в библиотеке, которую вы уже знали. К сожалению, не всегда полезные знания можно применить в новой ситуации.

Количество знаний, которое вы можете перенести от одной задачи к другой, зависит от множества факторов и может сильно различаться. Вот факторы, которые влияют на это количество:

- *Уровень владения* — насколько хорошо вы решили задачу, информация о которой уже сохранена в долговременной памяти. Чем лучше вы понимаете задачу, тем больше вероятность того, что полученные знания вы сможете использовать при решении другой задачи. Например, опытный программист на Java получит больше пользы от дополнительных знаний при изучении Python, чем начинающий программист на Java. Как мы уже видели в предыдущих главах, опытный программист владеет бóльшим количеством стратегий и ментальных моделей, которые можно использовать для решения задач на любом языке программирования.
- *Сходство* — это то, насколько похожи задачи. Например, реализовать знакомый алгоритм на незнакомом языке программирования будет проще, чем реализовать на незнакомом языке программирования новый алгоритм.
- *Контекст* — насколько похож контекст. Важна не только схожесть задач, но и контекст, в котором вы решаете задачи. Например, если вы программируете на двух разных языках в одной интегрированной среде разработки, то трансференция между этими двумя языками произойдет с большей вероятностью. Именно этот факт является хорошим аргументом для программирования на нескольких языках в одной интегрированной среде разработки. Однако контекст не ограничивается одним компьютером: значение может иметь даже то, работаете ли вы в одном и том же офисе с одними и теми же людьми. Чем больше похожих вещей, тем выше шанс трансференции знаний.
- *Качественные признаки* — ваше понимание того, какие знания могут пригодиться в будущем. Если кто-то скажет, что знание JavaScript поможет выучить Python, то вы с большей вероятностью начнете искать сходства между языками. Поэтому вместо того, чтобы сразу начинать учить новый язык программирования или концепт, сначала найдите общие признаки между тем, что вы хотите изучать, и тем, что вы уже знаете. Какие ваши знания могут помочь вам выполнить новую задачу?
- *Ассоциации* — как сильно вам кажется, что задачи похожи. Например, Java и JavaScript кажутся похожими, хотя это два разных языка. Следовательно, в вашей долговременной памяти может существовать сильная связь между Java и

JavaScript, но слабая связь между Python и Scala. Ваша событийная память, хранящая воспоминания о пережитых событиях, также играет большую роль в построении связей между воспоминаниями и знаниями. Например, если вы учили Java и C# в одном и том же кабинете, то связь между ними будет сильнее, чем если бы вы учили их в разных кабинетах.

- *Эмоции* — ваше отношение к задаче. Ваши эмоции могут влиять на возможность трансференции. Например, если вам нравится работать с двоичными деревьями и новая задача очень похожа на них, то вы захотите примерить те же самые стратегии для решения новой задачи.

7.1.2. Разные виды трансференции

Существует несколько критериев деления трансференции на виды. Если у вас есть достаточный словарный запас для различных видов трансференции, то вы будете строить более реалистичные ожидания о трансференции между языками программирования. Некоторые программисты считают, что синтаксис языков программирования не играет большой роли, и если вы знаете один язык, то вы сможете легко выучить второй и третий языки. Конечно, знание одного языка облегчит изучение второго, но такая стратегия не всегда работает. Знание видов трансференции поможет вам более эффективно изучать новые языки и фреймворки.

Осознанная и неосознанная трансференция

Трансференция автоматизированных умений отличается от трансференции умений, которые были получены осознанно. Трансференция автоматизированных умений называется *неосознанной трансференцией*. В программировании такая трансференция может произойти, когда вы неосознанно нажимаете <Ctrl>+<C> и <Ctrl>+<V> в новом редакторе. Трансференция более сложных задач называется *осознанной трансференцией*. При этом виде трансференции вы действуете чаще всего осознанно. В качестве примера данного вида трансференции можно представить ситуацию, когда вы работаете с новым языком программирования и решаете объявить переменную, потому что знаете, что так делается на большинстве языков программирования.

Близкая и дальняя трансференция

Ранее вы узнали, что чем больше две области похожи, тем больше умений можно перенести. Другой способ деления трансференций на виды — это оценивание расстояния между предметными областями. *Близкая трансференция* — перенос умений происходит между областями, близкими друг к другу, например между исчислением и алгеброй или C# и Java. *Дальняя трансференция* — это перенос умений между очень разными областями, например латинский язык и логика или Java и Prolog. Так как сходство является фактором, влияющим на трансференцию, успешность дальней трансференции намного меньше, чем близкой трансференции.

УПРАЖНЕНИЕ 7.2. Вспомните несколько ситуаций, когда вы сталкивались с трансференцией. Какой это был вид трансференции? Заполните следующую таблицу:

Ситуация	Неосознанная	Осознанная	Близкая	Дальняя

7.1.3. Знания: добро или зло?

Помимо перечисленных в предыдущем разделе видов трансференции существуют еще две главные категории. Вид трансференции, когда знание чего-то помогает изучать новую тему или решить новую задачу (именно этот вид мы до сих пор рассматривали), называется *положительной трансференцией.*

Когда происходит положительная трансференция, вам не нужно заново создавать ментальную модель, т. к. в вашей памяти уже существуют похожие ментальные модели, которые долговременная память использует в других областях. Например, когда вы знаете язык Java, то у вас уже создана ментальная модель цикла и вы знаете, что у цикла есть переменная-счетчик, тело и условие прерывания. Практически в любом языке программирования все циклы будут иметь эти характеристики, и это позволяет вам создать новую ментальную модель. Однако, как вы уже могли заметить, трансференция не всегда бывает положительной. Когда имеющиеся знания мешают вам получать новые знания, то это *негативная трансференция.* Эдсгер Вибе Дейкстра (Edsger W. Dijkstra), нидерландский профессор компьютерных наук и создатель «алгоритма Дейкстры», сказал, что обучение языку BASIC стоит прекратить и запретить, т. к. это «калечит мозги».

И хотя я не верю в то, что наш мозг можно повредить изучением какого-нибудь языка программирования, доля правды в этом высказывании есть, потому что причиной ошибок могут быть неверные предположения о коде. Неверные предположения, в свою очередь, могут быть вызваны негативной трансференцией. Например, на Java нельзя использовать переменные без инициализации. Опытный программист, работающий на Java, может предположить, что при работе на Python все переменные тоже должны инициализироваться и что компилятор предупредит его, если он забудет это сделать. Это может вызвать замешательство и привести к ошибкам.

Даже в похожих языках, таких как Java и C#, существует риск негативной трансференции, т. к. ментальные модели этих языков похожи, но не идентичны. Например, на Java есть концепция, которая называется *проверяемыми исключениями* — это исключения, которые проверяются на этапе компиляции. Если вы не укажете эти исключения в блоке `try-catch`, то скомпилировать код не получится. Проверяемые исключения — это особенность языка Java, так что люди, только начинающие учить этот язык, могут не понимать того, что он отличается от того, к чему они обычно привыкли. Они не только не признают факта, что придерживаются неправильной ментальной модели, но и доказывают всем обратное!

Если вы забудете инициализировать переменную или неправильно обработаете исключение, то эти относительно небольшие ошибки можно легко исправить. Однако есть примеры более серьезной негативной трансференции. Например, многие программисты, знакомые с объектно-ориентированными языками, испытывают трудности с изучением функциональных языков, например F#, т. к. в обеих парадигмах имеются функции, но работают они совершенно по-другому.

УПРАЖНЕНИЕ 7.3. Вспомните ситуацию, когда вы сделали неверное предположение о какой-нибудь концепции языка программирования. Могла ли на это повлиять негативная трансференция из одного языка в другой?

7.1.4. Сложности трансференции

В предыдущем разделе вы увидели, что трансференция знаний может быть положительной и негативной и что положительная трансференция не является чем-то само собой разумеющимся. Для того чтобы произошла положительная трансференция, две ситуации должны быть очень похожи. Дальняя трансференция, когда знания из одной области «перепрыгивают» в другую, не очень похожую область, редко происходит спонтанно.

К сожалению, исследования показывают, что процесс трансференции очень сложен, и у большинства людей этот процесс не происходит автоматически. Шахматы очень часто считают одним из возможных источников трансференции, т. к. многие люди уверены, что умение играть в шахматы улучшает интеллект, логическое мышление и память. Однако ученые до сих пор не подтвердили данное предположение. В *главе 2* мы обсуждали работу Адриана де Гроота: результаты его эксперимента показали, что при запоминании случайной расстановки фигур память опытных шахматистов не лучше, чем у начинающих. Другие эксперименты подтвердили эти результаты и показали, что опытные шахматисты не всегда лучше запоминают числа или визуальные фигуры. Скорее всего, навыки игры в шахматы нельзя применить при игре в другие логические игры, например Лондонский Тауэр (головоломка, похожая на головоломку Ханойская башня).

Похоже, что это верно и для программирования. Многие программисты утверждают, что умение программировать улучшает навыки логического мышления и даже повышает общий интеллект. Однако существует несколько исследований, в которых изучались когнитивные эффекты программирования, и результаты показали закономерность, похожую на закономерность в шахматах. В 1987 году Гавриел Саломон (Gavriel Salomon) из Тель-Авивского университета провел обзорное исследование, и результаты показали, что большинство исследований, рассматривающих влияние обучения программированию, не имеют сильного эффекта. Проведенные Саломоном исследования показывают, что дети легко приобретают некоторые навыки программирования, но эти навыки не переносятся в другие когнитивные области.

Из этого можно сделать вывод: то, что вы выучили один язык программирования, не дает гарантию того, что полученные знания помогут вам выучить второй язык.

Это может расстроить, потому что вы уже представляете себя профессиональным программистом, а медленное изучение материала на уровне начинающего и использование стратегий в виде дидактических карточек для изучения синтаксиса могут показаться ненужными. Я могу дать вам еще один совет: если вы собираетесь учить новый язык и хотите улучшить свое мышление, то выбирайте язык, который фундаментально отличается от тех, которыми вы уже владеете. Избегайте ложного расширения ваших вкусов от «музыки кантри» до «музыки из вестернов».

Итак, в данном разделе показано, что дальняя трансференция, например из SQL в JavaScript, маловероятна, и вам потребуется изучить большой объем нового синтаксиса, а также новых стратегий, чтобы достигнуть профессионального уровня на новом языке программирования. Практика тоже может отличаться: например, многое из того, что вы знаете о многократном использовании и абстракциях на JavaScript, будет неактуально для работы на SQL.

Обратите внимание на сходства и различия между двумя языками — так вы более эффективно будете учить новый язык.

УПРАЖНЕНИЕ 7.4. Подумайте о языке программирования, который вы учите в данный момент, или о языке, который вы хотите начать учить. Сравните его с языком (языками), который вы уже знаете. Что у этих языков общего? Чем они различаются?

Заполните следующую таблицу. Она поможет вам определить, где может произойти трансференция и на какие моменты следует обратить особое внимание.

	Сходства	Различия	Комментарии
Синтаксис			
Система типов			
Концепции программирования			
Время выполнения			
Среда разработки			
Среда тестирования			

7.2. Заблуждения. Ошибки в мышлении

Первая часть этой главы была посвящена трансференции знаний из одной предметной области в другую. Когда накопленная информация не позволяет решать нам новую задачу, то это называется негативной трансференцией. В этой части мы поговорим о последствиях негативной трансференции.

Представьте ситуацию, когда вы сделали ошибку в коде. Например, вы неправильно инициализировали экземпляр, вызвали неправильную функцию или допустили ошибку на единицу в списке. Ошибки могут быть вызваны забывчивостью или

невнимательностью программиста: вы случайно забыли код, использовали неправильный метод или допустили ошибку в вычислении граничного значения.

Однако есть и более серьезные причины ошибок в коде, когда при чтении кода вы делаете неверное предположение. Возможно, вы думали, что инициализация экземпляра будет происходить в другом месте кода, или думали, что используете правильный метод, или считали, что структура данных не позволит вам взаимодействовать с элементами за пределами структуры данных. Если вы уверены, что ваш код должен работать, но он все равно выдает ошибку, то, скорее всего, у вас *заблуждение*.

В повседневной речи слово «заблуждение» используется как синоним слов «ошибка» или «замешательство», однако формальное определение отличается. Для того чтобы убеждение стало заблуждением, оно должно:

- быть неверным;
- не меняться в разных ситуациях;
- внушать доверие.

Существует множество распространенных заблуждений. К примеру, многие считают, что самая острая часть перца чили — это его семена. Но семена чили совсем не острые! Это заблуждение, потому что:

1. Оно неверное.
2. Если люди верят, что семена одного сорта перца чили острые, то они будут считать, что семена всех сортов перца чили острые.
3. Они верят в то, что это правда, поэтому при готовке удаляют семена перца.

Заблуждение «семена чили острые» — это результат слухов, которые люди просто рассказывали друг другу. Однако негативная трансференция очень часто играет роль в создании заблуждений. Например, многие люди верят, что при обжарке мяса «соки запечатываются внутри», т. к. другие продукты, например яйца, слегка твердеют при нагревании. Такие люди считают, что при жарке создается прочный «щит», который не выпускает содержащуюся внутри влагу. Знания об одном типе продукта ошибочно переносятся на другой тип, и это приводит к заблуждению — на самом деле при обжарке мясо теряет много влаги.

Заблуждения часто случаются и в программировании. Новички иногда думают, что переменная, например `temperature`, может содержать только одно значение, которое нельзя изменить. И хотя профессиональному программисту это покажется глупым, есть причины, из-за которых можно предположить, что у переменной может быть только одно значение. К примеру, это предположение может сложиться под влиянием знаний из области математики, где переменные не меняют своего значения в рамках одного доказательства или упражнения.

Другая причина возникновения этого заблуждения лежит в самом программировании. Я замечала, что студенты, которые раньше работали с файлами и файловыми системами, проводили неверную трансференцию знаний о файлах на переменные. Так как операционные системы позволяют создавать только один файл с опреде-

ленным именем (в одной папке), то студенты могут ошибочно предположить, что переменная `temperature` уже используется в коде и не может содержать второе значение, точно так же, как имя файла может использоваться только для одного файла.

7.2.1. Исправление заблуждений путем концептуальных замен

Заблуждения — это ошибочное мышление, от которого очень трудно избавиться. Так как заблуждения вызывают у нас доверие, то бывает очень трудно изменить мнение о них. Очень часто бывает недостаточно просто указать на ошибку в чьем-то мышлении. Вместо того, чтобы пытаться изменить заблуждение, вам нужно заменить ошибочный образ мышления другим. Недостаточно сказать начинающему программисту, что значение переменной может измениться: они должны приобрести новое понимание переменной как концепции.

Замена заблуждения на основе языка программирования, который вы уже знаете, на верную ментальную модель для языка, который вы изучаете, называется *концептуальной заменой*. В данном случае существующая ментальная модель изменяется, замещается или усваивается новыми знаниями. Именно такое изменение знаний, а не добавление новой информации к уже существующей схеме отличает концептуальную замену от других способов обучения.

Сам факт того, что знания, которые вы уже выучили и которые хранятся в долговременной памяти, нужно изменить, делает обучение на основе концептуальной замены более трудным по сравнению с обычным обучением. Именно поэтому от заблуждений очень тяжело избавиться: понимание того, почему ваше мышление неверно, не всегда или вовсе не помогает.

Следовательно, при изучении нового языка программирования нужно потратить много времени и энергии на «забывание» существующих знаний об изученных языках программирования. Например, когда вы учите Python, при этом уже зная Java, вам придется забыть некоторые элементы синтаксиса, например то, что вам всегда нужно определять тип переменной. Вам придется отказаться от некоторых вещей: например, вы больше не сможете полагаться на тип переменной при принятии того или иного решения в коде. Несмотря на факт того, что Python характеризуется динамической проверкой типов, вам может потребоваться много времени для того, чтобы научиться думать о типах переменных, т. к. для этого требуется концептуальная замена.

7.2.2. Подавление заблуждений

Помните задачу, в которой мы надевали на снеговика свитер, а затем я просила вас решить, увеличится или уменьшится скорость таяния снеговика? Возможно, сначала вы решили, что скорость таяния увеличится, ведь когда вы надеваете свитер, вам становится теплее, верно? Но со снеговиком это работает иначе. Как только вы одеваете снеговика, свитер начинает удерживать холод и замедлять таяние.

Скорее всего ваш мозг активировал известную концепцию: свитера согревают. Данная концепция правдива для теплокровных, для людей, однако затем ваш мозг перенес ее на ситуацию со снеговиком. И это не из-за того, что вы глупый человек!

Долгое время считалось, что как только человек узнает то, как все работает, то устаревшие и неправильные представления удаляются из памяти, а их место занимает новая информация. Однако, учитывая то, что мы знаем о человеческом мозге, такая ситуация маловероятна. Считается, что воспоминания нельзя полностью забыть или заменить; просто с течением времени воспоминания извлекаются реже. Однако неверные представления все еще хранятся в вашей памяти, и ваш мозг может их использовать, даже если вы не хотите этого.

Исследования показали, что люди могут придерживаться старых представлений, несмотря на то, что они успешно пользуются верной информацией. Работа Игаля Галили (Igal Galili) и Варды Бар (Varda Bar) из Еврейского университета в Иерусалиме показала, что студенты хорошо справляются со знакомыми упражнениями, но при решении сложных упражнений они возвращаются к простым и неправильным рассуждениям[1]. Это показывает, что в памяти одновременно могут находиться несколько представлений; просто вспомните пример со снеговиком: для нас свитер означает тепло, и при этом у нас есть представление о том, что свитер сохраняет холод. Когда нам нужно решить, согревает ли свитер снеговика, эти два представления могут конфликтовать между собой. Мы должны абстрагироваться от представления о том, что свитер означает тепло. Возможно, вы пришли к правильному ответу в тот момент, когда у вас в голове возникло «Подождите-ка...» и вы начали рассуждать, а не доверять интуиции и старым знаниям.

Мы точно не знаем, как мозг решает то, какую из концепций нужно использовать, но мы знаем точно, что *сдерживание* играет в этом большую роль. Обычно мы ассоциируем сдерживание с чувством неловкости или стеснения. Недавние исследования указывают на то, что при активации сдерживающих механизмов наш мозг может выбрать верную концепцию.

УПРАЖНЕНИЕ 7.5. Представьте ситуацию, когда вы неправильно поняли концепцию определенного языка программирования. Например, я очень долгое время думала, что все функциональные языки используют «ленивое» вычисление и что все «ленивые» языки обязательно должны быть функциональными, т. к. я знала один ленивый и функциональный язык — это был Haskell. Какого заблуждения вы придерживались долгое время? Как оно возникло?

7.2.3. Заблуждения о языках программирования

Было проведено множество исследований на тему заблуждений в области программирования, а особенно тщательно рассматривались заблуждения, которых придерживаются начинающие программисты. Юха Сорва (Juha Sorva), сейчас уже доцент

[1] См. «Motion Implies Force: Where to Expect Vestiges of the Misconception?», Игаль Галили и Варда Бар (1992), **https://www.tandfonline.com/doi/abs/10.1080/0950069920140107**.

Университета Аалто в Финляндии, в 2012 году написал диссертацию, в которой он перечислил 162 заблуждения, с которыми сталкиваются начинающие программисты[2]. Все эти заблуждения исследуются. Советую вам ознакомиться со всем списком, но сейчас мы рассмотрим некоторые заблуждения, которые требуют отдельного внимания:

- *Заблуждение 15. Элементарное присваивание сохраняет уравнения и нерешенные выражения.* Это заблуждение указывает на то, что иногда люди предполагают, что присвоение переменных сохраняет отношения между ними. Придерживающийся этого заблуждения человек считает, что если мы напишем `total = maximum + 12`, то значение `total` будет как-то связано со значением `maximum`.

  Это приводит к тому, что если в коде значение `maximum` изменится, то человек будет ожидать изменения значения переменной `total`. Это заблуждение интересно тем, что оно очень логично. Представьте язык программирования, в котором отношения между переменными представлены в виде системы уравнений. Есть языки программирования, которые работают похожим образом, например Prolog.

  С этим заблуждением часто сталкиваются люди, имевшие дело с математикой. Связанное с этим заблуждение мы рассматривали ранее в этой главе, и оно заключается в том, что переменная может иметь только одно значение. Это верно действительно только для математики.

- *Заблуждение 33. Как только условие становится* `false`*, цикл* `while` *завершается.* Это заблуждение касается того, что люди не понимают, когда срабатывает условие прекращения цикла `while`. Люди, придерживающиеся данного заблуждения, полагают, что условие цикла проверяется в каждой строке и что цикл сразу же останавливается, когда условие становится `false`. Скорее всего данное заблуждение связано со значением, которые присвоено ключевому слову *while* (пока). Когда мы слышим от кого-то: «Я буду сидеть и читать книгу, пока идет дождь», мы думаем, что говорящий постоянно следит за погодой, и как только дождь заканчивается, он уходит, не дочитав книгу до конца. Данное заблуждение не является признаком того, что человек ничего не понимает в программировании или запутался. Вполне логично думать, что код, так похожий на английский язык, будет вести себя похожим образом.

  Данное заблуждение является хорошим примером того, как значение английских ключевых слов влияет на программирование. К тому же вы снова можете представить язык программирования, в котором условие остановки цикла `while` постоянно оценивается и цикл сразу же останавливается, когда условие становится `false`.

  Похожее заблуждение связано с тем, что имя переменной влияет на то, какое значение она может содержать: например, мы можем думать, что переменная

[2] См. таблицу A-1 (стр. 3593-68) в «Visual Program Simulation in Introductory Programming Education», **http://lib.tkk.fi/Diss/2012/isbn9789526046266/isbn9789526046266.pdf**.

с именем minimum не может содержать большое значение (заблуждение 17 из списка Сорвы)

- *Заблуждение 46. Для передачи параметров нужны разные имена переменных в сигнатуре и месте вызова.* Люди, придерживающиеся данного заблуждения, полагают, что имя переменной можно использовать только один раз, в том числе и при работе с функциями. Когда вы учитесь программированию, вы узнаете, что имена переменных можно использовать только один раз. Если вам нужно создать новую переменную, то вам придется давать ей новое имя. Однако, когда мы работаем с методами и функциями, а также их вызовами, то ограничение в одно имя для одной переменной не играет никакой роли. Оказывается, разрешается использовать одно и то же имя переменной как внутри функции, так и в другой части кода. В действительности это больше, чем просто разрешается, — использование одного и того же имени для разных переменных внутри функций и в другой части кода является обычной практикой. Такие примеры можно часто увидеть в описаниях функций. Например, при изучении функций вы очень часто можете встретить вот такой код:

```
def square(number):
return number * number

number = 12
print(square(number))
```

Такой код можно встретить и в реальной жизни. Например, при извлечении метода в среде разработки многие среды разработки будут дублировать имя переменной при объявлении и вызове функции. Так как в реальном коде это практикуется, то и преподаватели могут использовать это в обучении. Данный пример является интересным примером заблуждения, который передается внутри языка программирования; его причиной не являются имеющиеся знания по математике или английскому языку. Бывают моменты, когда мы понимаем определенную концепцию языка программирования, но трансференция полученных знаний в другие концепции этого же языка программирования не происходит.

7.2.4. Предотвращение заблуждений при изучении нового языка программирования

Мы практически ничего не можем сделать с заблуждениями. При изучении нового языка или системы программирования вы столкнетесь с негативной трансференцией. Однако существует несколько стратегий, которые могут помочь.

Во-первых, важно понимать то, что даже если вы поняли все правильно, вы все равно можете ошибаться. Сохраняйте объективность и не торопитесь.

Во-вторых, вы можете изучить популярные заблуждения, чтобы впоследствии избегать их. Возможно, вы не сможете сразу понять, когда вы делаете ошибочные предположения, а когда — нет. Поэтому будет полезно воспользоваться списком популярных заблуждений. В упражнении 7.5 вы познакомитесь с потенциальными

областями, в которых у вас могут возникнуть заблуждения. При изучении нового языка программирования вы можете опираться на список Сорвы и определить, каких заблуждений вам стоит опасаться. Используйте этот список для определения заблуждений, которые могут быть актуальны для изучаемого вами языка программирования.

Последний совет, который я могу вам дать, — не бойтесь задавать вопросы программистам, которые изучали в том же порядке те же языки программирования. Между каждой парой языков есть особая связь, которая может вызвать заблуждения. Так как их слишком много, я не буду их здесь перечислять. Иногда бывает полезно спросить совета у тех, кто мог оказаться в такой же ситуации и столкнуться с такими же заблуждениями.

7.2.5. Выявление заблуждений в новой базе кода

В предыдущих разделах мы в основном рассматривали заблуждения в языках программирования, вызванные негативной трансференцией знаний в новый язык программирования.

Аналогичным образом вы можете придерживаться ложных представлений о кодовой базе, с которой работаете. Делая предположения на основе предыдущего опыта работы с языком программирования, фреймворком, библиотекой, или о предметной области кода, значениях имен переменных, или о намерениях других программистов, вы рискуете впасть в заблуждение.

Одним из способов пресечения заблуждений является программирование в паре или большой группе. Вы будете сталкиваться с идеями и предложениями других программистов и вскоре поймете, что между вами и другими людьми будет что-то не сходиться — значит, кто-то придерживается заблуждения.

Профессиональному программисту (или эксперту в другой области) очень трудно осознать, что это его ошибка, поэтому всегда проверяйте свои предположения о коде: запускайте его или используйте комплексный тест. Если вы уверены в том, что определенное значение никогда не может быть ниже нуля, почему бы не добавить тест, чтобы это проверить? Тест не только определит, ошибаетесь вы или нет, но и подтвердит ваши догадки о том, что значение всегда должно быть положительным. Тест сообщит вам эту информацию в будущем, и это очень важно, потому что, как мы теперь знаем, заблуждения редко удаляются из памяти и могут появиться снова, даже если мы стараемся придерживаться верной модели.

Из этого можно сделать вывод, что документация — это третий способ, который может помочь избавиться от заблуждений об определенной функции, методе или структуре данных в кодовой базе. Если вы поняли, что придерживаетесь определенного заблуждения, то вы можете добавить в соответствующих местах в коде документацию, чтобы другие люди не начали придерживаться такого же заблуждения.

Выводы

- Хранящиеся в долговременной памяти знания можно использовать в незнакомых ситуациях. Иногда хранящиеся знания ускоряют процесс обучения или помогают лучше решать новые задачи. Это называется положительной трансференцией.
- Трансференция знаний из одной области в другую может быть негативной. Это происходит тогда, когда уже имеющиеся знания препятствуют получению новых знаний или решению новых задач.
- Вы можете использовать положительную трансференцию для быстрого и эффективного изучения нового материала. При положительной трансференции в вашей долговременной памяти ищется дополнительная информация, которая упрощает усвоение новой информации.
- Иногда вы можете заблуждаться в том, что ваши представления соответствуют действительности и что другие представления неверные.
- Осознание своей неправоты не всегда помогает избавиться от заблуждений. Для этого вам нужна новая ментальная модель, которая заменит устаревшую.
- Даже если вы создали правильную ментальную модель, вы все равно можете придерживаться ошибочного мнения.
- Во избежание заблуждений используйте тестирование и документирование кодовой базы.

Часть III

О хорошем коде

В *частях I* и *II* мы рассматривали роль кратковременной, долговременной и рабочей памяти при чтении кода и размышлении о коде. В *части III* мы сосредоточимся на том, как написать хороший код: как именно можно написать понятный код, использовать правильные имена и избегать «запаха» кода. Мы также обсудим, как можно улучшить навыки написания кода для решения сложных задач.

8 Совершенствуем навыки присваивания имен

В этой главе:

- сравним различные точки зрения на присваивание имен;
- рассмотрим взаимосвязь между именами и когнитивными процессами;
- рассмотрим влияние различных способов присваивания имен;
- рассмотрим влияние неудачных имен на ошибки в коде;
- рассмотрим, как структурировать имя переменной для наилучшего понимания.

В *части I* мы рассматривали различные когнитивные процессы, связанные с чтением кода, а также хранением информации в долговременной памяти и ее извлечением. Мы также рассматривали хранение информации в кратковременной памяти и обработку кода в рабочей памяти. В *части II* мы рассмотрели, как мы думаем о коде, какие ментальные модели формируются во время работы над кодом и как мы говорим о коде. В *части III* мы рассмотрим процесс написания кода, а не чтение кода или размышление о коде.

Главная цель этой главы состоит в изучении того, как нужно называть элементы кода, например переменные, классы или методы. Так как теперь мы достаточно знаем о том, как наш мозг обрабатывает код, мы можем углубиться в то, почему правильное присваивание имен так важно для понимания кода. Хорошие имена помогают долговременной памяти искать актуальную информацию, которую мы уже знаем о предметной области кода. С другой стороны, плохие имена заставляют вас делать неверные предположения о коде, которые приводят к заблуждениям.

Несмотря на то что имена элементов — это важная часть кода, научиться присваивать правильные имена очень сложно. Часто имена создаются при составлении плана решения задачи или при непосредственном решении задачи. Во время этого вы, скорее всего, испытываете высокую когнитивную нагрузку: ваша рабочая

память полностью задействована в создании ментальной модели, а также использовании ментальной модели при работе с кодом. В такой ситуации размышление над правильным именем переменной лишь вызовет дополнительную когнитивную нагрузку, которую ваш мозг хотел бы избежать. Следовательно, имеет смысл выбирать простое имя или общее обозначение элемента одной категории, чтобы не превысить объем рабочей памяти.

В этой главе мы рассмотрим не только важность, но и сложность присваивания имен. Ознакомившись с основами присваивания имен и когнитивной обработки, мы изучим влияние имен на два аспекта программирования. Во-первых, какие виды имен упрощают понимание кода. Во-вторых, как плохие имена влияют на появление ошибок в коде. Мы закончим главу рекомендациями по придумыванию хороших имен.

8.1. Почему присваивание имен так важно

Присвоить переменной хорошее имя очень сложно. Фил Картон (Phil Karlton), программист из Netspace, сказал, что в компьютерной науке есть только две главные проблемы: присваивание имен элементам и аннулирование кеша. И правда, многие программисты сталкиваются с проблемой, касающейся присваивания имен.

Очень сложно передать весь смысл класса или структуры данных с помощью одного точно выраженного слова. Дрор Фейтельсон (Dror Feitelson), профессор компьютерных наук в Еврейском университете в Иерусалиме, провел исследование для того, чтобы понять, насколько сложно придумать простое и понятное имя. В исследовании приняли участие примерно 350 человек, которых попросили выбрать имя в разных ситуациях в сфере программирования. В исследовании принимали участие как студенты, так и профессиональные программисты со средним опытом работы 6 лет. Участников попросили выбрать имена для переменных, констант и структур данных, а также для функций и их параметров. Исследование Фейтельсона подтвердило, что присваивать имена очень сложно или что, по крайней мере, сложно выбирать те имена, которые выбирают и другие. Вероятность того, что два разработчика выберут одно и то же имя, была крайне низкой. Для 47 элементов, которым нужно было присвоить имя (т. е. переменных, констант, структур данных, функций и параметров), вероятность того, что два человека присвоят одно и то же имя, составила только 7%.

Несмотря на то что присваивать имена сложно, выбор правильных имен для элементов программирования очень важен. Прежде чем мы подробно рассмотрим связь между присваиванием имени и когнитивными процессами в мозге, давайте узнаем, почему присваивание имен так важно.

8.1.1. Почему присваивание имени так важно

Под именами идентификаторов мы подразумеваем все элементы кодовой базы, которые были названы самим программистом. Имена идентификаторов включают в себя имена, которые мы присваиваем типу (класс, структура, делегат, интерфейс

или тип-перечисление), переменной, методу, функции, модулю, библиотеке или пространству имен. Существует четыре причины, почему имена так важны.

Имена составляют существенную часть кодовой базы

Первая причина, по которой имена переменных играют важную роль, заключается в том, что во многих кодовых базах имена составляют бóльшую часть того, что вы будете читать. Например, в исходном коде Eclipse, в котором примерно два миллиона строк кода, под идентификаторы выделено 33% токенов и 72% всех символов.[1]

Имена играют роль в обзорах кода

Имена не только часто встречаются в коде, о них очень много говорят программисты. Мильтиадис Алламанис (Miltiadis Allamanis), исследователь из Microsoft Research в Кембридже, провел исследование того, как часто имена идентификаторов упоминаются в обзорах кода. С этой целью Алламанис проанализировал более 170 обзоров кода, в которых было более 1000 комментариев. Он обнаружил, что в каждом четвертом обзоре был комментарий о присваивании имен. Замечания, касающиеся имен идентификаторов, встречаются в 9% всех обзоров кода.

Имена — это самая удобная форма документации

Хотя официальная документация по коду может содержать много справочной информации, имена переменных являются важным типом документации, т. к. они находятся непосредственно в кодовой базе. Как мы уже видели в предыдущих главах, сбор информации из разных источников может привести к увеличению когнитивной нагрузки. Следовательно, при чтении документации на код программисты стараются не обращаться к сторонним источникам информации. Логично предположить, что самой удобной формой документации для программистов станут комментарии и имена элементов кода.

Имена могут служить маячками

В предыдущих главах мы рассматривали маячки — части кода, которые помогают читателю понять суть программы. Имена переменных могут стать важными маячками, которые помогают читателям понять код.

8.1.2. Разные точки зрения на присваивание имен

Очень важно выбрать хорошее и понятное имя. Многие исследователи пытались выяснить, что именно делает имя переменной хорошим или плохим, и все исследователи придерживаются разных точек зрения. Однако перед тем, как мы рассмотрим эти точки зрения, давайте активируем долговременную память и посмотрим с помощью следующего упражнения, что вы думаете об именах переменных.

---

[1] «Concise and Consistent Naming», Флориан Дайссенбок и Мартин Пицка, **https://www.cqse.eu/fileadmin/content/news/publications/2005-concise-and-consistent-naming.pdf**.

УПРАЖНЕНИЕ 8.1. Как вы думаете, что делает имя идентификатора хорошим? Можете ли вы привести пример хорошего имени?

Что делает имя плохим? Плохое имя — это противоположность хорошего имени, или вы можете вспомнить какие-то особенные характеристики плохих имен, с которыми вы сталкивались раньше? Можете ли вы привести пример плохого имени?

Теперь, когда вы думаете над тем, что именно делает имя элемента хорошим именем, давайте рассмотрим три разные точки зрения на методы присваивания имени от исследователей, которые изучают присваивание имени.

Хорошее имя можно определить синтаксически

Некоторые люди считают, что существуют правила синтаксиса имен, которые обязательно нужно соблюдать. Например, Саймон Батлер (Simon Butler), кандидат наук из Открытого университета (Open University) в Великобритании, составил список проблем имен переменных (табл. 8.1).

Таблица 8.1. *Список Батлера — соглашение об именах*

Имя	Описание	Пример плохого имени
Использование верхнего регистра	В идентификаторах должны правильно использоваться прописные буквы	`page counter`
Следующие друг за другом знаки подчеркивания	Идентификаторы не должны содержать несколько следующих друг за другом знаков подчеркивания	`page__counter`
Словарные слова	Идентификаторы должны состоять из слов и содержать аббревиатуры только в том случае, если такой вид написания используется чаще полного имени	`pag_countr`
Количество слов	Идентификаторы должны содержать в себе от двух до четырех слов	`page_counter_ converted_and_ normalized_value`
Избыточные имена	Идентификаторы не должны состоять более чем из четырех слов	`page_counter_ converted_and_ normalized_value`
Короткие имена	Идентификаторы должны состоять минимум из восьми букв, за исключением c, d, e, g, i, in, inOut, j, k, m, n, o, out, t, x, y, z	`P, page`
Порядок объявления идентификаторов перечислимого типа	Перечислимые типы должны объявляться в алфавитном порядке, если нет явных причин не делать так	`CardValue = {ACE, EIGHT, FIVE, FOUR, JACK, KING...}`
Внешние знаки подчеркивания	Идентификаторы не должны начинаться или заканчиваться со знака подчеркивания	`__page_counter_`
Кодирование идентификатора	Информация о типе не должна кодироваться в имени идентификаторов, как это делается, например, в венгерской нотации	`Int_page_counter`

Таблица 8.1 (продолжение)

Имя	Описание	Пример плохого имени
Длинное имя идентификатора	По возможности избегайте использования длинных имен идентификаторов	`page_counter_converted_and_normalized_value`
Игнорирование соглашений об именах	В именах идентификаторов не должны одновременно использоваться буквы верхнего и нижнего регистра нестандартным способом	`Page_counter`
Числовое имя идентификатора	Идентификаторы не должны полностью состоять из числительных или цифр	`FIFTY`

Хотя в списке Батлера есть правила разных типов, большинство их них являются синтаксическими. Например, правило «Внешние знаки подчеркивания» говорит о том, что имена не должны начинаться или заканчиваться знаками подчеркивания. Правила Батлера также подразумевают запрет на использование венгерской нотации, согласно которой нужно было бы, например, строковую переменную, содержащую какое-либо имя, назвать `strName`.

И хотя эти правила могут показаться незначительными, в предыдущих главах мы увидели, что ненужная информация в коде может вызвать внешнюю когнитивную нагрузку и помешать пониманию кода. Советую вам всегда иметь под рукой синтаксические правила вроде тех, что указаны в табл. 8.1.

Многие языки программирования, естественно, имеют соглашения об именах переменных, например руководство PEP8 для Python, которое рекомендует «змеиный регистр» для имен переменных, а также соглашение об именах языка Java, согласно которому для имен переменных следует использовать «верблюжий регистр».

Имена во всей базе кода должны быть единообразны

Еще один способ присвоить хорошее имя — это единообразие. Алламанис, работы которого мы рассмотрели ранее в этой главе, также размышлял о хороших именах. Он заявлял, что важным аспектом хорошего имени является единообразие имен во всей кодовой базе.

Возражения против практики присваивания неединообразных имен подтверждают то, что мы уже знаем о когнитивистике. Если одно и то же слово используется для обозначения похожих элементов, то мозгу будет намного проще найти связанную информацию, которая хранится в долговременной памяти. Саймон частично согласен с Аллиманисом: в его списке есть правило, согласно которому нельзя использовать буквы верхнего регистра бессистемно.

УПРАЖНЕНИЕ 8.2. Выберите фрагмент кода, с которым вы недавно работали. Выпишите все имена переменных, которые есть в этом фрагменте кода. Теперь подумайте об именах переменных, опираясь на перечисленные точки зрения.

Понятны ли имена с точки зрения синтаксиса? Они состоят из слов? Используются ли они единообразно во всей базе кода?

Имя	Синтаксические ошибки	Единообразие в базе коде

8.1.3. Важно грамотно подбирать имена

Дон Лори (Dawn Lawrie), ведущий научный сотрудник Университета Джонса Хопкинса, долгое время изучала присваивание имен и исследовала тенденции в присваивании имен[2]. Отличаются ли способы присваивания имен от тех, которые использовались десять лет назад? И как имена в кодовой базе меняются на протяжении долгого периода времени?

Чтобы ответить на эти вопросы, Лори проанализировала около 186 версий 78 кодовых баз, написанных на C++, C, Fortran и Java. В сумме все эти версии составили более 48 миллионов строк кода и охватили период в 30 лет. Лори рассматривала как проприетарный код, так и программы с открытым исходным кодом, включая такие известные кодовые базы, как Apache, Eclipse, MySQL, gcc и Samba.

Для проведения анализа качества имен Лори исследовала два аспекта практики присваивания имен. Во-первых, она обращала внимание на то, разделяются ли внутри имени слова с помощью знака нижнего подчеркивания или с помощью букв верхнего регистра. Лори утверждает, что состоящие из двух слов имена понимать легче. Во-вторых, она смотрела на то, встречаются ли слова, используемые в имени переменной, в словарях — данное правило указано в списке Батлера.

Так как Лори изучала разные версии одной и той же базы кода, она смогла проанализировать то, как со временем менялась практика присваивания имен. Она посмотрела, как качество присваивания имен изменилось для всех 78 кодовых баз, и обнаружила, что в современном коде в именах идентификаторов словарные слова используются чаще, чем в старом коде. Она также заметила, что в современном коде слова в именах чаще разделяются знаком подчеркивания. Лори связывает улучшение практики присваивания имен с развитием программирования как дисциплины. Размер кодовой базы не связан с качеством, так что, когда дело касается качества имен переменных, большие базы не работают лучше (или хуже).

Лори не только сравнивала старые базы кода с новыми, но и просматривала историю версий одной базы кода: так она отмечала изменения в практике присваивания имен с течением времени. Она обнаружила, что по мере устаревания кода имена переменных не становятся лучше. Из этого Лори делает важный вывод: «качество идентификаторов проявляется на ранних этапах работы с кодом». Если вы беретесь

---

[2] «Quantifying Identifier Quality: AN. Analysis of Trends», Дон Лори, Генри Филд и Дэвид Бинкли, **http://www.cs.loyola.edu/~lawrie/papers/lawrieJese07.pdf**.

за новый проект, то обратите особое внимание на выбор удачных имен, т. к. способ, с помощью которого вы присваиваете имена элементам в самом начале, будет использоваться при присваивании имен даже на более поздних этапах работы.

Исследование востребованности тестов на материале тех демо-проектов, что размещены на Github показало похожий результат: новые пользователи репозитория просматривают существующие тестовые программы и изменяют их, а не читают прикрепленные методические указания[3]. Когда в репозитории есть подходящая тестовая программа, новые участники чувствуют необходимость также добавить тестовые программы, что не противоречит правилам форума.

Заключения о практике присваивания имен

- Современный код больше соответствует правилам присваивания имен.
- В рамках одной базы кода практика присваивания имен не изменяется.
- В отношении присваивания имен между большими и маленькими базами кода нет никакой разницы.

До этого мы рассматривали две точки зрения на присваивание имен. Эти точки зрения представлены в табл. 8.2.

Таблица 8.2. *Разные точки зрения на присваивание имен*

Исследователь	Точка зрения
Батлер	Синтаксически похожие имена
Алламанис	Имена должны быть единообразны во всей базе кода

Согласно Батлеру, при присваивании имени мы можем следовать основным синтаксическим принципам. С другой стороны, Алламанис не предлагает правил или указаний по отношению присваивания имен, однако придерживается позиции, согласно которой плохое, но отвечающее единообразию имя предпочтительнее хорошего, но несовместимого с единообразием. К сожалению, не существует единого способа обозначения имен в коде, однако сам факт того, что даже исследователи придерживаются разных точек зрения, указывает на то, что для каждого человека хорошее имя будет выглядеть по-разному.

8.2. Когнитивные аспекты присваивания имен

Теперь, когда мы рассмотрели, почему присваивание имен так важно, и какие на это есть точки зрения, давайте посмотрим на присваивание имен с точки зрения того, что мы знаем о познании.

---

[3] «Creating a Shared Understanding of Testing Culture on a Social Coding Site», Рафаэль Фам и др. (2013), **http://etc.leif.me/papers/Pham2013.pdf**.

8.2.1. Форматирование имен поддерживает кратковременную память

Учитывая наши знания о когнитивной обработке кода мозгом, мы можем заметить, что обе точки зрения имеют право на жизнь с точки зрения когнитивистики. Это показано в табл. 8.3. Наличие четких правил форматирования имен переменных помогает кратковременной памяти при чтении кода понимать имена элементов.

Таблица 8.3. *Разные точки зрения на присваивание имен и их связь с познанием*

Исследователь	Точка зрения	Соответствует познанию, потому что...
Алламанис	Имена должны быть единообразны во всей базе кода	Помогает разбивать код на чанки
Батлер	Синтаксически похожие имена	Снижает когнитивную нагрузку во время обработки имен

Алламанис рекомендует практиковать присваивание единообразных имен во всей базе кода. Такой подход имеет смысл, т. к. он помогает разбивать код на чанки. Если бы все имена были отформатированы по-разному, то вам пришлось бы потратить много сил и времени для того, чтобы найти значение каждого отдельного имени.

Точка зрения Батлера также соответствует тому, что мы знаем о когнитивной обработке. Он советует использовать синтаксически похожие имена, например не начинать имя со знака подчеркивания или единообразно применять буквы верхнего регистра. Придерживаясь подобных правил, вы не испытаете дополнительную когнитивную нагрузку при чтении имен, т. к. релевантная информация всегда будет представлена в едином стиле. Ограничение Батлера в четыре слова для одного имени хотя и кажется выбранным наугад, однако оно соответствует емкости рабочей памяти, которая на данный момент оценивается в пределах от двух до шести чанков.

Улучшите единообразие имен

Для улучшения единообразия имен кодовой базы Алламанис реализовал свой метод обнаружения несовместимых с единообразием имен в виде программы, названной Naturalize (**http://groups.inf.ed.ac.uk/naturalize/**). Данный инструмент использует машинное самообучение для обучения хорошим (единообразным) именам, взятым из какой-нибудь базы кода, а затем предлагает новые имена для локальных переменных, аргументов, полей, методов и типов В первом исследовании авторы Naturalize сгенерировали 18 пул-запросов в различных кодовых базах с предложениями по улучшению имен 14 из этих запросов были приняты, что говорит о том, что результатам работы программы можно доверять К сожалению, сейчас Naturalize работает только для кода на Java.

В своей статье авторы Naturalize рассказывают чудесную историю о том, как они использовали свой инструмент для пул-запроса для фреймворка JUnit. Запрос не был принят, т. к., по словам разработчиков, предложенные изменения не соответствовали кодовой базе. Программа указала разработчикам на все остальные места, где приня-

тое в кодовой базе соглашение было нарушено, и предложила свой вариант. Их собственное соглашение нарушалось настолько часто, что для Naturalize неправильная версия выглядела более естественной!

8.2.2. Понятные имена лучше закрепляются в долговременной памяти

Две точки зрения на присваивание имен, которые мы рассматривали до этого, хотя и различаются, но имеют общие черты. Оба метода являются синтаксическими или статистическими, и для оценки качества имен по критериям обеих моделей может использоваться компьютерная программа. Как мы видели, модель Алламадиса тоже реализована программно.

Однако присваивание имен — это нечто большее, чем просто выбор правильного синтаксиса для имени переменной. Слова, которые мы выбираем, имеют большое значение. Ранее мы узнали, что при обдумывании кода наша рабочая память обрабатывает два типа информации. Вы можете увидеть это на рис. 8.1. Сначала имя переменной обрабатывает сенсорная память, а затем имя переходит в кратковременную память. Как мы знаем, кратковременная память обладает ограниченной емкостью, поэтому имя переменной делится на несколько частей — отдельные слова. Чем систематичнее отформатированы имена, тем больше вероятность того, что кратковременная память сможет опознать отдельные части имени. Например, чтобы найти и понять отдельные составляющие в имени `mncntrayg`, могут потребоваться значительные усилия. Другой пример: очень легко понять, для чего нужно, например, имя `name_counter_average`. Несмотря на то что это имя почти в два раза больше первого, для его чтения требуется намного меньше умственных усилий.

При обработке имен переменных не только кратковременная память отправляет информацию в рабочую память. Ваша долговременная память ищет связанную ин-

Рис. 8.1. Когда вы читаете имя, оно разбивается на отдельные чанки, а затем переходит в рабочую память В то же время в долговременной памяти происходит поиск информации, соотносящейся с частями имени переменной
Связанная информация переходит из долговременной памяти в рабочую память

формацию и также отправляет ее в рабочую память. Для этого когнитивного процесса важен выбор слов в имени идентификатора. Если вы будете использовать правильный концепт предметной области для имени переменной или класса, то вы поможете читателю вашего кода найти соответствующую информацию в его долговременной памяти.

8.2.3. Полезная информация в именах переменных

Как показано на рис. 8.2, в именах элементов могут содержаться три типа знаний, которые помогут вам быстро понять незнакомое имя:

1. Имена могут поддерживать ваше понимание предметной области кода. Например, для слова customer (покупатель) в вашей долговременной памяти найдется множество ассоциаций. Покупатель, вероятно, покупает продукт, у него должно быть имя, адрес и т. д.
2. Имя может поддерживать ваше представление о программировании. Например, концепт программирования «дерево» также разблокирует информацию в долговременной памяти. У дерева есть корень, дерево можно обойти, преобразовать в список и т. д.
3. В некоторых случаях в имени переменной содержится информация о соглашениях, известных вашей долговременной памяти. Например, переменная с именем `j` напомнит вам вложенный цикл, в котором `j` является счетчиком внутреннего цикла.

Рис. 8.2. Три типа знаний, которые хранятся в вашей долговременной памяти и могут присутствовать в именах переменных, могут помочь вам понимать имена элементов. знание предметной области (например, покупатель и поставка товаров), концепции программирования (список, дерево, хеш-таблица) и соглашения (например, `i` и `j` будут счетчиками циклов, а `n` и `m` будут размерами)

Выбирать имена станет намного проще, если принимать во внимание, как имя переменной будет поддерживать долговременную и кратковременную память будущего читателя.

УПРАЖНЕНИЕ 8.3. Выберите фрагмент малознакомого исходного кода. Это может быть фрагмент кода, с которым вы работали какое-то время назад, или фрагмент кода, написанный другим программистом в кодовой базе, с которой вы тоже работали.

Просмотрите код и выпишите имена всех элементов: переменных, методов и классов. Подумайте о том, как каждое из имен поддерживает ваши когнитивные процессы:

- Форматирование имени поддерживает вашу кратковременную память? Можно ли как-то изменить имя, чтобы отдельные части стали понятнее?
- Поддерживает ли имя вашу долговременную память в понимании предметной области? Можно ли как-то изменить имя, чтобы оно стало понятнее?
- Поддерживает ли имя вашу долговременную память в понимании концепций программирования? Можно ли как-то изменить имя, чтобы оно стало понятнее?
- Поддерживает ли имя вашу долговременную память в понимании, потому что имя основано на соглашении?

8.2.4. Когда стоит оценивать качество имен

Мы уже знаем, что из-за когнитивных процессов, связанных с программированием, присваивание имени становится нелегким делом. Когда вы решаете какую-то задачу, то, скорее всего, вы испытываете большую когнитивную нагрузку. Может, когнитивная нагрузка при решении задачи была такая большая, что вы не смогли подобрать хорошее имя для переменной. Думаю, все мы все хоть раз писали сложный код, в котором называли переменную `foo`, потому что мы хотели решить задачу, а не думать над именем переменной. К тому же понимание того, как надо назвать переменную, может прийти к вам позже в процессе программирования.

Следовательно, кодирование — это не самый удачный момент, когда нужно думать о том, как лучше назвать переменную. Думать о том, какое имя лучше всего присвоить, нужно вне процесса кодирования. Проверка кода — более подходящий момент подумать о качестве имен идентификаторов. Упражнение 8.4 послужит вам чек-листом для имен. Вы можете использовать его при проверке кода, обращая особое внимание на имена в коде.

УПРАЖНЕНИЕ 8.4. Перед началом проверки кода перечислите все имена идентификаторов, которые есть в измененном коде. Запишите эти имена вне кода, например в отдельном документе. Для каждого из имен ответьте на следующие вопросы:

- Ничего не зная о коде, вы понимаете значение имени? Например, знаете ли вы значение слов, из которых состоит имя?
- Имя кажется двусмысленным?
- В именах используются аббревиатуры, которые могут сбить с толку?
- Есть ли похожие имена? Если да, то относятся ли они к похожим элементам в коде?

8.3. Какие типы имен проще всего понимать

До сих пор мы рассуждали о том, почему важно дать элементу хорошее имя, а также рассматривали влияние различных имен на когнитивные процессы. Теперь давайте подробно рассмотрим варианты форматирования имен.

8.3.1. Использовать аббревиатуры или нет?

До этого мы познакомились с точкой зрения, что имена должны составлять комбинацию из словарных слов. Хотя использование полных слов кажется правильным выбором, давайте рассмотрим доказательства того, что имена, состоящие из полных слов, действительно легче понимать.

Йоханнес Хоффмайстер (Johannes Hofmeister), исследователь из Университета Пассау в Германии, провел исследование с участием 72 опытных программистов на C#. Программистам нужно было искать ошибки во фрагментах кода на C#. Хоффмайстеру было интересно, что при поиске ошибок важнее: значение или вид имен элементов. Программистам представили три типа программ: в одной программе имена идентификаторов были буквами, во второй программе — аббревиатурами, а в третьей программе имена были словами. Хоффмайстер попросил участников найти синтаксические и семантические ошибки. Затем он измерил время, потребовавшееся участникам для нахождения всех ошибок в предложенных программах.

Участники находили на 19% больше ошибок в минуту в программах, в которых имена были представлены словами, а не буквами и аббревиатурами. Особой разницы в скорости нахождения ошибок в программах с буквами и аббревиатурами не было.

Хотя другие исследования подтверждают факт того, что слова в именах переменных облегчают понимание кода, использование длинных имен переменных может иметь и негативные последствия[4]. Лори, работу который мы рассмотрели ранее в этой главе, провела исследование, в котором приняли участие 120 опытных программистов со средним стажем работы семь с половиной лет. Опытным программистам нужно было понять и запомнить исходный код с тремя разными видами идентификаторов: словами, аббревиатурами и буквами.

Участникам показали метод, в котором использовался один из трех стилей идентификаторов. Затем код убрали и попросили участников объяснить код простыми словами и вспомнить имена переменных, которые использовались в коде. В отличие от Хоффмайстера, Лори исследовала то, насколько слова участников соответствовали реальной функциональности кода. Она оценивала ответы участников по шкале от 1 до 5.

Полученные Лори результаты были похожи на результаты Хоффмайстера: имена элементов, состоящие из целых слов, воспринимались легче, чем имена с буквами

---

[4] «Effective Identifier Names for Comprehension and Memory», Дон Лори и др. (2007), **https://www.researchgate.net/publication/220245890_Effective_identifier_names_for_comprehension_and_memory**.

или аббревиатурами. Пересказ функций кода, в котором использовались имена элементов с целыми словами, был оценен на балл выше, чем пересказы кода с однобуквенными идентификаторами.

Исследование также показало и негативную сторону использования имен с полными словами. Проведя исследование повторно и изучив полученные результаты, Лори заметила, что для запоминания длинных имен переменных требуется много времени и усилий. Проблема запоминания таких имен состоит в количестве содержащихся в имени слогов, а не в его длине. Это понятно с когнитивной точки зрения: длинные имена занимают больше места в кратковременной памяти, и, скорее всего, длинное имя будет разбиваться на слоги. Для того чтобы выбрать хорошее имя переменной, вам нужно уметь находить баланс между понятными словами, которые улучшат способность читателя понимать код и искать ошибки, и краткостью аббревиатур, которая улучшает способность читателя запоминать.

Основываясь на своем исследовании, Лори советует с осторожностью использовать соглашение об именах, которое включает в себя префикс или суффикс элемента. Все это требует тщательной оценки, чтобы убедиться, что добавленная информация перевешивает добавленные затраты на имена, которые сложно запомнить.

> **БУДЬТЕ ОСТОРОЖНЫ С ПРЕФИКСАМИ И СУФФИКСАМИ**
>
> Лори советует с осторожностью использовать соглашение об именах, которое включает в себя префикс или суффикс элемента

Однобуквенные имена переменных

Мы знаем, что целые слова являются лучшими именами элементов в сравнении с аббревиатурами или буквами, как с точки зрения поиска ошибок в коде, так и с точки зрения запоминания имени. Однако на практике широко применяются и одинарные буквы. Гал Беньямини (Gal Beniamini), исследователь из Еврейского университета в Иерусалиме, изучил то, как одинарные буквы используются на C, Java, JavaScript, PHP и Perl. Для каждого из этих пяти языков программирования Беньямини выгрузил с GitHub 200 самых популярных проектов — это более 16 Гбайт исходного кода.

Полученные Беньямини результаты показали, что разные языки программирования придерживаются разных соглашений об однобуквенных именах переменных. Например, для Perl наиболее часто используются три буквы: v, i и j, в то время как на JavaScript в качестве имени распространены буквы i, e и d. На рис. 8.3 представлено использование всех 26 букв латинского алфавита на пяти языках программирования, которые исследовал Беньямини.

Беньямини смотрел не только на частоту использования однобуквенных имен переменных, но также и изучал то, какие ассоциации возникают у программистов с буквами. Например, при использовании буквы i многие программисты думают об итераторе цикла, а при использовании букв x и y они представляют координаты на плоскости. Но с чем ассоциируются буквы b, f, s или t? Ассоциируют ли их с чем-то общеизвестным? Знание ассоциаций, которые программисты делают по отноше-

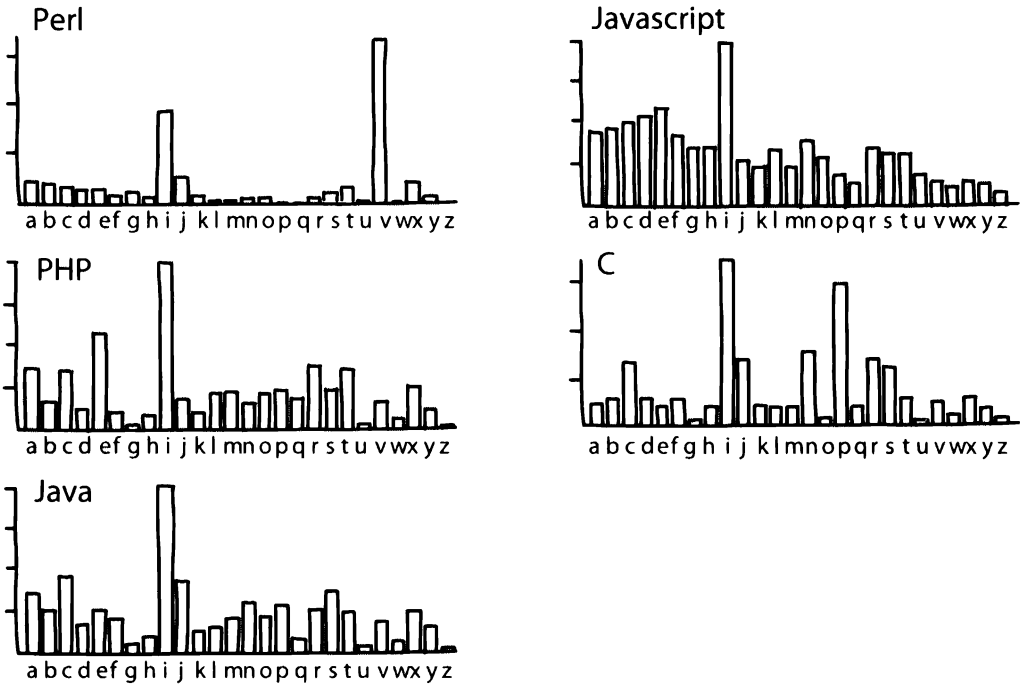


Рис. 8.3. Однобуквенные имена переменных, которые используются в пяти языках программирования, проанализированных Беньямини

нию к именам переменным, помогает предотвратить ошибки, а также понять, почему у других людей код вызывает замешательство.

Для того чтобы понять, какие типы программисты ассоциируют с именами переменных, Беньямини опросил 96 опытных программистов. Он просил программистов называть один или несколько типов элементов, с которыми у них ассоциируются однобуквенные переменные. Как вы можете видеть на рис. 8.4, люди в данном случае не придерживаются единого мнения. Исключениями являются буква s, которую многие программисты ассоциируют со строкой, буква c, которую принимают за символ, а также буквы i, j, k и n, которые ассоциируют с целыми числами. Остальные буквы подходят под любой тип переменной.

Слегка удивляет тот факт, что буквы d, e, f, r и t чаще всего ассоциируются с числами с плавающей точкой, а буквы x, y и z — как с целыми числами, так и с числами с плавающей точкой. Это может означать только то, что когда они используются как координаты, они могут являться как целыми числами, так и числами с плавающей точкой.

Результаты Беньямини показали, что мы не можем принимать предположения других людей на веру. Мы можем думать, что определенная буква точно отвечает за определенный тип элемента и помогает читателю понимать код, но такое случается очень редко. Следовательно, если вы хотите, чтобы ваш код понимали, то используйте в качестве имени переменной слова или соглашение об именах.

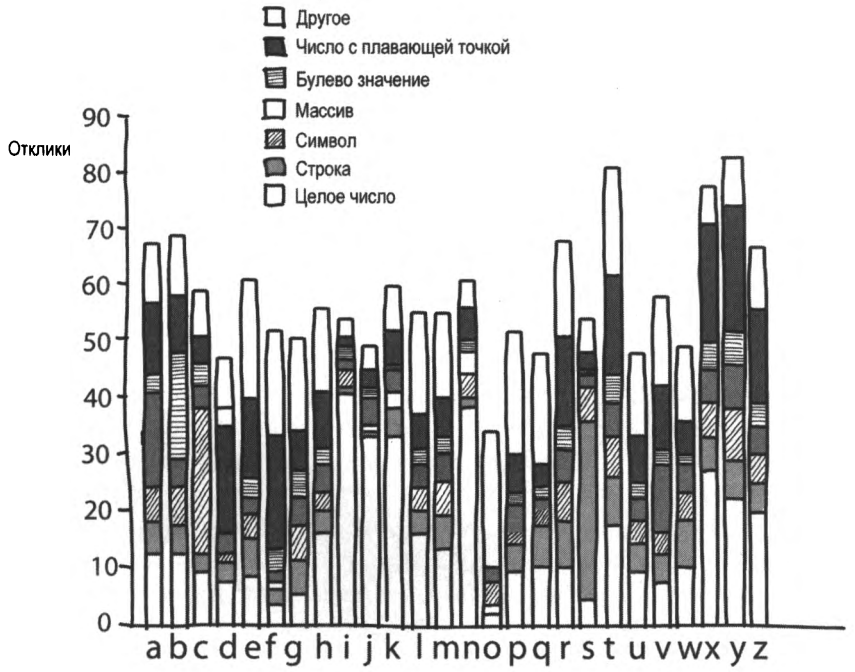


Рис. 8.4. Типы, связанные с однобуквенными именами

УПРАЖНЕНИЕ 8.5. Запишите типы, которые ассоциируются с каждым из 26 однобуквенных имен переменных. Заполните таблицу. Сравните свои записи с записями ваших коллег. Есть ли такие буквы, по которым ваши предположения и предположения ваших коллег не совпадают? Можете ли вы найти место в кодовой базе, где эти буквы используются как переменные?

Буква	Тип	Буква	Тип	Буква	Тип
a		j		s	
b		k		t	
c		l		u	
d		m		v	
e		n		w	
f		o		x	
g		p		y	
h		q		z	
i		r			

8.3.2. Змеиный или верблюжий регистры?

Хотя у большинства языков программирования есть руководство по стилю оформления, в котором описывается также и то, как должны выглядеть имена переменных, единого мнения нет. Как хорошо известно, все языки, относящиеся к семейству C, в том числе C, C++, C# и Java, используют верблюжий регистр: первая буква любой переменной написана нижним регистром, а каждое новое слово в имени переменной начинается с буквы верхнего регистра, например `customerPrice` или `nameLength`. С другой стороны, в языке Python используется соглашение под названием змеиный регистр: в таком случае слова в именах идентификаторов разделяются знаками подчеркиваниями, например `customer_price` или `name_length`.

Дэйв Бинкли (Dave Binkley), профессор компьютерной науки в Университете Лойолы в Мэриленде, провел исследование, в котором он изучал различия в понимании переменных, записанных с помощью верблюжьего и змеиного регистров[5]. Бинкли хотел знать, влияет ли выбор стиля имени переменной на скорость и точность понимания кода. В исследовании приняло участие 135 человек, из которых не все были программистами. Участникам показали предложение, в котором описывалась переменная, например «Преобразовывает список в таблицу». После чего участникам предлагалось четыре варианта множественного выбора, один из которых соответствовал этому предложению, например такие: `extendListAsTable`, `expandAliasTable`, `expandAliasTitle` и `expandAliasTable`.

Результаты исследования показывают, что использование верблюжьего регистра приводит к большей точности как среди программистов, так и среди непрограммистов. Вероятность выбора правильного варианта имени, написанного верблюжьим регистром, была выше на 51,5%. Однако цена такой высокой точности — скорость. Участником требовалось на полсекунды больше, чтобы найти имена элементов, написанные верблюжьим регистром.

Бинкли также изучил влияние обучения программированию на эффективность испытуемых. В данном случае он сравнивал результаты людей без обучения и людей с многолетним обучением. Участники, прошедшие обучение, в основном учились с использованием верблюжьего регистра.

Сравнивая людей с разным опытом, Бинкли обнаружил, что программисты, проходившие обучение с использованием верблюжьего регистра, быстрее находили правильные имена элементов, написанные верблюжьим регистром. Видимо, обучение одному стилю идентификаторов отрицательно сказывается на эффективности в случае других стилей. Результаты исследования показали, что участники, учившиеся с использованием верблюжьего регистра, медленнее работали с переменными, имена которых были написаны змеиным регистром. При этом их результаты были хуже, чем результаты участников, которые не обучались совсем.

Учитывая наши знания о когнитивной обработке, данные результаты не являются для нас чем-то удивительным. Если люди часто практикуются в использовании

---

[5] «Relating Identifier Naming Flaws and Code Quality: An Empirical Study», Саймон Батлер (2009), **http://oro.open .ac.uk/17007/1/butler09wcreshort_latest.pdf**.

имен, написанных верблюжьим регистром, то им будет проще разбивать имена на чанки и понимать их смысл.

Конечно, если вы работаете с базой кода, в которой используется змеиный регистр, то было бы неразумно менять все имена переменных в соответствии с результатами данного исследования. Единообразие базы кода тоже является важным аспектом. Однако если у вас есть возможность выбирать соглашение об именах, то вы можете выбрать верблюжий регистр.

8.4. Влияние имен на ошибки кода

До этого мы изучали, почему присваивание имени имеет большое значение и какие виды имен проще понять. Однако неправильные способы присваивания имени также могут стать причиной возникновения ошибок.

8.4.1. В коде с некачественными именами больше ошибок

Саймон Батлер, чью работу мы рассмотрели ранее, проанализировал взаимосвязь между плохими именами и ошибками. В 2009 году[6] он провел исследование, в котором изучил связь между плохими именами и плохим кодом. В своем исследовании Батлер использовал репозитории проектов с открытым исходным кодом, написанные на Java, включая Tomcat и Hibernate.

Батлер создал программу для извлечения имен переменных из кода, написанного на Java, а также для поиска нарушений правил присваивания имен. Так Батлер обнаружил плохие имена в восьми проектах. Как объяснялось в *разд. 8.1*, Батлер рассматривал как структуру имен, например использование двух знаков подчеркивания подряд, так и компоненты имен, например являются ли они словарными словами.

Затем Батлер сопоставил расположение плохих имен элементов с расположением ошибок в коде. Для нахождения ошибок он использовал программу FindBugs, которая с помощью статического анализа обнаруживает потенциальное расположение ошибок. Интересно то, что в исследовании Батлера была обнаружена статистически значимая связь между проблемами присваивания имен и качеством кода. Результаты исследования говорят о том, что плохое имя может указывать на код, который будет полностью неправильным, а не просто трудным для чтения, понимания и сопровождения.

Конечно, связь между расположением ошибок и плохих имен не является причинно-следственной связью. Может быть, причиной ошибок и плохих имен является неопытный или невнимательный программист. Также ошибки могут возникать в тех местах, где программист сталкивается со сложной проблемой. Такие трудности могут быть связаны с неудачными именами и другим способом. Как мы уже

---

[6] «Relating Identifier Naming Flaws and Code Quality: An Empirical Study», Саймон Батлер (2009), **http://oro.open.ac.uk/17007/1/butler09wcreshort_latest.pdf**.

обсуждали ранее, при написании кода программист испытывает высокую когнитивную нагрузку, потому что он выполняет сложную задачу. Может случиться так, что предметная область кода очень сложна, поэтому придумать хорошее и понятное имя трудно, и сложность выбора хорошего имени приводит к замешательству и ошибке.

Итак, хотя выявление проблем присваивания имен не обязательно предотвратит ошибки в коде, локализация плохих имен может помочь найти места, где код можно улучшить и предотвратить ошибки. Это служит еще одной причиной для более внимательного поиска неудачных имен в коде: улучшив имена элементов, вы можете косвенно уменьшить количество ошибок или по крайней мере сократить время исправления ошибок, т. к. чем лучше имена, тем понятнее код.

8.5. Как выбирать хорошие имена

Мы уже знаем, что выбор неудачного имени приводит к серьезным последствиям: снижается уровень понимания кода и повышается вероятность допустить ошибку. Фейтельсон, работу которого мы рассматривали ранее, изучал также то, как программисты выбирают и присваивают хорошие имена[7].

8.5.1. Шаблоны имен

В проводимом Фейтельсоном опросе по выбору переменных он заметил, что, хотя программисты редко выбирают одинаковые имена для переменных, они могут понимать имена, выбранные другими программистами. Когда в его исследовании один программист выбирал имя элемента, практически все программисты понимали выбранное имя. Причина этого кажущегося противоречия в том, что разработчики использовали то, что Фейтельсон называет *шаблонами имен*.

Шаблоны имен — это типовые комбинации элементов имени переменной. Например, когда надо было найти имя для максимальной месячной прибыли, были выбраны все имена, показанные в табл. 8.4. Все имена в таблице нормализированы, так что max может представлять собой max, maximum, a benefit может быть benefits. Имена в таблице приведены в порядке уменьшения популярности.

Это помогает понять, почему вероятность выбора одного и того же имени двумя разработчиками в исследовании Фейтельсона была настолько мала. Расхождение в именах переменных в основном было у программистов, которые использовали разные шаблоны имен переменных.

И хотя все эти имена концептуально представляют одно и то же, различия в их стиле очень заметны. Не все программисты, которые участвовали в исследовании Фейтельсона, работали с одной базой кода; однако даже при работе с одной базой кода становится ясно, что программисты будут выбирать разные шаблоны имен. Учитывая наши знания о когнитивистике и долговременной памяти, мы понимаем, что не стоит использовать разные шаблоны имен в пределах одной базы кода.

---

[7] «How Developers Choose Names», Дрор Фейтельсон, **https://www.cs.huji.ac.il/~feit/papers/Names20TSE.pdf**.

Таблица 8.4. *От наиболее используемых к менее используемым формам имен переменных*

max_benefit
max_benefit_per_month
max_benefit_num
max_monthly_benefit
benefits
benefits_per_month
max_num_of_benefit
max_month_benefit
benefit_max_num
max_number_of_benefit
max_benefit_amount
max_acc_benefit
max_allowed_benefit
monthly_benefit_limit

Во-первых, говоря терминами когнитивной нагрузки, поиск релевантного концепта (в данном случае — benefit) в имени переменной и разных местах имени вызовет ненужную внешнюю когнитивную нагрузку. Умственная энергия, потраченная на поиск нужной концепции, не будет использована для понимания имен. В этой главе мы уже видели, что людей можно научить искать переменные в определенном стиле (пример с верблюжьим и змеиным регистрами). И хотя исследования шаблонов имен не проводились, люди, вероятно, будут лучше понимать переменные, написанные по определенному шаблону, если они будут часто их использовать.

Во-вторых, если имена переменных похожи, то лучше использовать один шаблон имен переменных: так вы упростите поиск подходящей информации в долговременной памяти. Например, имя `max_benefit_amount` может напомнить вам о коде, который вы писали для расчета максимальной суммы процентов (maximum interest amount), если у переменной было имя `max_interest_amount`. Вашей долговременной памяти трудно будет найти похожий код, если переменная называлась `interest_maximum`, даже если вычисления были похожи.

Так как похожие шаблоны лучше всего поддерживают вашу рабочую и долговременную память, желательно создать связку разных шаблонов имен переменных, которые вы будете использовать в каждой базе кода. Если вы начинаете новый проект, то согласование шаблонов может стать хорошим шагом в этом направлении. Если вы работаете с готовой базой кода, то вы можете создать или извлечь список имен переменных из базы, посмотреть, какие шаблоны имен уже используются, и подумать об их дальнейшем улучшении.

УПРАЖНЕНИЕ 8.6. Создайте список имен переменных и функций/методов части вашей базы кода. Это может быть один класс, весь код в одном файле или весь код, связанный с определенной функциональностью. С помощью следующей таблицы проверьте, к какому шаблону относится каждое из имен. В именах из таблицы X представляет собой количество, например проценты по НДС, а Y — фильтр количества, например «в месяц для одного клиента».

Обсудите результаты со всей командой. Какие шаблоны используются чаще всего? Можно ли записать имена переменных с помощью другого шаблона, чтобы элемент соответствовал единообразию базы кода?

Шаблон	Переменная	Функция/метод
max_X		
max_X_per_Y		
max_X_num		
X		
X_per_Y		
max_num_of_X		
max_Y_X		
X_max_num		
max_number_of_X		
max_X_amount		
max_acc_X		
max_allowed_X		
Y_X_limit		
max_X		
Прочие		

8.5.2. Трехступенчатая модель Фейтельсона для хороших имен переменных

Мы уже видели, что программисты часто используют много разных шаблонов имен для одних и тех же объектов, тогда как использование одинаковых шаблонов может упростить понимание кода. Основываясь на этих выводах, Фейтельсон создал трехступенчатую модель, которая поможет программистам находить хорошие имена:

1. Выберите концепцию, которая должна быть в имени.
2. Выберите слова для представления каждой концепции.
3. Создайте имя с использованием этих слов.

Трехступенчатая модель во всех деталях

Давайте во всех деталях рассмотрим три ступени модели. Первая ступень, выбор концепции, зависит от предметной области, и выбор концепции может стать самым важным решением при присваивании имени. Особое внимание нужно уделять цели имени, которое должно отражать то, что делает объект и для чего он нужен. Когда вам кажется, что для объяснения имени нужно писать комментарий, или, когда вы встречаетесь с комментарием, который расположен в месте, близком к имени элемента, то формулировку комментария можно вставить в имя переменной. В некоторых случаях в имя переменной можно вставить указание того, что это за информация: например что длина находится в горизонтальной или вертикальной плоскости, вес измеряется в килограммах или что в буфере хранятся вводимые пользователем данные, которые кажутся небезопасными. Иногда при преобразовании данных мы можем использовать новое имя. Например, после того как входной параметр был проверен, его можно сохранить в другой переменной с именем, в котором указано, что данные безопасны.

Вторая ступень модели — это выбор слов для представления каждой концепции. Часто самым очевидным выбором является выбор правильных слов или одного слова, т. к. оно используется в предметной области кода или во всей базе кода. Однако в своих исследованиях Фейтельсон заметил, что было несколько случаев, когда участники предлагали множество противоположных вариантов. Такое разнообразие может вызвать проблему, когда разработчики не понимают, несут ли синонимы один и тот же смысл или между ними есть какие-то различия. *Лексика* проекта, в которой выделены все важные определения и указаны возможные синонимы, поможет программистам подобрать хорошее имя.

Фейтельсон также отмечает, что порядок выполнения ступеней можно не соблюдать. Иногда вы можете выбрать слова, из которых будет состоять имя переменной, без учета выбранной концепции. В таких случаях важно все же выбрать концепции.

Третья ступень модели — это создание имени с помощью выбранных слов, что приводит к использованию определенного шаблона имен. Как вы уже знаете, при выборе имени важно, чтобы оно соответствовало единообразию базы кода. Единообразие имен кода поможет вам найти важные элементы в имени элемента, а также упростит построение связи одного имени с другими именами в коде. Фейтельсон также советует использовать шаблоны так, чтобы они соответствовали естественному языку, на котором написано определение имени. Например, на английском вы скорее скажете the maximum number of points (максимальное количество баллов), а не the point maximum (максимум баллов). Следовательно, вы предпочтете имя `max_points`, а не `points_max`. Есть еще один способ сделать имена переменных более «естественными» — использование предлога, например `indexOf` или `elementAt`.

Успех трехступенчатой модели Фейтельсона

После создания трехступенчатой модели Фейтельсон провел второе исследование, в котором приняли участие 100 человек.

Новым участникам объяснили модель и показали ее использование на примере. После участникам дали те же имена, которые давались участникам, принимавшим участие в первом исследовании Фейтельсона. Затем двух независимых судей попросили сравнить пары двух имен: одно имя было из первого исследования, участники которого не были знакомы с моделью Фейтельсона, а второе имя из эксперимента, участники которого умели работать с моделью. Судьи не знали, какое имя из какого исследования было взято.

Результаты судей показали, что имена, выбранные участниками, которые использовали модель, оценивались выше, чем имена, выбранные в первом эксперименте. Соотношение оценок имен составляло два к одному. Следовательно, использование трехступенчатой модели Фейтельсона приводит к присваиванию хороших имен.

Выводы

- Существует множество точек зрения на то, что именно делает имя хорошим именем. Одни считают, что нужно придерживаться синтаксических правил, например использовать верблюжий регистр, а другие считают, что особое внимание нужно уделять единообразию базы кода.
- При прочих равных условиях имена переменных, написанные верблюжьим регистром, запомнить легче, чем имена переменных, написанные змеиным регистром. Записанные змеиным регистром имена переменных находятся в коде быстрее, чем переменные, записанные верблюжьим регистром.
- Если в коде встречаются неправильные имена, то, скорее всего, в таком коде будут и ошибки. Однако неправильные имена не всегда являются причиной ошибок.
- Существует множество шаблонов имен, которые используются для создания имен переменных, но для успешного понимания кода лучше ограничиться небольшим набором шаблонов.
- Трехступенчатая модель Фейтельсона (какие концепции нужно применять, какие слова использовать и как их сочетать) может помочь придумать имена высшего качества.

9 Боремся с плохим кодом и когнитивной нагрузкой. Две концепции

В этой главе:

- вы узнаете о связи между запахами кода и когнитивными процессами, особенно когнитивной нагрузкой;
- изучите связь между плохими именами и когнитивной нагрузкой.

Будучи опытным программистом, я уверена, что вы сталкивались с кодом, который было очень просто читать, а также с кодом, на понимание которого вам пришлось потратить много сил и времени. Причину, по которой код может быть сложен для понимания, мы рассматривали в предыдущих главах, когда обсуждали кратковременную, долговременную и рабочую память: вы просто испытываете слишком высокую когнитивную нагрузку. Когнитивная нагрузка возникает в тот момент, когда ваша рабочая память перегружается и мозг становится не в состоянии обрабатывать поступающую информацию. В предыдущих главах мы подробно рассмотрели то, как нужно читать код. Мы знаем, что иногда для более простого и легкого чтения кода нам нужно расширить наши знания о синтаксисе, концепциях или предметной области.

В этой главе мы исследуем то, что известно о написании кода с когнитивной точки зрения. Мы узнаем, какой код вызывает наибольшую когнитивную нагрузку, а также то, как улучшить код для более простой работы с ним. Мы рассмотрим две причины, по которым код может вызывать когнитивную нагрузку. Во-первых, код бывает трудно понять, потому что у него запутанная структура. Во-вторых, код может приводить в замешательство из-за того, что его «начинка» непонятна для пользователя. Выяснив, что именно делает код тяжелым для чтения, вы сможете научиться писать код, который будет легче понимать и сопровождать, а это значит, что для чтения и внедрения кода вашей команде (и вам) не нужно будет затрачивать много сил и энергии, и количество совершаемых ошибок уменьшится.

9.1. Почему код с запахами кода создает большую когнитивную нагрузку

В этой главе мы рассмотрим, как писать код, который не будет приводить других пользователей в замешательство. Говоря другими словами, мы научимся писать код, который не будет вызывать большую когнитивную нагрузку. Первая концепция, которая станет примером того, почему код может приводить в замешательство, — это идея *запахов кода*, т. е. частей кода, структуру которых можно сделать лучше. (Запахи кода были придуманы Мартином Фаулером (Martin Fowler). Впервые он написал о них в своей книге «*Рефакторинг: улучшение проекта существующего кода*» [СПб.: «Диалектика», 2019] в 1999 году.) Примером запахов кода может послужить слишком длинный метод или слишком сложный оператор переключения.

Возможно, вы уже сталкивались с запахами кода. Если нет, то в следующем разделе дается краткая информация о запахах кода. После краткого описания мы подробно рассмотрим запахи кода и их связь с когнитивными процессами, особенно с когнитивной нагрузкой. Выражение «Этот класс слишком большой» может быть полезным, но откуда мы знаем, насколько большой «слишком большой» и от чего это зависит.

9.1.1. Краткая информации о запахах кода

Фаулер описывает перечень запахов кода вместе со стратегиями их устранения, которые он назвал *рефакторингом*. Примеры запахов кода — это длинные методы, классы, которые пытаются быть многофункциональными, или сложные операторы переключения. Как мы видели ранее, термин «рефакторинг» стал употребляться вне контекста запахов кода, потому что рефакторинг может улучшать в целом весь код, а не только устранять запахи кода. Например, замена цикла генератором списков большинством программистов принимается за рефакторинг; при этом в цикле необязательно должны быть запахи кода.

В своей книге Фаулер описывает 22 запаха кода, которые мы свели в табл. 9.1. Хотя Фаулер в своей книге не разграничивает их, однако все запахи можно разделить на разные уровни, что также показано в табл. 9.1. Некоторые запахи кода могут относиться к одному методу, например «Длинный метод», а другие могут относиться ко всей кодовой базе, например «Комментарии». В следующем разделе мы рассмотрим запахи кода каждого уровня.

Таблица 9.1. *Краткий обзор запахов кода по Фаулеру и уровней, к которым они относятся*

Запах кода	Объяснение	Уровень
Длинный метод	Метод не должен состоять из большого количества строк кода, выполняющих разные вычисления	Уровень метода
Длинный список параметров	В методе не должно быть слишком много параметров	Уровень метода

Таблица 9.1 *(продолжение)*

Запах кода	Объяснение	Уровень
Операторы переключения	В коде не должно быть больших операторов переключения, для улучшения кода можно использовать полиморфизм	Уровень метода
Альтернативные классы с разными интерфейсами	В одном коде не должно быть двух классов, которые кажутся разными, но на самом деле имеют похожие поля и методы	Уровень класса
Одержимость элементарными типами	Избегайте злоупотребления элементарными типами в классе	Уровень класса
Неполнота библиотечного класса	Методы нужно добавлять в библиотечный класс, а не в случайные классы	Уровень класса
Большой класс	Класс не должен состоять из большого количества полей и методов, из-за чего становится непонятно, что делает класс	Уровень класса
Ленивый класс	Класс должен быть полезен для кода и выполнять какую-то функцию	Уровень класса
Класс данных	В классах не должны находиться только данные, в них также должны находиться методы	Уровень класса
Временное поле	В классах не должны находиться ненужные временные поля	Уровень класса
Группы данных	Данные, которые часто используются вместе друг с другом, связаны и должны находиться в одном классе или структуре	Уровень класса
Расходящиеся модификации	Обычно изменения являются локальными, желательно в пределах одного класса Если вам нужно исправить код во множестве мест, то это означает плохую структуру кода	Уровень базы кода
Завистливые функции	Когда из класса В часто ссылаются на методы из класса А, то эти методы принадлежат классу В и должны быть перемещены туда	Уровень базы кода
Неуместная близость	Классы не должны быть слишком связанными друг с другом	Уровень базы кода
Дублирование кода (клоны кода)	Одинаковый или очень похожий код не должен находиться в нескольких частях базы кода	Уровень базы кода
Комментарии	В комментариях должно описываться то, почему написан этот код, а не что он делает	Уровень базы кода
Цепочка вызовов	Избегайте цепочек сообщений, которые представляют собой длинные цепочки вызовов, где методы вызывают методы, которые вызывают методы и т д	Уровень базы кода
Посредник	Если класс делегирует слишком много полномочий, не лучше ли его удалить?	Уровень базы кода
Параллельные иерархии наследования	Когда вы создаете подкласс одного класса, вам нужно создать подкласс другого класса. Это говорит о том, что функциональные возможности двух классов могут принадлежать к общему классу	Уровень базы кода

Таблица 9.1 (окончание)

Запах кода	Объяснение	Уровень
Отказ от наследства	Когда классы наследуют поведение, которое они не используют, то такое наследование не нужно	Уровень базы кода
Стрельба дробью	Обычно изменения являются локальными и применяются в рамках одного класса. Если вам нужно исправить код во множестве мест, то это означает плохую структуру кода	Уровень базы кода
Теоретическая общность	Не добавляйте в кодовую базу код на «пускай будет»; добавляйте только необходимые вещи	Уровень базы кода

Запахи кода на уровне метода

Примером запахов кода на уровне метода является метод, состоящий из большого количества строк кода и выполняющий несколько функций одновременно. Считается, что у такого метода запах «Длинный метод», или «Всемогущий метод». Другим запахом кода является метод, у которого слишком много параметров. Согласно Фаулеру, такой метод имеет запах «Длинный список параметров».

Читателям, которые не знакомы с данной концепцией, настоятельно рекомендую прочитать книгу Фаулера. В табл. 9.1 представлены 22 предложенных Фаулером запаха кода, а также уровень, на котором они возникают.

Запахи кода на уровне класса

Кроме запахов кода, существующих на уровне метода, есть запахи кода на уровне класса. Например, «Большой класс», который иногда называется *всемогущим классом (God class)*. Большой класс — это класс, который выполняет так много функций, что перестает быть понятным абстрактным представлением. Обычно «всемогущие классы» создаются постепенно. Сначала вы можете создать класс, который будет обрабатывать отображение учетной записи клиента. В этом классе могут быть методы, отвечающие за подробную и точную разметку информации о клиенте, например `print_name_and_title()` или `show_date_of_birth`. Постепенно функциональность класса будет расширяться с помощью другого метода, выполняющего простые вычисления, например `define_age()`. Затем вы добавляете еще методы, которые могут составить всех клиентов определенного представителя и т. д. В какой-то момент класс перестанет представлять собой информацию об одном клиенте, а будет представлять информацию обо всех операциях в приложении и, следовательно, станет всемогущим классом.

И наоборот, класс может содержать слишком мало методов и полей, чтобы быть понятным абстрактным представлением. Такому классу Фаулер дал название «Ленивый класс». Ленивый класс тоже может создаваться в течение какого-то времени. В таком случае функциональность переходит в другие классы. Ленивый класс также может быть создан в качестве фиктивного модуля программы, предназначенного для дальнейшего заполнения, чего так никогда и не происходит.

Запахи кода на уровне базы кода

Запахи кода могут возникнуть и на уровне базы кода. Например, когда в кодовой базе в нескольких местах появляется очень похожий код, то у кодовой базы запах «Дублирование кода», который также называется «Клоны кода». Пример клона кода показан на рис. 9.1. Другой пример запаха кода на уровне базы кода — это когда в коде есть несколько методов, которые постоянно обмениваются информацией. Этот запах называется «Цепочка вызовов».

```
int foo(int j) {
  if (j < 0)
    return j;
  else
    return j++;
}
```

Программа А

```
int goo(int j) {
  if ( j < 0 )
    return j;
  else
    return j+2;
}
```

Программа В

Рис. 9.1. Пример клона кода; функции `foo` и `goo` очень похожи, но не являются идентичными

Влияние запахов кода

Наличие запахов кода не всегда означает то, что в коде есть ошибка. Однако известно, что чаще всего в коде с запахом есть ошибки. Фоутс Хом (Foutse Khomh), профессор программной инженерии Политехнической школы в Канаде, изучил кодовую базу Eclipse хорошо известной интегрированной среды разработки для Java и других языков программирования. Он изучил разные версии базы кода Eclipse и проследил за тем, как запахи кода влияют на появление ошибок. Он обнаружил, что всемогущие классы больше всего способствуют появлению ошибок во всех версиях Eclipse, а всемогущие методы способствуют появлению ошибок в Eclipse 2.1[1,2].

Он рассмотрел не только влияние запахов кода на ошибки, но также и на возможность изменения кода. Он обнаружил, что код с запахами с большей вероятностью изменится, чем код без запахов. Было показано, что запахи «Большой класс» и «Длинный метод» оказывают негативное влияние на возможность изменения кода: классы с этими запахами кода изменятся с большей вероятностью, чем классы без запахов, в более чем 75% версий Eclipse[3].

---

[1] «An Empirical Study of the Bad Smells and Class Error Probability in the PostRelease Object-Oriented System Evolution», Journal of Systems and Software, часть 80, № 11, 2007, стр. 1120–1128, Вэй Ли и Раед Шатнави, **http://dx.doi.org/10.1016/j.jss.2006.10.018**.

[2] «The Impact of Code Smells on Software Bugs: A Systematic Literature Review», Алоизио Каир и др. (2018), **https://www.mdpi.com/2078-2489/9/11/273**.

[3] «An Exploratory Study of the Impact of Antipatterns on Software Changeability», Фоутс Хом и др., **http://www.ptidej.net/publications/documents/Research+report+Antipatterns+Changeability+April09.doc.pdf**.

УПРАЖНЕНИЕ 9.1. Вспомните код, который вы недавно редактировали или исправляли. Было ли это связано с запахами кода? Если да, то на каком уровне возникли эти запахи?

9.1.2. Как запахи кода вредят мышлению

Теперь, когда мы подробно рассмотрели запахи кода, давайте рассмотрим связанные с ними когнитивные проблемы. Если вы не хотите написать код с запахами, то важно понимать, как именно они вредят мышлению. Давайте исследуем связь между запахами кода и когнитивными процессами, особенно когнитивной нагрузкой.

Запахи кода Фаулера основаны на его работе и личном опыте кодирования. Хотя Фаулер и не видел такой связи, многие запахи кода можно связать с когнитивными функциями мозга. Основываясь на том, что мы уже знаем о рабочей памяти и долговременной памяти, мы можем понять эффект кода, в котором есть запахи кода.

В предыдущих главах мы рассматривали различные формы замешательства, связанные с разными когнитивными процессами. Запахи кода также связаны с различными когнитивными процессами, что мы и рассмотрим далее.

«Длинный список параметров», сложные «Операторы переключения» — перегрузка рабочей памяти

Учитывая то, что мы знаем о рабочей памяти, мы можем понять, почему «Длинный список параметров» и сложные «Операторы переключения» так трудны для чтения: оба запаха связаны с перегруженной рабочей памятью. В *части I* книги мы объяснили, что емкость рабочей памяти составляет шесть элементов, так что вполне логично, что список параметров, содержащий в себе больше шести параметров, будет слишком трудным для запоминания. При чтении кода будет очень сложно хранить все параметры в рабочей памяти. Следовательно, метод будет слишком труден для понимания.

Конечно, в этом есть свои тонкости. Например, при чтении кода не все параметры будут считаться за отдельные чанки. Давайте посмотрим на сигнатуру метода, представленную в листинге 9.1.

Листинг 9.1. Сигнатура метода на Java принимает в качестве параметров две x- и две y-координаты

```
public void line(int xOrigin, int yOrigin, int xDestination, yDestination) {}
```

Скорее всего, ваш мозг примет их за два, а не за четыре чанка: первый чанк — это координаты x и y, а второй чанк — это точка привязки координат. Следовательно, ограниченное количество параметров, которое вы сможете запомнить, зависит от контекста и уже имеющихся у вас знаний. Однако длинные списки параметров с большей вероятностью приведут к перегрузке рабочей памяти. К перегрузке рабочей памяти также могут привести и сложные операторы переключения.

«Всемогущий класс», «Длинный метод» — невозможно эффективно разбить код на чанки

При работе с кодом мы постоянно создаем абстрактные представления. Вместо того, чтобы поместить всю функциональность в одну функцию `main()`, мы делим функциональность на небольшие функции с говорящими именами. Группы связанных атрибутов и функций объединяются в классы. Преимущество распределения функциональности по отдельным функциям, классам и методам состоит в том, что их имена могут служить документацией.

Когда программист вызывает `square(5)`, то он сразу понимает, какое значение вернется. Однако есть еще одно преимущество имен классов и методов, которое заключается в том, что с их помощью можно разбить код на чанки. Например, когда вы видите блок кода, содержащий в себе функцию с именем `multiples()` и функцию с именем `minimum()`, то вы можете понять, это эта функция высчитывает наименьший общий знаменатель; при этом вам даже не нужно подробно изучать код. Вот почему запахи кода, связанные с большими блоками кода, включая «Всемогущий класс» и «Длинный метод», вредят коду: вы не можете сразу понять код и начинаете читать его построчно.

«Клоны кода» — невозможно правильно разбить код на чанки

Запах «Клоны кода», или «Дублирование кода» возникает тогда, когда в кодовой базе есть много фрагментов похожего кода.

С нашими знаниями о рабочей памяти нам легко понять, почему дублированный код является запахом кода. Давайте еще раз рассмотрим два метода, представленных теперь уже на рис. 9.2. Когда вы видите вызов функции `goo()`, очень похожий на `foo()`, то ваша рабочая память может затребовать у долговременной памяти информацию о `foo()`. Ваша рабочая память словно говорит: «Эта информация нам еще понадобится». Затем вы смотрите на код самой функции, но ваша рабочая память, получившая информацию о функции `foo()`, будет воспринимать его не как функцию `goo()`, а как функцию `foo()`.

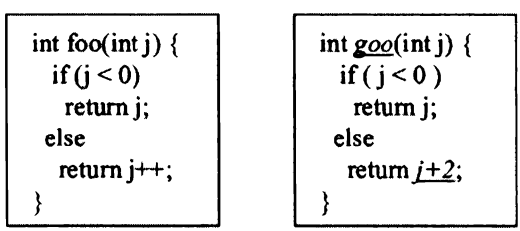


Рис. 9.2. Две функции с похожими именами, но не совсем похожим функционалом
Так как имена и реализация этих функций так похожи, наш мозг, вероятно, будет путать эти методы

Следовательно, ваш мозг поместит метод `goo()`, слегка отличающийся от метода `foo()`, в одну категорию с `foo()` точно так же, как несколько вариантов сицилианской защиты в представлении множества шахматистов были запомнены как «сици-

лианские». Вот так же может возникнуть неверное представление и о том, что метод `goo()` — это метод `foo()`, даже несмотря на то, что методы возвращают разные значения. Мы уже знаем, что подобные неправильные представления могут оставаться в памяти надолго. И вам может потребоваться много времени для того, чтобы окончательно осознать, что метод `goo()` и метод `foo()` — это не один и тот же метод.

УПРАЖНЕНИЕ 9.2. Снова посмотрите на код с запахами кода из упражнения 9.1. Какие когнитивные процессы стали причиной неверного понимания кода?

9.2. Зависимость когнитивной нагрузки от плохих имен

В данной главе мы сосредоточили внимание на создании кода, легкого для понимания. До этого мы рассматривали запахи кода, предложенные Фаулером, например «Длинный метод» и «Клоны кода», а также их влияние на когнитивную нагрузку.

Запахи кода — это части кода, в которых есть *структурные антипаттерны проектирования*: например, сам по себе код правильный, но его структура затрудняет обработку кода. В коде также могут присутствовать *семантические антипаттерны проектирования*: у кода правильная структура, правильно созданные классы с небольшими методами, но их имена приводят в замешательство. Такие проблемы описаны во второй коцепции — *лингвистических антипаттернах проектирования*. Так как обе концепции рассматривают разные аспекты кода, они хорошо работают вместе.

9.2.1. Лингвистические антипаттерны проектирования

Впервые лингвистические антипаттерны проектирования описала Венера Арнаудова (Venera Arnaoudova), сейчас являющаяся профессором в штате Вашингтон. Арнаудова описывает лингвистические антипаттерны проектирования как различия между лингвистическими элементами в коде и ролями, которые они играют. Лингвистические элементы кода описываются как части кода на естественном языке, например сигнатуры методов, документация, имена элементов, типы или комментарии. Антипаттерны проектирования возникают в тот момент, когда лингвистический элемент больше не соответствует своей роли. В качестве примера можно привести переменную с именем `initial_element`, которая содержит не элемент, а индекс, или переменную с именем `isValid`, предполагающую булево значение, но содержащую целое число.

Лингвистические антипаттерны проектирования часто встречаются в именах методов или функций, когда имя не соответствует тому, что на самом деле делает метод или функция. Например, по имени функции можно понять, что она возвращает коллекцию, но на самом деле она возвращает один объект: например, метод `getCustomers`, возвращающий булево значение. И хотя данный метод имеет право на

существование в том случае, если он проверяет наличие клиента, он все еще может приводить в замешательство.

Арнаудова описывает шесть видов лингвистических антипаттернов проектирования, которые представлены в табл. 9.2.

Таблица 9.2. *Шесть лингвистических антипаттернов проектирования Арнаудовой*

Методы, выполняющие больше функций, чем описано в имени
Методы, выполняющие меньше функций, чем описано в имени
Методы, выполняющие прямо противоположное тому, что описано в имени
Идентификаторы, имена которых говорят, что они содержат больше, чем содержит элемент
Идентификаторы, имена которых говорят, что они содержат меньше, чем содержит элемент
Идентификаторы, имена которых говорят противоположное тому, что содержит элемент

Определив лингвистические антипаттерны проектирования, Арнаудова изучила их на примере семи проектов с открытым кодом. Она обнаружила, что антипаттерны довольно-таки распространены; например, 11% методов установки вдобавок к настройке поля еще и возвращают значение. У 2,5% методов имя метода и соответствующий ему комментарий давали противоречивые описания работы метода, а огромные 64% идентификаторов, начинавшихся с `is`, не были булевым значением.

> **ПРОВЕРЬТЕ СВОЙ КОД НА НАЛИЧИЕ В НЕМ ЛИНГВИСТИЧЕСКИХ АНТИПАТТЕРНОВ ПРОЕКТИРОВАНИЯ**
>
> Вам интересно, если ли в вашем коде лингвистические антипаттерны проектирования? Основываясь на своем исследовании, Арнаудова разработала Linguistic Anti-Pattern Detector (LAPD), который может обнаружить антипаттерны проектирования в коде на Java. LAPD доступен в виде расширения плагина Eclipse Checkstyle[4].

И хотя на интуитивном уровне мы можем догадываться, что лингвистические антипаттерны проектирования могут привести в замешательство и соответственно вызвать дополнительную когнитивную нагрузку, данный факт подтвержден наукой. Но перед тем, как мы рассмотрим влияние лингвистических антипаттернов на когнитивную нагрузку, давайте сначала узнаем, как именно можно измерить когнитивную нагрузку.

9.2.2. Измерение когнитивной нагрузки

В предыдущих главах мы рассказывали о когнитивной нагрузке, а также перегрузке рабочей памяти. Мы также показали вам несколько примеров упражнений, которые вызывают высокую когнитивную нагрузку: например, чтение кода, когда связанная информация находится в разных частях методов или файлах, или чтение кода, в котором есть множество незнакомых ключевых слов или концепций программи-

[4] **http://www.veneraarnaoudova.ca/linguistic-anti-pattern-detector-lapd/.**

рования. Однако мы до сих пор не рассмотрели то, как можно измерить когнитивную нагрузку.

Шкала Пааса для когнитивной нагрузки

При измерении когнитивной нагрузки ученые очень часто пользуются шкалой Пааса, разработанной нидерландским психологом Фредом Паасом (Fred Paas), который сейчас является профессором в Университете им. Эразма Роттердамского в Роттердаме. Шкала показана в табл. 9.3.

В последние несколько лет шкала Пааса подвергалась критике, т. к. это довольнотаки маленький тест, состоящий всего из одного вопроса. Также остается непонятным, могут ли участники отличить, например, очень высокую нагрузку от сверхвысокой нагрузки.

Но, несмотря на все недостатки и критику, шкала Пааса широко распространена. В предыдущих главах мы рассмотрели стратегии чтения кода, а также практиковались в решении упражнений. Когда вы читаете незнакомый код, вы можете воспользоваться шкалой Пааса, которая поможет вам понять код и ваше отношение к коду.

Таблица 9.3. *В шкале Пааса участники самостоятельно оценивали свою когнитивную нагрузку по 9-балльной шкале*

Сверхнизкое умственное напряжение
Очень маленькое умственное напряжение
Маленькое умственное напряжение
Довольно низкое умственное напряжение
Среднее умственное напряжение
Довольно высокое умственное напряжение
Высокое умственное напряжение
Очень высокое умственное напряжение
Сверхвысокое умственное напряжение

УПРАЖНЕНИЕ 9.3. Откройте незнакомый фрагмент кода и используйте шкалу Пааса для оценки умственного напряжения, которое вы испытали в попытке понять код. Подумайте о том, почему фрагмент кода вызвал у вас такую когнитивную нагрузку. Данное упражнение поможет вам понять, чтение каких видов кода дается вам сложнее всего.

	Уровень когнитивной нагрузки	Причина
Сверхнизкое умственное напряжение		
Очень маленькое умственное напряжение		
Маленькое умственное напряжение		

(окончание)

	Уровень когнитивной нагрузки	Причина
Довольно низкое умственное напряжение		
Среднее умственное напряжение		
Довольно высокое умственное напряжение		
Высокое умственное напряжение		
Очень высокое умственное напряжение		
Сверхвысокое умственное напряжение		

Измерение нагрузки по глазам

В дополнение метрики, основанной на восприятии участников, в новых исследованиях все чаще используется биометрика. Измеряя реакцию организма на определенную задачу, можно определить когнитивную нагрузку, которая была вызвана тем, чем занят человек.

Примером биометрических измерений может стать отслеживание движений глаз человека. С помощью устройств отслеживания взгляда можно узнать, насколько человек сконцентрирован. Это можно определить, например, по скорости моргания человека, которая является мерой того, как часто человек моргает. Несколько исследований показали, что человек всегда моргает по-разному, и частота моргания может зависеть от того, чем он занят в данный момент. Результаты некоторых исследований показали, что когнитивная нагрузка также влияет на частоту моргания; чем сложнее задача, тем реже человек моргает. Второй показатель, по которому можно определить когнитивную нагрузку, — это зрачок. Исследования показали, что чем сложнее задача, тем большую когнитивную нагрузку испытывает человек; при этом нагрузку определяли по размеру зрачка[5].

Предположение, почему моргание связано с когнитивной нагрузкой, заключается в том, что моргание указывает на то, насколько сильно ваш мозг пытается увеличить ваше воздействие на трудную задачу и пытается таким образом получить как можно больше зрительных раздражителей. Именно по этой причине при решении трудных задач зрачок увеличивается — большой зрачок может охватывать больше информации, соответственно ваш мозг тоже будет искать больше связанной информации.

Измерение нагрузки по коже

В добавок к измерениям по глазам кожа также может рассказать нам о когнитивной нагрузке. Температура тела и потоотделение являются хорошими показателями когнитивной нагрузки.

---

[5] «Task-Evolved Pupillary Response to Mental Workload in Human-Computer Interaction», Шамси Икбал и др. (2004), **https://interruptions.net/literature/Iqbal-CHI04-p1477-iqbal.pdf**.

И хотя эти биометрические способы определения когнитивной нагрузки могут казаться классными, исследование показало, что они очень часто коррелируют со шкалой Пааса. Так что если вы захотите использовать фитнес-трекер для того, чтобы понять, насколько легко читается ваш код, то вы можете просто выполнить упражнение 9.3.

Измерение нагрузки по мозгу

В *главе 5* мы познакомились с аппаратом фМРТ как способом измерения того, за какие виды деятельности отвечает наш мозг. Аппараты фМРТ не ошибаются, однако у них есть ограничения в использовании. Например, когда участники подключены к аппарату, они должны лежать неподвижно — следовательно, в этот момент они не могут заниматься кодированием. Даже чтение кода происходит немного «искусственно», т. к. код можно вывести только на маленьком экране. Тот факт, что люди, подключенные к аппарату, не могут двигаться, сильно ограничивает их взаимодействие с кодом; к примеру, они не могут листать код или щелкать на идентификаторах, чтобы перейти к их определению в коде. Они также не могут искать ключевые слова или элементы с помощью сочетания клавиш <Ctrl>+<F>. Из-за таких ограничений фМРТ используются альтернативные способы измерения активности мозга.

Запись биотоков мозга

Одним из альтернативных способов измерения активности мозга является *запись биотоков мозга*, или *электроэнцефалограмма* (ЭЭГ). Аппарат ЭЭГ измеряет колебания активности нейронов головного мозга, вызванные измерением колебаний электрической активности мозга. Вторым альтернативным способом измерения активности мозга является *функциональная ближняя инфракрасная спектроскопия* (fNIRS). При проведении fNIRS также может использоваться оголовье, которое позволяет провести более приближенное к реальности исследование.

Как мы узнаем из следующего раздела, аппарат fNIRS использовался для понимания взаимосвязи между лингвистической и когнитивной нагрузкой.

В аппарате fNIRS применяется инфракрасный свет и датчики света. Так как гемоглобин в крови интенсивно поглощает свет, то данный аппарат может использоваться для мониторинга оксигенации головного мозга. Инфракрасный свет проходит через мозг, однако часть света достигает датчиков света на оголовье. С помощью количества света, достигшего датчиков света, определяется насыщенность гемоглобина кислородом. Если кровь перенасыщена кислородом, то когнитивная нагрузка повышена.

Аппарат fNIRS особенно чувствителен к двигательным артефактам и свету, поэтому участники по мере возможностей не должны много двигаться и прикасаться к устройству во время измерений. И хотя fNIRS удобнее в использовании, чем фМРТ, данный аппарат проигрывает оголовью для ЭЭГ.

Функциональная fNIRS-томография и программирование

В 2014 году Такао Накагава (Takao Nakagawa), исследователь из Института науки и технологии Нары в Японии, измерил активность мозга во время ознакомления с кодом с помощью портативного аппарата fNIRS[6]. Участников попросили прочитать два варианта алгоритма, написанного на C. Один вариант был обычным, а второй вариант был специально усложнен. Например, счетчики циклов и другие значения были так изменены, чтобы переменные обновлялись с нерегулярной скоростью. Такие изменения не повлияли на функциональные возможности программы.

Участникам, носившим оголовье аппарата fNIRS, показали как исходный, так и усложненный код. Чтобы исключить возможность активации эффекта накопленного опыта, участник случайным образом получал либо исходный, либо усложненный вариант кода.

Результаты исследования показали, что у 8 из 10 участников при изучении усложненной программы увеличивался перенасыщенный кислородом кровоток. Из данного результата можно сделать вывод, что когнитивную нагрузку при программировании можно определить с помощью измерения церебрального кровотока аппаратом fNIRS.

9.2.3. Лингвистические антипаттерны и когнитивная нагрузка

С помощью аппарата fNIRS исследователи смогли измерить влияние лингвистических антипаттернов проектирования на когнитивную нагрузку. В 2018 году Сара Фахури (Sarah Fakhoury), в настоящее время являющаяся аспиранткой, работающей под руководством Арнаудовой, изучала связь между лингвистическими антипаттернами и когнитивной нагрузкой. Каждый из 15 участников читал фрагменты кода, взятые из проектов с открытым кодом. Исследователи специально добавляли в фрагменты ошибки и просили участников найти их. Само по себе задание было не так важно; особое внимание уделялось тому, что обнаружение ошибок позволяло участникам понять код.

Исследователи внесли еще несколько изменений в код, чтобы получилось четыре варианта фрагментов кода:

1. Фрагменты, в которые добавили лингвистические антипаттерны проектирования.
2. Фрагменты, в которые добавили структурные антипаттерны.
3. Фрагменты с обеими ошибками.
4. Фрагменты без ошибок.

---

[6] «Quantifying Programmers' Mental Workload during Program Comprehension Based on Cerebral Blood Flow Measurement: A Controlled Experiment», Такао Накагава и др., **https://posl.ait.kyushu-u.ac.jp/~kamei/publications/Nakagawa_ICSENier2014.pdf**.

Участников разделили на четыре группы, и каждая группа читала фрагменты в разном порядке, чтобы исключить любой эффект накопленного опыта.

В этом исследовании использовалось устройство отслеживания взгляда, определяющее, как люди читают код, а также аппарат fNIRS, определяющий когнитивную нагрузку. Сначала результаты исследования показали, что фрагменты кода, в которые были добавлены лингвистические антипаттерны, просматривались чаще, чем остальная часть кода, — эти данные были взяты с устройств отслеживания взгляда.

Кроме того, аппарат fNIRS показал, что наличие в коде лингвистических антипаттернов значительно увеличивает приток оксигенированной крови (т. е. участник испытывает повышенную когнитивную нагрузку).

Интересно еще и то, что в данном исследовании предлагались фрагменты кода, в которые были добавлены структурные антипаттерны. Код был оформлен способом, который противоречил принятым стандартам оформления кода на Java; например открывающие и закрывающие скобки стояли не на нужных строках и были написаны с неверным отступом. Сложность кода также была увеличена, например, с помощью дополнительных циклов.

Это позволило исследователям сравнить влияние структурных антипаттернов и лингвистических антипаттернов. Реакция участников на фрагменты со структурными антипаттернами была негативной, а один из участников сказал, что «Ужасное оформление сильно увеличивает нагрузку». Однако исследователи сравнили результаты по фрагментам со структурными антипаттернами и фрагментам, в которых не было никаких ошибок, и заявили, что не заметили особой разницы в показателях когнитивной нагрузки.

9.2.4. Почему лингвистические антипаттерны вызывают замешательство

Мы уже знаем, что код с большим количеством лингвистических антипаттернов вызывает большую когнитивную нагрузку. Для того чтобы определить связь между лингвистическими антипаттернами и когнитивной нагрузкой, нужно проводить дополнительные исследования с использованием измерений мозговой активности. Но мы, основываясь на наших знаниях о рабочей и долговременной памяти, можем сами предположить влияние лингвистических антипаттернов на работу человека.

При чтении кода с лингвистическими антипаттернами возникают две когнитивные проблемы. В *части I* мы рассматривали трансференцию во время обучения: при чтении незнакомого материала, например кода, который писали не вы, ваша долговременная память ищет связанную информацию. При чтении противоречивого имени вы получите неправильную информацию. Например, прочитав имя `retrieveElements()`, вы сразу можете подумать о функции, возвращающей список элементов. Вы понимаете, что вы можете отсортировать, отфильтровать или изменить возвращаемый элемент, однако это нельзя сделать с одним элементом.

Вторая причина, по которой лингвистические антипаттерны вызывают замешательство, заключается в том, что они могут приводить к неправильному разбиению кода

на чанки, как и клоны кода. Когда вы читаете имя переменной, например `isValid`, вы предполагаете, что переменная является булевым значением. И на этом ваш мозг прекращает думать об имени переменной; но если бы вы обратили особое внимание на имя, то увидели бы, что оно используется в качестве списка потенциальных возвращаемых значений. В попытке сохранить энергию ваш мозг принимает неверное решение. Как мы уже видели ранее, такие неверные предположения могут оставаться в вашей памяти продолжительное время.

Выводы

- Запахи кода, например «Длинный метод», указывают на проблемы со структурой кода. Существует несколько когнитивных причин, почему запахи кода вызывают дополнительную когнитивную нагрузку. Например, «Дублированме кода» не позволяет удачно разбить код на чанки, а «Длинный список параметров» перегружает рабочую память.
- Измерить когнитивную нагрузку можно разными способами, и одним из таких способов является использование биометрических датчиков — например, измерение частоты моргания или температуры тела. Если вы хотите измерить когнитивную нагрузку, то вы можете воспользоваться шкалой Пааса.
- Лингвистические антипаттерны указывают на места в кодовой базе, где код выполняет действие, не соответствующее имени. Это вызывает дополнительную когнитивную нагрузку. Возможно, это вызвано тем, что при попытке поддержать мышление долговременная память находит неверную информацию. Вы также можете неправильно разбить код на чанки, т. к. ваш мозг делает неверное предположение о работе кода.

10 Совершенствуем навыки решения сложных задач

В этой главе:

- сравним роли, которые играют мнемонические системы в решении задач;
- изучим то, как автоматизация навыков помогает нам решать большие и сложные задачи;
- поймем, как улучшить долговременную память для легкого решения задач.

В последних главах мы рассматривали то, что *не* стоит делать при кодировании и почему. Мы рассматривали влияние плохих имен в *главе 8*, а также влияние запахов кода на способность понимать код в *главе 9*.

Ранее, в *главе 6*, мы обсуждали различные способы поддержки рабочей памяти при решении задач программирования. В этой главе мы снова рассмотрим несколько стратегий, которые помогут вам решить задачи, однако в этот раз особое внимание мы уделим улучшению долговременной памяти.

Сначала мы рассмотрим, что значит решить задачу. После того как мы во всех подробностях рассмотрим решение задач, мы узнаем, как лучше всего это делать. К концу главы вы узнаете две стратегии, которые улучшат ваши навыки программирования и решения задач. Первая стратегия, которую мы рассмотрим в этой главе, — это *автоматизация* (способность «на автомате» выполнять небольшие задачи). Это полезная стратегия, потому что чем меньше времени вы тратите на мелкие задачи, тем легче вы будете решать сложные задачи. Затем мы рассмотрим решение задач с помощью кода, написанного другими программистами, как средство усовершенствования ваших навыков решения задач.

10.1. Что такое решение задач

Цель этой главы — научить вас стратегиям, которые помогут вам улучшить навык решения задач, а также рассказать вам о роли долговременной памяти в решении задач. Однако сначала давайте рассмотрим, что значит решать задачи.

10.1.1. Элементы решения задач

Решение задачи состоит из трех важных элементов:

- целевое состояние, которого мы хотим достигнуть. Как только мы достигаем целевого состояния, задача считается решенной;
- исходное состояние, в котором сначала находится задача, которую нужно решить;
- правила, указывающие нам, как именно из исходного состояния достичь целевого состояния.

Например, попробуйте сыграть в крестики-нолики. Исходное состояние — это пустое поле, желаемое состояние — три крестика в ряд, а в правилах говорится о том, что вы можете поставить крестик в любую свободную клетку на поле. Когда на уже существующий сайт вы добавляете строку поиска, то исходным состоянием является база кода, а желаемым состоянием является, например, удовлетворенный пользователь или прохождение модульного тестирования. В программировании условия задачи часто формулируются как некие рамки: например, реализовать на JavaScript данную конкретную возможность или не провалить дополнительные тесты при реализации этой новой возможности.

10.1.2. Пространство состояний

Все шаги, допустимые при решении задачи, называются *пространством состояний* задачи. В игре крестики-нолики пространством состояний являются все клетки. Для небольших задач, например игры в крестики-нолики, пространство состояний можно визуализировать. На рис. 10.1 показана часть пространства состояний игры в крестики-нолики.

При добавлении кнопки на сайт все возможные программы на JavaScript являются пространством состояний. Решение задачи теперь зависит от того, примете ли вы верное решение и добавите ли правильные строки в код для достижения поставленной цели. Говоря другими словами, решение задачи означает просмотр пространства состояний самым оптимальным образом, а также достижение целевого состояния за минимальное количество шагов.

УПРАЖНЕНИЕ 10.1. Просмотрите то, что вы создали с помощью кода в последние несколько дней. Какие черты были у этой задачи?

- Какого целевого состояния вы хотели достичь?
- Как проверялось целевое состояние: вручную вами, вручную кем-то еще, модульным или приемочным тестированием?

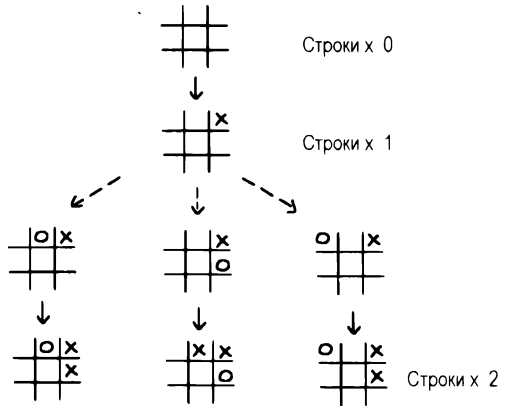


Рис. 10.1. Часть пространства состояний игры в крестики-нолики, где пунктирные стрелки означают ход за нолики, а сплошные стрелки — ход за крестики. Целевое состояние для крестиков — три крестика в ряд

- Каким было исходное состояние задачи?
- Какие были правила и ограничения?

10.2. Какую роль при решении задач программирования играет долговременная память

Теперь, когда мы знаем, что означает решить задачу, давайте рассмотрим, что в этот момент происходит в вашем мозге. В *главе 6* рассказывается, что происходит с рабочей памятью при решении задач программирования. Если вы испытываете большую когнитивную нагрузку, то ваш мозг не сможет правильно обрабатывать информацию — в таком случае процесс программирования усложнится. Однако долговременная память тоже играет роль при решении задач.

10.2.1. Решение задачи — это отдельный когнитивный процесс?

Некоторые считают, что решение задачи — это навык общего характера, и значит, что это конкретный процесс в вашем мозге. Венгерский математик Джордж Полиа (George Pólya) — знаменитый мыслитель в области решения задач. В 1945 году Полиа написал короткую, но известную книгу «Как решать задачу». В книге предлагается «система мышления», подходящая для решения любой задачи. Система мышления состоит из трех шагов:

1. Понимание задачи.
2. Составление плана.
3. Выполнение плана.

Но несмотря на популярность подхода, исследования показывают, что решение задачи не является навыком общего характера или когнитивным процессом. Существуют две причины, по которым стратегии решения задач работают не так хорошо, и эти причины связаны с долговременной памятью.

При решении задач вы используете долговременную память

Во-первых, при решении задачи вы обращаете внимание на желаемое целевое состояние и правила, которые нужно учитывать. Сама задача также влияет на выбор решения. Давайте рассмотрим влияние имеющихся знаний на решение задачи на примере программирования. Представим, что вам нужно реализовать код для того, чтобы определить, является ли данная строка ввода `s` палиндромом. Написать код для этой задачи нужно на Java, APL и BASIC. Давайте посмотрим, как шаги Полиа будут работать с задачей программирования.

- Поймите задачу.

  Мы считаем, что для вас, как программиста, понять задачу не составит никакого труда. Вы также сможете сформулировать несколько тестовых примеров для проверки кода.

- Составьте план.

  Второй шаг Полиа выполнить чуть труднее. Ваш план зависит от возможностей языка программирования, на котором вы собираетесь решать задачу.

  Например, есть ли в выбранном языке обратный метод или функция? В таком случае вы сможете просто проверить, совпадает ли строка `s` с `reverse(s)`. Возможно, вы знаете, что на Java можно использовать StringBuldier, у которого уже есть функция `reverse()`, но имеется ли такая функция в языках APL или BASIC? Без доступа к важной информации, составляющей основу решения задачи, составить план очень сложно.

- Выполните план.

  Третий шаг Полиа также зависит от вашего понимания языка программирования. Я могу сказать, что на APL есть функция `reverse()` (на BASIC этой функции нет), но вы уже сталкивались с APL и знаете, что она не будет называться `reverse()`, т. к. все ключевые слова языка APL являются операторами. Это усложняет выполнение плана.

Вашему мозгу проще решить знакомые задачи

Вторая причина, по которой общие стратегии решения задач чаще всего не работают, заключается в работе долговременной памяти. В *главе 3* мы обсуждали то, что долговременная память представляет собой сеть, где все воспоминания и информация связаны друг с другом. Мы также узнали тот факт, что при решении задачи наш мозг извлекает из долговременной памяти информацию, которая связана с решаемой задачей.

Использование общий стратегии решения задач, например «составление плана» Полиа, создает когнитивные нарушения. В долговременной памяти может хранить-

ся много полезных стратегий, и мозг будет пытаться извлечь их при решении задачи. Однако, когда мы пытаемся решить задачу с помощью общей стратегии, то нужные стратегии могут быть не найдены. Как мы уже отмечали в *главе 3*, долговременная память нуждается в подсказках, которые помогут найти нужные воспоминания. И чем понятнее и конкретнее подсказка, тем больше вероятность нахождения нужной информации. Например, когда вам необходимо выполнить деление хвоста списка, вряд ли ваша долговременная память извлечет нужные воспоминания, если вы начнете составлять план. Размышление о таких вещах, как деление хвоста списка или вычитание кратных делителя, с большей вероятностью позволит вам составить правильный план.

Как мы говорили в *главе 7*, трансференция знаний из одной области, допустим шахмат, в другую область, например математику, вряд ли пройдет успешно. Это же правило работает и со стратегиями решения задач: вы не можете применять стратегию решения задачи из одной области к другой области.

10.2.2. Как научить долговременную память решать задачи

Теперь мы знаем, что решение задачи — это не когнитивный процесс. И у нас возникает вопрос: как практиковаться в решении задач? Чтобы разобраться с этим, давайте рассмотрим то, как думает наш мозг. Ранее уже говорилось, что все мысли формируются в рабочей памяти. Мы также знаем, что рабочая память не только формирует мысли, но и связана с долговременной и кратковременной памятью.

Когда вы думаете о какой-то задаче, например реализации кнопки отбора в приложении, то ваша рабочая память принимает решения касаемо того, что вы делаете. Но перед тем, как ваша рабочая память примет решение, она должна сделать две вещи. Во-первых, она получает информацию от кратковременной памяти о задаче, например о требованиях к кнопке или о коде, который вы прочитали.

В это время в долговременной памяти ищется релевантная информация. Связанная информация, например реализация сортировки или информация о кодовой базе, также поступает в рабочую память. Чтобы лучше понять методы решения задач, давайте изучим второй из этих процессов — поиск информации в долговременной памяти.

10.2.3. Два вида памяти, наиболее существенные при решении задачи

Немного позже в этой главе мы рассмотрим две стратегии, которые помогут вам улучшить навык решения задач. Но перед этим давайте рассмотрим виды памяти, которые есть у людей, и их роль при решении задач. Понимание видов памяти играет большое значение, т. к. разные виды памяти создаются по-разному.

В долговременной памяти может быть нескольких видов памяти. (Все виды показаны на рис. 10.2.) Во-первых, это *процедурная* (или *имплицитная*) память, которая хранит воспоминания о моторных навыках, или навыках, которые находятся ниже

уровня сознания человека. Имплицитная память — это, например, умение завязывать шнурки или кататься на велосипеде.

Второй тип памяти, играющий роль при решении задачи, — это *декларативная* (или *эксплицитная*) память. Существуют факты, которые вы можете вспомнить, и вы уверены в том, что вы знаете их. Например, факт, что Барак Обама был 44-м президентом США, или что цикл `for` на Java пишется как `for (i = 0; i < n; i++)`.

Как показано на рис. 10.2, декларативная память включает в себя две подсистемы: эпизодическую память и семантическую память. Эпизодическая память — это то, что мы имеем в виду, когда говорим слово «память». Воспоминания о событиях, например поездке в летний лагерь, когда вам было 14, первой встрече с вашим избранником или о том, как вы потратили целых три часа на поиск ошибки в коде, а оказалось, что ошибка была в тестовой программе.

Рис. 10.2. Существуют разные виды памяти. Процедурная (или имплицитная) память отвечает за то, как что делать. Декларативная (или эксплицитная) память содержит информацию о том, что мы точно знаем. Декларативная память также делится на воспоминания, которые вы испытали и которые хранятся в эпизодической памяти, и факты, которые вы знаете и которые хранятся в семантической памяти

Другая подсистема декларативной памяти называется *семантической* памятью. Семантическая память — это память фактов, концепций, например, что лягушка на французском звучит как *grenoulle*, или что 7 умножить на 5 — это 35, или что класс на Java используется для объединения данных.

Семантическая память — это то, что мы тренировали в *главе 3* с помощью дидактических карточек. Эпизодическая память создается очень легко, и, как и в случае с семантической памятью, воспоминания будут извлекаться намного легче, если вы будете часто думать о них.

Какие виды памяти играют роль при решении задач

Все эти виды памяти играют роль при программировании. Если задуматься о том, какая же память используется при программировании, то первой на ум приходит эксплицитная память. Ведь помнить при программировании, как сделать цикл на

Java, нужно всегда. Однако, как показано на рис. 10.2, другие виды памяти также играют роль при решении задач.

Эпизодическая память используется тогда, когда вы вспоминаете ваше решение задачи в прошлом. Например, вы решаете задачу, связанную с иерархией. Вы вспоминаете, что раньше вы использовали древовидную схему. Исследования показывают, что при решении задач профессионалы чаще всего полагаются на эпизодическую память. Профессионалы в некотором смысле воссоздают знакомую задачу, а не решают ее. А это значит, что вместо того, чтобы искать новое решение, люди полагаются на уже проверенное решение, которое было верно для похожей задачи. В этой главе мы подробно рассмотрим то, как укрепить эпизодические воспоминания, чтобы улучшить навык решения задач.

Как показано на рис. 10.3, при программировании человек полагается и на имплицитную память. Например, многие программисты могут печатать вслепую — за это отвечает процедурная память. Кроме алфавита, вы также можете использовать разнообразные сочетания клавиш, даже не замечая этого. Например, при опечатке вы нажимаете <Ctrl>+<Z> или «на автомате» добавляете закрывающую скобку к открывающей. Имплицитная память также играет роль в том случае, когда вы автоматически ставите точку останова в строке, где, как вам кажется, есть ошибка. Иногда то, что мы называем интуицией, на самом деле случается с вами при решении задач, похожих на те, которые вы решали ранее, — вы просто знаете, что нужно делать, не представляя, как именно это делается.

Рис. 10.3. Различные виды памяти и их роль при программировании

Потеря знаний или навыков

Хотя мы и видели, что имплицитная память помогает быстро выполнять знакомые задачи, наличие этой памяти может иметь негативный эффект. В *главе 7* мы обсуждали негативную трансференцию: знание чего-то одного вредит изучению чего-то другого. Большой объем имплицитной памяти может навредить гибкости вашего ума. Например, вы научились печатать вслепую на клавиатуре Qwerty, но переучиваться для работы на клавиатуре Dvorak будет намного сложнее, чем если бы вы

никогда не учились печатать на Qwerty. Частично это связано с тем, что у вас есть имплицитная память о том, как нужно что-то делать.

Возможно, вы сталкивались с трудностью отказа от имплицитной памяти — например, при изучении второго языка программирования, синтаксис которого полностью отличался от синтаксиса первого языка. Например, при переходе с C# или Java на Python вы какое-то время будете заключать операторы в фигурные скобки. Я перешла с C# на Python уже много лет назад, но даже спустя столько лет очень часто при итерации по списку пишу `foreach` вместо `for`. Это происходит из-за имплицитной памяти, созданной при программировании на C#, которая до сих пор не забылась.

УПРАЖНЕНИЕ 10.2. В следующий раз, когда вы начнете писать код, следите за воспоминаниями, которые вы используете в процессе.

Воспользуйтесь данной таблицей, чтобы понять, какие программы или задачи активируют разные виды памяти. Вы можете выполнить это упражнение несколько раз с разными программами. Если вы выполните упражнение несколько раз и будете следить за своим прогрессом, то вы заметите, что при работе с малознакомыми языками программирования или проектами вы будете полагаться на семантическую память; при этом при работе со знакомым кодом или языком программирования вы будете опираться на процедурные и эпизодические воспоминания.

Программа или задача	Процедурная	Эпизодическая	Семантическая

10.3. Автоматизация: создание имплицитной памяти

Теперь, когда мы знаем, почему задачи так сложно решать и какие виды памяти играют роль в решении задач, давайте посмотрим, как можно улучшить навыки решения задач. На самом деле улучшить этот навык можно двумя способами. Первый способ — это *автоматизация*. Если вы несколько раз до идеала довели какое-то действие, то в дальнейшем вы сможете совершать это действие «на автомате», совершенно не задумываясь: например, ходить, читать, завязывать шнурки.

Многие люди довели повседневные навыки до автоматизма: например, вождение машины или катание на велосипеде, а также навыки, необходимые в такой области, как математика. Например, вы научились выносить за скобки уравнения типа $x^2 + y^2 + 2xy$. Если вы один из тех, кто автоматизировал этот навык, то вы без каких-либо усилий сможете сразу превратить это выражение в $(x + y)^2$. Здесь важно то, что возможность без труда преобразовывать уравнения помогает решать более

сложные вычисления. Я очень часто думаю об автоматизации как о процессе, который в игре открывает новое умение. Когда вы в игре научились делать двойной прыжок, вам открываются области уровня, до которых вы не могли добраться ранее.

Например, если вы можете без особого труда разложить уравнение на множители, то, посмотрев на уравнение, подобное такому:

$$\frac{x^2 + y^2 + 2xy}{(x+y)},$$

вы сразу же поймете, что ответом будет $(x + y)$. Если вы еще не автоматизировали этот навык, то решение этой задачи дастся вам намного труднее.

Как вы уже поняли, автоматизация навыков программирования является решающим фактором при решении сложных задач. Однако как достичь автоматизации навыков?

Перед автоматизацией навыков давайте рассмотрим то, как можно улучшить имплицитную память при программировании. Ранее мы уже обсуждали, что иногда имплицитная память может мешать при изучении нового языка программирования, например когда вы продолжаете вставлять фигурные скобки, решая задачу на языке Python. Это может показаться маленькой ошибкой, но она вызовет некоторую когнитивную нагрузку. Как было сказано в *главе 9*, когнитивная нагрузка показывает, насколько занят и заполнен мозг. Когда вы испытываете слишком большую когнитивную нагрузку, то думать о чем-то очень сложно. Имплицитная память интересна тем, что в то время, когда вы практикуете имплицитную память, ваш мозг почти не тратит энергию. Например, если вы умеете кататься на велосипеде или печатать вслепую, то у вас с этим не возникнет никаких сложностей. Так как эти упражнения почти не создают когнитивной нагрузки, вы можете одновременно ездить на велосипеде и есть мороженое или водить машину и разговаривать по телефону.

10.3.1. Имплицитная память с течением времени

Чем больше у вас сохранено информации о программировании в имплицитной памяти, тем проще вам будет решать большие задачи, во время решения которых вы испытываете когнитивную нагрузку. Но как увеличить имплицитную память? Чтобы понять это, давайте рассмотрим, как она создается в мозге.

В *главе 4* мы рассматривали механизм создания воспоминаний, например с помощью набора дидактических карточек, на которых была написана информация, которую мы хотели выучить. Однако эти стратегии больше всего полезны для декларативных знаний. Информация, хранящаяся в эксплицитной памяти, требует вашего непосредственного внимания. Например, скорее всего, вам понадобилось время на запоминание того, что цикл `for` на Java пишется как `for (i = 0; i < n; i++){}`. Вы также знали, что вы очень хотите выучить эту информацию. Вот почему это называется эксплицитной памятью — для того чтобы запомнить информацию, вам нужно обращать на нее внимание.

С другой стороны, имплицитная память формируется иначе — повторением. В детстве мы множество раз пытались есть суп ложкой, а спустя какое-то время выучили этот навык. Память о том, как что-то делать, создана не мышлением, а практикой. Именно поэтому это называется имплицитной памятью. При формировании имплицитной памяти нужно пройти три этапа, показанных на рис. 10.4.

Рис. 10.4. Три этапа сохранения информации когнитивный, ассоциативный и автономный

Когнитивный этап

Человек, изучающий что-то новое, находится на когнитивном этапе. Сейчас человеку нужно разделить новую информацию на небольшие фрагменты, а также ему нужно подумать над поставленной задачей.

Например, когда вы учились индексировать список, начинающийся с нуля, вы затрачивали энергию на отслеживание индекса — это показано на левой части рис. 10.4. На когнитивном этапе в вашем мозге формируются или обновляются схемы. Например, когда вы учились индексировать список, начинающийся с нуля, в вашем мозге уже хранилась схема для счета вне области программирования. Только в сохраненной схеме счет начинался с единицы, так что вашему мозгу нужно было просто слегка изменить схему, чтобы можно было начинать счет и с нуля.

Ассоциативный этап

Следующий этап — ассоциативный. На данном этапе вам нужно активно повторять новую информацию до тех пор, пока вы не запомните ее и у вас не сложатся ассоциации. Например, вы видите открывающую скобку и нервничаете, если не видите закрывающую скобку. Вы понимаете, что лучше писать сразу обе скобки — так вы точно не забудете закрывающую скобку. Говоря другими словами, эффективные действия быстро запомнятся, а неэффективные действия будут забыты.

Чем труднее задача, тем больше времени требуется для прохождения ассоциативного этапа. Простые факты запоминаются быстрее. В приведенном ранее примере, когда индекс списка начинался с нуля, спустя какое-то время вы поймете, что для получения правильного индекса вам просто нужно вычесть 1 из номера желаемого элемента списка.

Автономный этап

Наконец вы переходите на автономный этап (иногда его называют процедурным этапом), где навык уже приобретен. Например, вы «на автомате» проводите индексирование списка и всегда делаете это правильно, вне зависимости от контекста или типа данных. Когда вы видите список, операцию со списком, то вы можете сразу сказать нужный индекс, не проводя никаких вычислений.

Когда вы достигаете автономного этапа, мы говорим о том, что вы автоматизировали навык. Вы можете решать задачи без особых усилий, а применение автоматизированного навыка не вызовет когнитивной нагрузки.

Для того чтобы понять всю пользу автоматизации, давайте посмотрим упражнение 10.3. Если вы опытный программист на Java, то вы сразу же заполните пробелы в коде. Шаблон цикла `for` достаточно распространен, так что вы допишете его, совершенно не задумываясь над этим; возможно, вы даже «на автомате» допишете обратный цикл.

УПРАЖНЕНИЕ 10.3. Закончите эти программы на Java, вставив пропущенные участки вместо прочерков __ как можно быстрее:

```
for (int i = ; __ <= 10; i = i + 1) {
  System.out.println(i);
}
```

```
public class FizzBuzz {
    public static void main(String[] args) {
        for (int number = 1; number <= 100; __++) {
            if (number % 15 == 0) {
                System.out.println("FizzBuzz");
            } else if (number % 3 == 0) {
                System.out.println("Fizz");
            } else if (number % 5 == 0) {
                System.out.println("Buzz");
            } else {
                System.out.println(number);
            }
        }
    }
}
```

```
public printReverseWords(String[] args) {
    for (String line : lines) {
        String[] words = line.split("\\s");
        for (int i = words.length - 1; i >= 0; i__)
            System.out.printf("%s ", words[i]);
        System.out.println();
    }
}
```

10.3.2. Почему автоматизация помогает программировать быстрее

Мы можем создать большую базу умений и навыков (скептики назовут их уловками), которую будем постоянно дополнять новыми навыками. Американский психолог Гордон Логан (Gordon Logan) утверждает, что автоматизация происходит путем извлечения воспоминаний из эпизодической памяти, в которой также сохраняются воспоминания об обычной жизни. Выполнение таких задач, как вынесение множителя уравнения за скобки или чтение буквы, создает новое воспоминание, которое представляет собой экземпляр воспоминаний о задаче. Так как каждое воспоминание представляет собой экземпляр некоего базового класса, например класса «воспоминания о вынесении за скобки», теория называется *теорией экземпляров*.

Когда вы сталкиваетесь с похожей задачей, вы можете не думать над ней — в отличие от того, у кого не хватает памяти экземпляров, — вы вспоминаете, как раньше решали похожую задачу, и применяете уже проверенное решение. Согласно Логану, процесс автоматизации завершается в тот момент, когда вы полностью полагаетесь на эпизодическую память и совсем не задумываетесь над задачей. Такое автоматическое выполнение задач происходит очень быстро и без лишних затрат энергии, т. к. извлечение из памяти необходимой информации проходит быстрее, чем активные размышления над задачей. Если вы полностью автоматизируете выполнение задачи, то у вас даже не будет нужды проверять выполненную задачу. Кстати говоря, при осознанном выполнении задачи у вас может появиться желание вернуться к выполненному заданию и перепроверить его на наличие ошибок.

Я уверена, что для многих навыков, необходимых для программирования и решения задач, вы уже находитесь на автономном этапе. Написание цикла `for`, индексация списка или создание класса, скорее всего, уже являются автоматизированными навыками. Следовательно, при программировании эти задачи не будут увеличивать вашу когнитивную нагрузку.

Однако, в зависимости от вашего опыта и навыков, могут встретиться задачи, которые вы еще не довели до автоматизма. Как я уже говорила, при изучении Python я очень долго запоминала цикл `for`. Я даже дидактическими карточками пользовалась! Но, несмотря на практику, каждый раз, когда я садилась за написание кода, я писала `for each` вместо `for`. Мне нужно было изменить имплицитную память. Перед тем как мы подробно рассмотрим стратегии изменения памяти, давайте проверим наши навыки и определим, какие из них можно улучшить.

УПРАЖНЕНИЕ 10.4. Начните новый сеанс программирования. Подумайте о навыках, которые вы используете при программировании. Попытайтесь понять, на каком уровне автоматизации находится каждый из ваших навыков, и запишите полученные результаты в следующую таблицу. Данные вопросы помогут вам определить уровень автоматизации ваших навыков.

- ❒ Нужно ли вам уделять особое внимание навыку? Если да, то вы находитесь на когнитивном этапе.

- Вы можете применить навык, но только полагаясь на стратегии? Если да, то вы находитесь на ассоциативном этапе.
- Вы можете с легкостью применить навык, размышляя при этом над другими задачами? Если да, то вы находитесь на автономном этапе.

Навык	Когнитивный	Ассоциативный	Автономный

10.3.3. Улучшение имплицитной памяти

Теперь, когда мы знаем о трех этапах владения навыками, давайте посмотрим, как можно использовать осознанную практику для улучшения навыков, в которых вы еще не достигли полной автоматизации. Как говорилось в *главе 2*, основная идея осознанной практики состоит в том, чтобы выполнять маленькие задачи очень много раз, пока вы не достигните совершенства. Например, в спортивной интервальной тренировке осознанная практика увеличения скорости — это бег, а в музыке тональная лестница — это осознанная практика расположения пальцев.

В программировании осознанная практика не используется. Когда вы учите создание циклов `for`, то осознанный набор 100 циклов `for` покажется чем-то странным. Однако развитие маленьких навыков поможет вам с легкостью решать большие задачи, т. к. вся когнитивная нагрузка будет направлена на решение этих больших задач.

Осознанная практика может выполняться разными способами. Например, вы можете писать много похожих, но отличающихся друг от друга программ, и практиковать необходимый вам навык. Например, если вы практикуетесь в написании цикла `for`, то напишите много разных циклов `for`: вперед, назад, с использованием переменной-счетчика с разным шагом и т. д..

Когда вы сталкиваетесь со сложной концепцией программирования, вы можете адаптировать другие программы, а не писать их с самого начала. Адаптация кода поможет вам сосредоточить все внимание на отличии новой концепции от уже знакомых концепций. Например, если вы пытаетесь понять список, то сначала вы можете написать несколько программ, в которых вместо списка используются циклы. Потом вы берете код и адаптируете его до тех пор, пока не начнете использовать список. Вы также можете вручную отменять изменения в коде, чтобы посмотреть на разницу. Сравнение различных версий кода и похожесть концепций укрепится в вашей памяти — мы уже наблюдали такой процесс в *главе 3*, когда рассматривали использование дидактических карточек.

Как и в случае с дидактическими карточками, интервальное повторение очень помогает в обучении чему-либо. Каждый день уделяйте немного времени практике и продолжайте повторять до тех пор, пока вы не доведете выполнение задачи до автоматизма. Я хочу еще раз акцентировать ваше внимание на том, что эта стратегия

редко применяется в программировании. И хотя сначала она может показаться вам странной, продолжайте пытаться. Чем-то это похоже на поднятие тяжестей: каждое повторение делает вас и вашу память чуть сильнее.

10.4. Обучение на основе кода и его объяснения

В этой главе мы узнали, что не существует общих навыков решения задач и что если просто много программировать, это не поможет нам лучше решать задачи. Мы также узнали, что вы можете использовать осознанную практику для улучшения навыков программирования. И хотя довести небольшие навыки программирования до автоматизма необходимо, этого недостаточно для решения больших задач.

Вторая стратегия, которой вы можете воспользоваться для улучшения навыка решения задач, — это осознанное изучение того, как другие люди решают задачи. Знание того, как другие решают задачи, часто называют *примерами с решением*.

Австралийский психолог Джон Свеллер (John Sweller), сформулировавший теорию когнитивной нагрузки, описанной в *главе 4*, исследовал важность зависящих от предметной области стратегий для способностей к решению задач.

Свеллер обучал маленьких детей математике путем решения алгебраических уравнений. Однако вскоре его ждало разочарование: решая классические задачи по алгебре, дети получили не так много новых знаний. В тот момент Свеллер заинтересовался экспериментами со стратегией обучения решению задач. Для того чтобы получить более подробную информацию, Свеллер в 1980-х годах сам провел несколько исследований. В его исследованиях принимали участие 20 девятиклассников (дети в возрасте от 14 до 15 лет) из австралийской средней школы. Он разделил учеников на две группы; каждой группе предложили решить алгебраические уравнения, например: «a = 7 – 4a; найдите значение a».

Однако между группами была разница, показанная на рис. 10.5. Обе группы решали одни и те же уравнения, но вторая группа решала уравнения без дополнительной помощи (уверена, вы так же решали уравнения в средней школе). Первой группе дали не только уравнения, но и *примеры с решением*, в которых подробно описывался метод решения уравнений. Вы можете думать о примерах с решением как о рецептах, в которых подробно описаны шаги, необходимые для решения уравнений.

После того как обе группы детей решили уравнения, Свеллер и Купер сравнили их результаты. Конечно, неудивительно то, что первая группа решила уравнения быстрее второй группы, т. к. у них был пример решения задачи. Первая группа решала уравнения примерно в пять раз быстрее второй группы.

Свеллер и Купер также проверили показатели обеих групп по разным задачам, т. к. именно из-за этого некоторые люди не хотят применять «рецепты» в обучении; смысл в том, что слепое следование шагам из рецепта не имеет практической поль-

зы и не дает научиться чем-то новому. И вот интересный результат: дети из первой группы успешнее справлялись с разными задачами, при решении которых можно было использовать правила вычисления (т. е. шаги из рецепта): например, вычитание одного и того же значения из обеих частей уравнения или деление обеих сторон на одно и то же число.

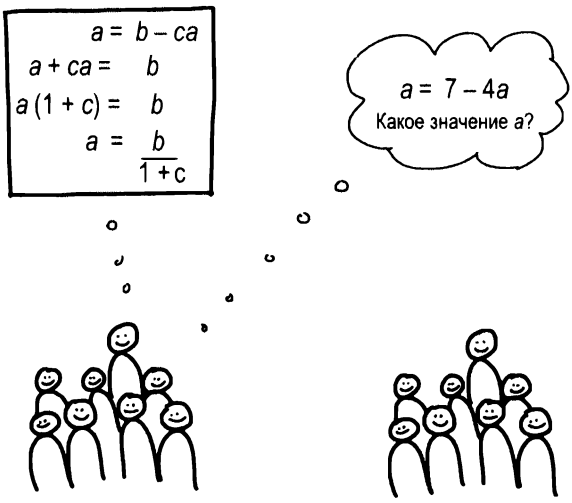


Рис. 10.5. Обе группы решали одни и те же уравнения, но группе с левой стороны были даны примеры с решением, которые показывали ученикам, как решать такие уравнения

Данный результат был подтвержден многими исследованиями среди разных возрастов и предметов, включая математику, музыку, шахматы, спорт и программирование.

10.4.1. Новый вид когнитивной нагрузки: соответствующая нагрузка

Для многих опытных программистов результаты исследования Свеллера могли показаться неожиданными. Мы часто думаем так: если нам надо, чтобы дети хорошо справлялись с задачами, нужно просто позволить им решать их; если мы хотим стать хорошими программистами, то нам просто нужно много программировать. Однако результаты исследования Свеллера показывают, что это не так. Давайте подробнее рассмотрим, почему наши мысли неверны. Почему группа, получившая примеры решения уравнений, справилась с заданиями лучше, чем группа, которая решала задачи самостоятельно? Объяснение, данное Свеллером, касается влияния когнитивной нагрузки на рабочую память.

Мы уже знаем, что рабочая память — это кратковременная память, участвующая в решении определенной задачи. Следовательно, хотя кратковременная память и может хранить от двух до шести элементов, что уже объяснялось в *главе 2*, рабочая память может обрабатывать от двух до шести слотов памяти, которые доступны для хранения информации.

Мы также знаем, что когда рабочая память заполнена, то она не может правильно функционировать. При заполненной рабочей памяти мозг также не может передавать информацию из рабочей памяти в долговременную память. Это показано на рис. 10.6.

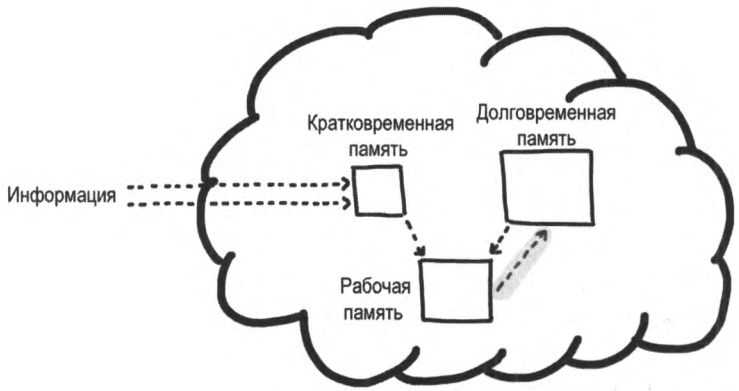


Рис. 10.6. Соответствующая нагрузка необходима для активации выделенной стрелки и передачи информации в долговременную память. Если рабочая память переполнена (т. е испытывает высокую нагрузку), то информация в долговременной памяти сохраняться не будет

Мы рассмотрели две формы когнитивной нагрузки: внутренняя нагрузка, вызванная самой проблемой, и внешняя нагрузка, вызванная формулировкой проблемы. Однако существует третья форма когнитивной нагрузки — это соответствующая нагрузка.

Соответствующая нагрузка (означает что-то вроде необходимой нагрузки) представляет собой процесс, с помощью которого мозг передает информацию обратно в долговременную память. Когда вся когнитивная нагрузка состоит из внутренней и внешней нагрузки, то места для соответствующей нагрузки не остается; говоря другими словами, вы не сможете вспомнить уже решенные задачи и способы, которыми вы их решали.

Именно поэтому после напряженного сеанса кодирования вы не можете вспомнить, что именно вы делали. Ваш мозг был настолько занят, что не мог запомнить ваши действия.

Теперь, когда вы знакомы с тремя формами когнитивной нагрузки, давайте снова обратимся к эксперименту с австралийскими девятиклассниками. Теперь мы понимаем, почему группа, пользовавшаяся примерами с решениями легче решала новые уравнения: т. к. их когнитивная нагрузка была низкой, они могли запоминать примеры решений и применять их на практике. Они узнали, что при решении уравнений нужно одну часть уравнения перенести на другую сторону от знака равенства, что при переносе знак «плюс» меняется на знак «минус» и что они всегда могут поделить обе части уравнения на одно и то же число.

Навыки, которые ученики приобрели во время решения уравнений с помощью примеров с решениями, пригодятся при решении практически всех задач алгебры, так что ученики из первой группы могут применять полученные знания при реше-

нии новых уравнений. Ученики из второй группы были больше сосредоточены на возникших сложностях, а не общих правилах решения таких уравнений.

Люди, которых волнует обучение с помощью примеров с решением, считают иначе. Я согласна с тем, что это звучит логично: если мы хотим, чтобы дети научились решать задачи, то они должны просто решать задачи. Такого образа мышления придерживаются и программисты: если мы хотим быть хорошими программистами, то нужно много практиковаться! Нужно создавать программы, пробовать что-то новое — и вы научитесь. Но, видимо, все это неправда.

Хотя исследования Свеллера сосредоточены на обучении математике, проводились аналогичные исследования и по обучению программированию. Исследования показали похожие результаты: дети получают больше пользы из чтения программ и их объяснений, а не из программирования[1]. Как говорит нидерландский психолог Пол Киршнер (Paul Kirschner): «Если вы будете делать что-то профессиональное, то это не сделает вас профессионалом».

10.4.2. Примеры с решением на практике

Мы уже знаем, что изучение кода и процесса его написания позволяет улучшить навыки программирования. Есть несколько источников, которые могут вам помочь изучать код.

Работайте вместе с коллегой

Во-первых, вы не обязаны изучать код в одиночку; намного больше пользы будет от совместной работы над кодом. Вы можете создать кружок читателей кода или работать с коллегами, которые в этом заинтересованы. Если вы делаете это с кем-то, то вы намного быстрее привьете себе привычку регулярно читать код (**https://code-reading.org/**). Если вы состоите в кружке, то вы можете обмениваться кодом, его объяснениями и учиться друг у друга.

В *главе 5* мы рассмотрели стратегии понимания кода, в том числе и резюмирование (создание краткого пересказа кода). Вы можете изучать собственный код и пересказ кода, однако больше пользы вы получите в том случае, если вы сначала резюмируете свой код, а затем обменяетесь данными с коллегой.

Используйте GitHub

Если вы занимаетесь практикой чтения кода в одиночку, то, к счастью, в Интернете есть множество исходных кодов и документации. Например, начать читать код можно с GitHub. Просто откройте архив кода, который вам знаком, например библиотеку, с которой вы работаете, и начните читать код. Будет лучше, если вы выберите код, предметная область которого вам хоть немного знакома. В выбранном вами коде не должно быть много незнакомых слов, понятий, которые вызовут

---

[1] «The case for case studies of programming problems» Communications of the ACM, часть 35, no. 3, 1992, Марсия Линн и Майкл Клэнси, **https://dl.acm.org/doi/10.1145/131295.131301**.

внешнюю когнитивную нагрузку, — так вы сможете полностью сосредоточиться на программировании.

Читайте книги или блоги об исходном коде

В блогах можно найти множество постов, в которых люди рассказывают о своем опыте решения задач программирования. Вы можете вооружиться знаниями других людей и применять их в программировании. Есть также несколько книг, в которых рассказывается о коде: например, две части книги *«Архитектура open-source-приложений»* Эми Браун (Amy Brown) и Грега Уилсона (Greg Wilson) или *«500 Lines or Less»* Эми Браун и Майкла ДиБернардо (Michael DiBernardo).

Выводы

- Многие программисты считают, что навык решения задач — это общий навык, но на самом деле это не так. Знания в области программирования влияют на скорость решения задач программирования.
- Долговременная память делится на несколько видов памяти, которые играют разную роль в решении задач — это имплицитная и эксплицитная память. Имплицитная память — это «мышечная память», или задачи, которые вы можете выполнять автоматически, например печать вслепую. Эксплицитная память хранит информацию, которую вы точно знаете и которую вам часто нужно вспоминать, например синтаксис цикла `for`.
- Для укрепления имплицитной памяти, связанной с программированием, автоматизируйте нужные навыки: например, печатайте вслепую или запоминайте сочетания клавиш.
- Для укрепления эксплицитной памяти, связанной с программированием, подробно рассмотрите существующий код, а также его объяснение и то, как его писали.

Часть IV

О совместной работе над кодом

Пока что в этой книге мы рассматривали только работу отдельного программиста, но на самом деле все ПО разрабатывается командами. В заключительной части мы рассмотрим, как перейти в потоковое состояние, когда вы можете писать код, не отвлекаясь на своих коллег. Мы также рассмотрим стратегии создания больших систем, в которых легко могут начать работать другие люди, а также поговорим о процессе адаптации новых работников.

11 Процесс написания кода

В этой главе:

- сравним различные активности, которые выполняются людьми во время работы с кодом;
- узнаем, как можно помочь мозгу продуктивно выполнять активности;
- узнаем, как отвлечение от работы влияет на программиста;
- поймем, как память помогает быстро вернуться к работе.

До сих пор мы рассматривали, какие когнитивные процессы играют роль при чтении и написании кода. В предыдущих главах мы рассмотрели стратегии написания легкого для понимания кода, а также то, как лучше всего решать задачи.

В этой главе мы отойдем от темы кода и рассмотрим, какие когнитивные процессы задействованы во время программирования. Во-первых, мы попытаемся разобраться в том, что мы имеем в виду, говоря, что кто-то программирует. Мы рассмотрим различные активности, которые составляют процесс программирования, а также исследуем то, как лучше всего поддерживать эти активности.

Во-вторых, мы рассмотрим когнитивные последствия самого страшного испытания в жизни программиста — отвлечения от работы. Выясним, почему отвлечение от программирования так раздражает, и узнаем, что можно сделать для того, чтобы снизить его негативный эффект. К концу этой главы вы научитесь поддерживать работоспособность в программировании и узнаете, как справляться с последствиями отвлечения.

11.1. Различные активности, выполняемые во время программирования

Когда вы занимаетесь программированием, вы делаете множество разных вещей. Впервые эти типы активностей были описаны британскими исследователями Томасом Грином (Thomas Green), Аланом Блэквеллом (Alan Blackwell) и Мариан Петре (Marian Petre) в их концепции когнитивных измерений, которая используется для оценки когнитивного воздействия языка программирования или базы кода (их работу мы рассмотрим более подробно в *главе 12*). Авторы выделяют пять типов активностей: поиск, осмысление, переписывание, исследование и наращивание.

На рис. 11.1 представлены все пять типов активностей, и задачи, которые вы, скорее всего, выполняете во время программирования. На рисунке также показано, из-за чего каждая из активностей может казаться сложной.

Активность			Задача			В чем сложность
	Выполнение	Кодирование	Проверка	Чтение	Рефакторинг	
Поиск	✓			✓		Кратковременная память
Осмысление	✓		✓	✓	✓	Рабочая память
Переписывание		✓				Долговременная память
Исследование	✓	✓	✓	✓	✓	Все три
Наращивание	✓	✓	✓	✓	✓	Все три

Рис. 11.1. Обзор активностей и систем памяти, которые при этом используются

11.1.1. Поиск

Поиск в коде — это активность, когда вы просматриваете кодовую базу в поисках нужной информации. Например, вы можете искать точное место ошибки, все места вызова определенного метода или место, где должна инициализироваться переменная.

В процессе поиска большую часть времени вы читаете и выполняете код, в основном используя точки останова и отладчик, а также реагируете на выводимые во время выполнения кода сообщения. Поиск очень сильно сказывается на вашей кратковременной памяти. Вам всегда нужно помнить, что вы ищете, какие части кода вы уже просмотрели и почему, а также что еще нужно найти. Следовательно, данная активность лучше всего поддерживается с помощью заметок, куда вы можете перенести часть воспоминаний — просто создайте отдельный документ или выписывайте необходимую вам информацию на листок бумаги. Указывайте, какую информацию вам надо найти, какую информацию вы уже нашли и где вы собираетесь искать дальше.

Как уже обсуждалось ранее, иногда следует внести в код временные изменения — таким образом вы облегчите выполнение поставленной задачи. Когда вы ищете что-то в коде, полезно оставлять небольшие комментарии у каждого фрагмента, в которых описывается, почему вы смотрели именно этот фрагмент кода. Например, заметка вроде «Я прочитала этот метод, т. к. подумала, что он может участвовать в инициализации класса страницы» может помочь вам, когда вы повторно обратитесь к этому фрагменту. И заметки особенно помогут вам, если вы понимаете, что не сможете выполнить поиск полностью, не прерываясь, т. к. вот-вот начнется совещание или наступит конец рабочего дня. Отметка места, на котором вы остановились, поможет вам через какое-то время продолжить поиск и закончить его.

11.1.2. Осмысление

Осмысление кода — это активность, когда вы читаете и выполняете код с целью понять его смысл и функциональные возможности. Эта активность похожа на поиск, однако на данном этапе у вас нет полного понимания того, что именно делает код, потому что вы написали этот код много лет назад или этот код написал другой программист.

Как мы видели в *главе 5*, программисты тратят около 58% рабочего времени на осмысление исходного кода, так что эта активность встречается довольно часто в вашей повседневной жизни.

Кроме чтения и выполнения кода осмысление может включать в себя и использование тестовых программ — так вы сможете лучше понять, как именно должен работать код. Как мы обсуждали в *главе 4*, осмысление может включать в себя и рефакторинг, который может упростить код.

Рефакторинг кода тесно связан с осмыслением из-за того, что осмысление — это активность, которая негативно влияет на рабочую память. Вам нужно рассуждать о коде, который вы не совсем понимаете. Следовательно, поддержка рабочей памяти — это лучшее, что вы можете сделать на данном этапе. Попробуйте начертить модель кода и корректируйте ее каждый раз, когда узнаете что-то новое. Так вы будете получать информацию из внешнего источника, а не из собственного мозга, — следовательно, снизится нагрузка на вашу память. Модель кода также поможет вам понять, есть ли у вас неверные представления о коде. К тому же если вы не можете за один раз закончить этап осмысления, то сделанные вами заметки и модель помогут вам быстро вернуться к работе.

11.1.3. Переписывание

Переписывание — это активность, где вы просто занимаетесь кодированием. На этом этапе вы придерживаетесь определенного плана по изменению кодовой базы. По правде говоря, переписывание включает в себя только кодирование.

При переписывании задействована долговременная память, т. к. вам постоянно нужно вспоминать синтаксические конструкции.

11.1.4. Наращивание

Наращивание включает в себя поиск, осмысление и переписывание. Когда вы расширяете базу кода, вы добавляете новый функционал, что включает в себя поиск мест(а), куда добавить код, а также осмысление уже существующего кода, чтобы знать, куда именно нужно добавить код, а затем воплощение идеи в синтаксисе, т. е. переписывание.

Так как наращивание включает в себя несколько действий, то данный этап может быть труден для всех трех видов памяти. Поэтому задачи наращивания, с которыми постоянно имеют дело опытные программисты, нуждаются в поддержке памяти в форме заметок, а также в рефакторинге, чтобы код стал легче для осмысления.

Ваш опыт работы с определенным языком программирования и базой кода влияет на то, какая память будет принимать наибольшее участие в активности. С одной стороны, если вы хорошо знаете язык программирования, то ваша долговременная память не будет напрягаться над вспоминанием синтаксиса. С другой стороны, если вы знаете кодовую базу, то ваша рабочая память и кратковременная память могут не испытывать нагрузки на этапе поиска и осмысления кода.

Однако если база кода или язык (или все вместе) вам не знакомы, то это может вызвать дополнительную нагрузку. Постарайтесь разделить задачу наращивания на несколько небольших задач. Размышление над тем, какую именно задачу вы выполняете, помогает вам поддерживать нужную память. Решите для себя, что свою работу вы начнете с поиска информации, затем вы осмыслите ее и добавите необходимый код.

11.1.5. Исследование

Последняя активность, которая рассматривается Грином, Блэквеллом и Петре, — это исследование кода. При исследовании вы по существу делаете «наброски» кода. Вы можете иметь приблизительное представление того, над чем вы будете работать дальше, но в процессе программирования вы все больше будете узнавать о предметной области задачи и конструкциях, которые вам нужно использовать в коде.

Как и приращение, исследование включает в себя несколько задач программирования: написание кода, выполнение кода, тестирование кода для того, чтобы проверить, в правильном ли направлении мы движемся, чтение кода и, возможно, рефакторинг, чтобы изменить код под свои нужды.

Во время исследования вы в значительной степени полагаетесь на средства интерактивной среды разработки, например выполняете проверку кода после внесенных изменений, используете средства автоматизации рефакторинга или применяете «поиск зависимостей» для быстрой навигации по коду.

Так как исследование зависит от других активностей, данный этап сложен для всех видов памяти, особенно для рабочей памяти, потому что вы составляете планы и цели сразу во время программирования. Вам может показаться, что документирование ваших планов негативно скажется на вашей эффективности, но я все же рекомендую делать некоторые заметки о ваших целях и проектных решениях. Та-

ким образом вы сможете освободить место в рабочей памяти для более глубокого обдумывания задачи.

11.1.6. А как же отладка?

При обсуждении данной концепции разработчики часто задают вопрос: почему среди активностей нет отладки? Ответ прост: обычно при отладке задействованы сразу все пять активностей. Отладка ведет к исправлению ошибки, но иногда перед тем, как исправить ее, вы должны выполнить поиск местоположения ошибки.

Соответственно отладка представляет собой последовательность исследования, поиска и осмысления, а затем написания кода. Очень часто отладку можно описать как сочетание всех пяти активностей.

УПРАЖНЕНИЕ 11.1. В следующий раз, когда вы будете заниматься программированием, подумайте о пяти активностях из концепции когнитивных измерений. Какая активность занимает больше времени? С какими препятствиями вы столкнулись при работе с кодом?

Активность	Задача	Затраченное время
Поиск		
Осмысление		
Переписывание		
Исследование		
Наращивание		

11.2. Программист отвлекся

Сегодня множество программистов работают в офисах открытого типа, где отвлекающие факторы — вещь обыденная. Но как отвлечение сказывается на мозге и нашей продуктивности? Рини ван Солиген (Rini van Solingen), профессор Делфтского технического университета в Нидерландах, еще в середине 90-х годов изучал влияние отвлечения на работу программиста.

Ван Солиген провел исследование, в котором он изучал две организации. Исследование показало похожие результаты в обеих организациях. Он обнаружил, что отвлечение — это обычная вещь в «жизни» программиста, и каждое отвлечение занимает от 15 до 20 мин. Около 20% всего рабочего времени программист отвлекается, а с ростом использования Slack и других мессенджеров отвлечение становится распространенным явлением[1].

[1] «Interrupts: Just a Minute Never Is», *IEEE Software*, ч. 15, № 5, Рини ван Солиген и др., **https://ieeexplore.ieee.org/document/714843**.

Существуют и более свежие исследования. Например, Крис Парнин, чью работу мы рассмотрели в *главе 3*, также исследовал отвлечение: он записал 10 000 сеансов программирования 86 программистов. Результаты этого исследования подтвердили вывод ван Солигена касаемо того, что прерывания уже стали частью рабочей жизни программиста. Согласно исследованию Парнина[2], средний программист без отвлечений работает только два часа в день. Программисты также согласны с тем, что отвлечение — это проблема. Исследование Microsoft показало, что примерно 62% программистов считают возвращение к работе после отвлечения большой проблемой[3].

11.2.1. Задачи программирования нуждаются в «разогреве»

В *главе 9* мы рассмотрели аппарат fNIRS как способ измерения когнитивной нагрузки. С помощью данного аппарата мы узнали не только то, какие типы кода вызывают нагрузку, но также и то, на какие задачи приходится больше всего нагрузки.

В 2014 году Такао Накагава, исследователь из Института науки и технологий Нары в Японии, измерил активность мозга с помощью аппарата fNIRS[4]. Участников попросили прочитать два варианта алгоритма, написанных на C. Первый вариант был обычным, в то время как второй вариант был специально усложнен. Например, были заменены счетчики циклов и другие значения, из-за чего переменные менялись часто и нерегулярно. Данные изменения не повлияли на функциональные возможности программы.

Накагава получил два интересных результата. Во-первых, у 9 из 10 участников во время выполнения задания когнитивная нагрузка не была постоянной величиной. Процесс программирования сложный не все время: какие-то моменты вызывают трудности, а другие делаются за пару мгновений. Во-вторых, исследователи отслеживали место в коде, где у участника максимально повышался кровоток и соответственно когнитивная нагрузка. Результаты показали, что именно в середине задания наблюдалась самая высокая когнитивная нагрузка.

Результаты исследования Накагавы указывают на то, что в задачах на осмысление кода есть своего рода фаза разогрева и фаза охлаждения, между которыми работа выполняется максимально интенсивно. Опытные программисты, вероятно, вспомнят это время разогрева, необходимое для того, чтобы «войти в форму», нужную для построения воображаемой модели кода и переписывания кода. Как уже обсуж-

---

[2] «Resumption Strategies for Interrupted Programming Tasks», Крис Парнин и Спенсер Ругабер, **https://ieeexplore.ieee.org/document/5090030**.

[3] «Maintaining Mental Models: A Study of Developer Work Habits», Томас ЛаТоза и др., **https://dl.acm.org/doi/10.1145/1134285.1134355**.

[4] «Quantifying Programmers' Mental Workload during Program Comprehension Based on Cerebral Blood Flow Measurement: A Controlled Experiment», Такао Накагава и др., **https://posl.ait.kyushu-u.ac.jp/~kamei/publications/Nakagawa_ICSENier2014.pdf**.

далось ранее в этой главе, это может помочь выделить подзадачи в большой задаче программирования.

11.2.2. Что происходит после отвлечения

Парнин также исследовал, что происходит после отвлечения от работы, и результат его нисколько не удивил — отвлечение сильно снижает эффективность. После прерывания программисту требуется около 15 мин, чтобы вернуться к работе. Если программиста отвлекают на этапе редактирования метода, то сразу же возобновить работу менее чем за минуту удается только в 10% случаев.

Что программисты делают для того, чтобы вернуться к работе над кодом? Согласно результатам Парнина, рабочая память теряет важную информацию о коде, над которым работает программист. Участники его исследования должны были приложить усилия для того, чтобы восстановить контекст. Для этого многие из участников просматривали несколько мест кода и только потом продолжали программировать. Некоторые из участников специально вставляли в код мелкие ошибки, например случайные символы, из-за которых происходила ошибка компиляции. Это служило тем, что Парнин назвал *указателем путевого заграждения*, — своего рода гарантией того, что код будет доведен до конца, а не останется в наполовину завершенном состоянии. Некоторые участники, чтобы вернуться к работе, также искали разницу между рабочей версией кода и исходной.

11.2.3. Как подготовиться к отвлечению

Теперь, когда мы знаем, что отвлечение — это дело обычное и его очень редко можно избежать, давайте посмотрим, что происходит в вашем мозге, когда вас отвлекают. Так мы сможем лучше подготовиться к отвлечению. В этой главе мы рассмотрим три стратегии, которые помогут нам справиться с отвлечением.

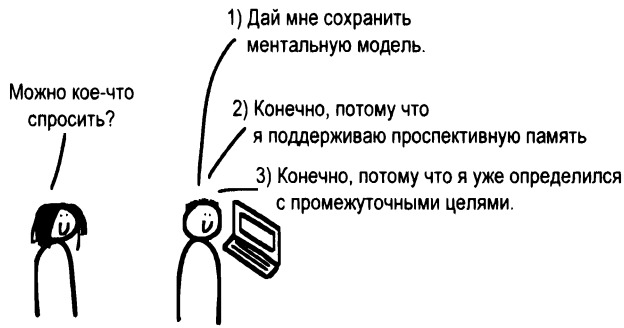


Рис. 11.2. Три способа возвращения к работе

Сохраняйте воображаемую модель

Ранее мы уже обсуждали стратегии, которые можно использовать для поддержки рабочей и кратковременной памяти: например, создание заметок и комментариев,

визуализация моделей и рефакторинг кода. Данные стратегии могут быть полезны и для возвращения к работе после перерыва.

Результаты исследования Накагавы показали, что на фазе разогрева программист чаще всего создает воображаемую (ментальную) модель кода. Если фрагменты модели находятся вне кода, то тогда вы быстро сможете восстановить всю модель. Также полезно писать заметки о модели в комментариях кода.

Некоторые программисты не одобряют написание подробных комментариев, т. к. вся нужная информация должна находиться в коде. Однако очень редко в коде можно отследить когнитивную деятельность программиста, из-за чего чаще всего создается неправильная воображаемая модель автора кода. Мы не привыкли объяснять в коде то, почему был выбран определенный подход, каковы наши цели или какие варианты рассматривались для реализации кода. Но когда подобные мысли нигде не записаны, то вы можете прийти к ним лишь случайно; в любом случае, это довольно долгий и сложный процесс. Джон Оустерхаут (John Ousterhout) описывает данный момент в своей книге «*The Philosophy of Software Design*» (Yaknyam Press, 2018) так: «Общая идея комментариев заключается в хранении информации, о которой думал программист, но которая не была отражена в самом коде».

Кроме того, что заметки очень полезны для людей, читающих код, документирование решений может быть полезным также для временного хранения вашей воображаемой модели, что значительно упрощает возвращение к работе после долгого перерыва. Фред Брукс в своей книге «*Мифический человеко-месяц*» пишет, что комментарии очень важны на этапе осмысления программы, потому что они всегда под рукой. И хотя комментарии на листке или в документе очень полезны, поиск нужных документов, когда вы возвращаетесь к работе, может привести к тому, что вы будете пытаться запомнить лишнюю информацию.

Когда вас отвлекают, а у вас есть возможность не отвечать какое-то время, например когда вам приходит сообщение в Slack или к вам подходит коллега, который может немного подождать, то бывает полезно «разгрузить память» в комментарии в коде. Конечно, это не всегда помогает, но лучше всего подстраховаться.

Помогите своей проспективной памяти

Для того чтобы разобраться со второй стратегией, давайте чуть подробнее рассмотрим различные виды памяти. В *главе 10* мы рассмотрели два вида памяти: процедурную и декларативную, или имплицитную и эксплицитную.

Еще один вид памяти, но который касается будущего, а не прошлого, называется *проспективной памятью* — памятью о запланированных в будущем действиях. Данный вид памяти тесно связан с планированием и решением задач. Когда вы говорите себе, что нужно не забыть по дороге домой купить молоко, или когда вы напоминаете себе, что надо будет сделать рефакторинг этого ужасного кода, вы используете проспективную память.

Тема поддержки проспективной памяти освещалась в нескольких исследованиях. В них описывается, как программисты пытаются справиться с плохой проспективной памятью. Например, программисты часто добавляют к фрагменту, над которым

они работают, заметки to-do, которые потом напомнят им о том, что фрагмент кода нужно закончить или как-то улучшить[5]. Естественно, и многие программисты это могут подтвердить, к подобным комментариям часто не возвращаются и не заканчивают то, что в них написано. На рис. 11.3 показана история поиска на GitHub, из которого видно, что код со словом to-do запрашивался 136 миллионов раз.

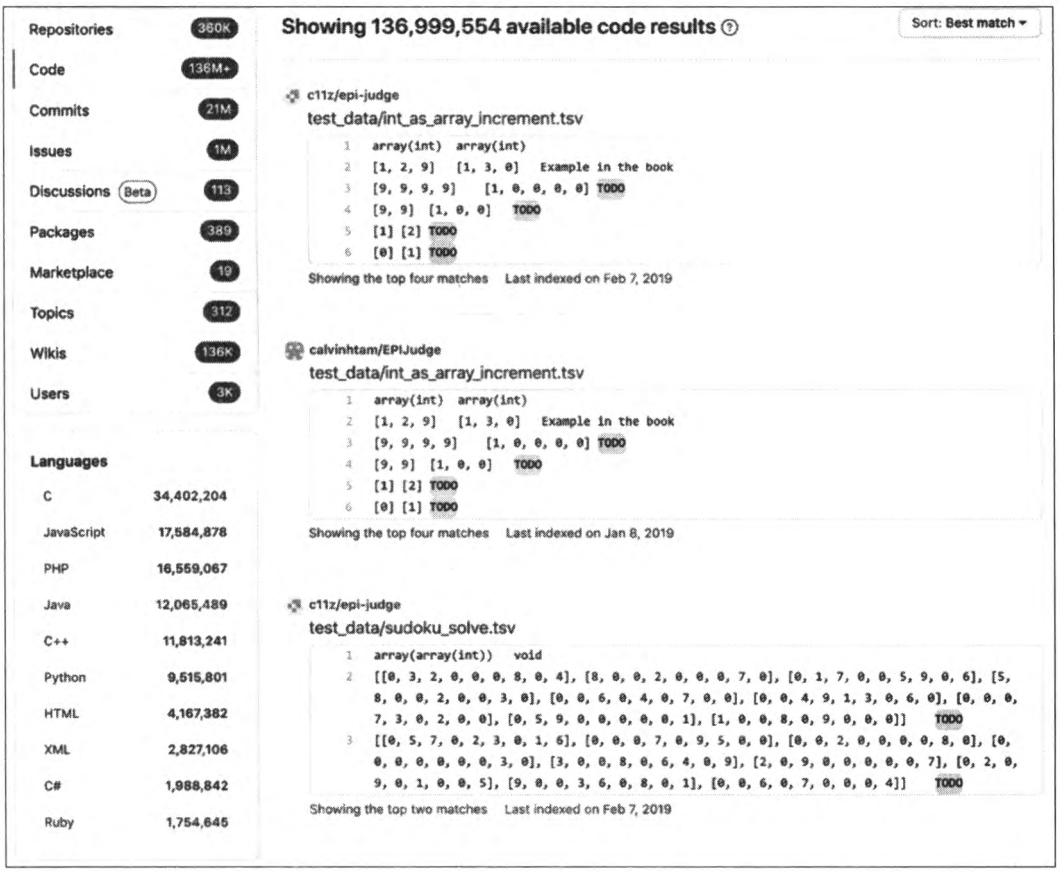


Рис. 11.3. Заметки to-do могут оставаться в коде невыполненными

Кроме заметок to-do и умышленных ошибок компиляции программисты также применяют стратегии, которыми пользуются другие офисные работники: на рабочий стол они вешают стикеры и отправляют письма на свою электронную почту. Конечно, стикеры и электронные письма не имеют никакого отношения к кодовой базе, но они тоже могут быть очень полезны.

Парнин также разработал плагин для Visual Studio, который позволяет программистам поддерживать проспективную память в тот момент, когда им нужно прервать

[5] «TODO or to Bug: Exploring How Task Annotations Play a Role in the Work Practices of Software Developers», Маргарет Энн Стори и др., **https://dx.doi.org/doi:10.1145/1368088.1368123**.

работу[6]. Например, плагин позволяет добавлять элементы to-do в код и указывать дату, до которой их нужно выполнить.

Определитесь с промежуточными целями

Третья стратегия, которая поможет вам быстро вернуться к работе, называется определением промежуточных целей. Данная стратегия подразумевает, что вы будете записывать то, на какие небольшие этапы можно разбить задачу. Например, при синтаксическом анализе и реструктуризации текста можно выделить такие этапы:

1. Разобрать текст в дерево синтаксического анализа.
2. Отсортировать дерево синтаксического анализа.
3. Преобразовать дерево синтаксического анализа обратно в текст.

Эти этапы легко запомнить, но если вас часто отвлекают от работы, то у вас могут возникнуть трудности с возвращением к работе над кодом и осознанием, что вы собирались делать. Если я работаю с большой задачей, то сначала я пытаюсь написать все этапы в виде комментариев в моем коде, например:

```
# выполнить синтаксический анализ текста
# получить дерево синтаксического анализа
# отсортировать дерево синтаксического анализа и
# преобразовать дерево обратно в текстовый формат
```

Следовательно, я могу писать такие заметки в каждом фрагменте кода; в таком случае у меня всегда есть план, которого я буду придерживаться. Исследование программистов, проведенное Лорен Маргуле (Lauren Margulieux), профессором кафедры образовательных наук Университета штата Джорджия, показало, что предлагаемые вами промежуточные цели программисты используют для создания воображаемой модели решения задачи[7].

Промежуточные цели полезны для организации собственных мыслей после длительного отвлечения, однако они могут быть полезны и в других ситуациях. Например, некоторые промежуточные цели, написанные в самом коде, могут так и остаться в виде комментариев и стать частью документации кода. Промежуточные цели также могут использоваться при совместной работе, когда опытный программист разрабатывает промежуточные цели, а его подчиненные программисты выполняют их.

11.2.4. Когда отвлекать программиста

В *главе 9* мы рассмотрели различные способы измерения когнитивной нагрузки, с которой сталкиваются люди. Но есть и другие способы определения когнитивной нагрузки, которую вызывает та или иная задача.

---

[6] **https://marketplace.visualstudio.com/items?itemName=chrisparnin.attachables**

[7] «Subgoal-Labeled Instructional Material Improves Performance and Transfer in Learning to Develop Mobile Applications», Лорен Маргуле и др., **https://doi.org/10.1145/2361276.2361291**

Один из примеров — это *измерение скорости выбора при выполнении двух совмещенных действий.* Выполнение двух совмещенных действий — это две задачи, одна из которых выполняется в тот момент, когда человек решает исходную задачу. Например, при решении уравнения на экране в случайное время появляется буква А. Когда появляется эта буква, участник должен как можно быстрее нажать клавишу с этой буквой. То, насколько быстро и точно участник выполнит вторую задачу, является показателем того, какую когнитивную нагрузку испытывает человек. Исследователи считают, что измерение скорости выбора при выполнении двух совмещенных действий может быть хорошим способом оценки когнитивной нагрузки. Как вы уже могли догадаться, у данного способа есть свои недостатки: вторая задача тоже вызывает когнитивную нагрузку, отвлекая от исходной задачи.

Используя такой способ измерения, исследователи изучили связь между когнитивной нагрузкой и отвлечениями. Брайан Бэйли (Brian P. Bailey), профессор компьютерных наук Иллинойского университета, большую часть своей карьеры посвятил исследованию причин и последствий отвлечения.

В исследовании, проведенном в 2001 году[8], приняло участие 50 человек. Участникам предложили выполнить задание, во время которого их будут отвлекать. В эксперименте использовались разные типы задач, от подсчета частоты появления слов в сетке до чтения текста и ответов на вопросы. При выполнении этих задач участников отвлекали нерелевантной информацией, например показывали заголовки последних новостей или обновления фондового рынка. Это исследование проводилось как эксперимент с двумя группами участников. Одну группу постоянно отвлекали, пока участники выполняли основную задачу, а другую группу отвлекали после выполнения задачи.

И хотя результаты исследования Бэйли не оказались чем-то удивительным, они помогают понять отвлечение. Бэйли обнаружил, что выполнение задачи, когда тебя отвлекают, занимает намного больше времени, чем выполнение задачи, когда тебя не отвлекают. Он также выяснил, что людям, которых постоянно отвлекают, задания кажутся более сложными.

Бэйли измерял не только время и сложность задач, но и исследовал эмоциональное состояние участников во время всего исследования. Участники отвечали на вопросы, в которых их просили определить уровень их раздражения и беспокойства каждый раз, когда появлялась отвлекающая задача. Результаты опроса показали, что уровень раздражения зависит от отображения прерывающей задачи. Участники группы, которые отвлекались во время выполнения основной задачи, были раздражены больше, чем участники второй группы. То же самое касается и уровня беспокойства. Участники, которых отвлекали, испытывали больше стресса. Повторное исследование 2006 года, в котором использовалась похожая схема, показало, что участники, которых отвлекали, совершали в два раза больше ошибок[9].

---

[8] «The Effects of Interruptions on Task Performance, Annoyance, and Anxiety in the User Interface», Брайан Бэйли и др., **http://mng.bz/G6KJ**.

[9] «On the Need for Attention-Aware Systems: Measuring Effects of Interruptions on Task Performance, Error Rate, and Affective State», Computers in Human Behavior, ч. 22, № 4, стр. 685–708, 2006, Брайан Бэйли, **http://mng.bz/zGAA**.

Основываясь на этих результатах, мы могли бы сделать вывод, что программистам, которым постоянно приходится отвлекаться, может быть полезнее отвлекаться в более подходящее время (например, после выполнения основной задачи). Основываясь на этой идее, Мануэла Цюгер (Manuela Züger), в то время аспирант Цюрихского университета, разработала интерактивный светильник FlowLight.

FlowLight (**https://emea.embrava.com/pages/flow**) — это аппаратный источник света, который программист размещает на рабочем столе или возле экрана компьютера. На основе действий с компьютером, например скорости набора или щелчков мышью, FlowLight определяет, насколько глубоко программист погружен в задачу и испытывает ли он высокую когнитивную нагрузку. Обычно такое состояние называют «в ударе» или «в потоке». Если программист находится «в ударе» и его нельзя отвлекать, то на FlowLight мигает красный индикатор. При меньшей активности красный индикатор FlowLight постоянно светится. Если разработчика можно отвлечь, то на FlowLight будет светиться зеленый индикатор.

Цюгер опробовала FlowLight в большом исследовании с более чем 400 участниками из 12 стран и обнаружила, что FlowLight помогает сократить количество отвлечений на 46%. Многие участники исследования продолжили использовать FlowLight даже после окончания исследования. FlowLight также доступен для покупки[10].

11.2.5. Пара слов о многозадачности

Читая об отвлечении, вы могли задуматься о многозадачности. Неужели отвлечение настолько плохо? Разве наш мозг не может выполнять одновременно несколько задач, подобно многоядерному процессору?

Многозадачность и автоматизация

К сожалению, есть доказательства того, что люди не могут поддерживать многозадачность во время решения сложных когнитивных задач. Возможно, этот факт кажется вам весьма неубедительным, ведь вы можете одновременно читать эту книгу и слушать музыку, или слушать эту книгу во время бега или вязания. Человек же делает два дела одновременно, верно?

Вспомните три этапа сохранения информации: когнитивный этап, ассоциативный этап и автономный этап. Или, говоря другими словами, вы не можете делать два или более дела одновременно, если вы не достигли в них автономного этапа. Например, чтение на родном языке — это не то, чему вам все еще необходимо учиться, так что вы можете что-то читать или слушать на нем, параллельно с этим занимаясь другими делами, которые также автоматизированы, например вязанием. Но когда вы читаете особенно сложный отрывок книги, наполненный множеством новых идей, то вы скорее всего выключите музыку и сосредоточитесь на тексте. Это

[10] «Reducing Interruptions at Work: A Large-Scale Field Study of FlowLight», Мануэла Цюгер, **https://www.zora.uzh.ch/id/eprint/136997/1/FlowLight.pdf**.

ваш мозг говорит вам о том, что он не может выполнять несколько дел одновременно. Например, именно поэтому при парковке водителям приходится выключать радио.

Если вы не верите своему мозгу, то есть научные данные, показывающие, что в реальности многозадачность не так хороша, как может казаться.

Исследования многозадачности

В 2009 году Энни Бет Фокс (Annie Beth Fox), сейчас профессор количественных методов исследований Института медицинских профессий Массачусетской больницы общего профиля[11], сравнила студентов, которые читали текст и одновременно обменивались сообщениями, и студентов, которые работали только с текстом. Обе группы понимали текст одинакового хорошо, но группе, которая отвлекалась на переписку, потребовалось на 50% больше времени как на чтение текста, так и на ответы на вопросы по теме.

В 2010 году нидерландский психолог Пол Киршнер (Paul Kirschner) провел исследование, в котором приняли участие около 200 студентов. В исследовании Киршнер спросил их об их пристрастии к Facebook. Активные пользователи учились так же долго, как и те участники, которые не были пользователями Facebook, однако средний балл активных пользователей соцсети был значительно ниже. Данный вывод особенно верен для тех студентов, которые отвечают на сообщение сразу после его получения. Интересен тот факт, что люди, которые выполняют сразу несколько задач, чувствуют себя очень продуктивными[12].

В исследовании, где студенты выполняли упражнение, одновременно переписываясь с партнером, сами студенты оценили свою успеваемость как удовлетворительную, в то время как их партнеры оценили их успеваемость намного ниже. Это может означать то, что не стоит совмещать программирование и переписку в Slack[13].

Выводы

- При программировании вы выполняете комбинацию различных активностей программирования: поиск, осмысление, переписывание, наращивание и исследование. Каждая активность оказывает соответствующее влияние на разные виды памяти. Следовательно, каждая активность может поддерживаться с помощью разных средств.

---

[11] «Distractions, Distractions: Does Instant Messaging Affect College Students' Performance on a Concurrent Reading Comprehension Task?» *Cyberpsychology and Behavior*, ч. 12, стр. 51–53, 2009, Энни Бет Фокс и др., **https://doi.org/10.1089/cpb.2008.0107**.

[12] «Facebook® and Academic Performance», *Computers in Human Behavior*, ч. 26,№ 6, стр. 1237–1245, 2010, Пол Киршнер и Эрин Карпински, **http://mng.bz/jBze**.

[13] «Impact of Simultaneous Collaborative Multitasking on Communication Performance and Experience», Линбэй Сюй (2008), **http://mng.bz/EVNR**.

- Отвлечения от работы не только раздражают, но и негативно влияют на производительность программиста, т. к. для возвращения к прежнему темпу работы нужно время на изменение воображаемой модели.
- Для того чтобы снизить негативный эффект отвлечения, делайте заметки или комментарии.
- Сознательно поддерживайте проспективную память: если вы не можете что-то сделать, создайте план действий.
- В периоды низкой когнитивной нагрузки постарайтесь ограничить количество отвлечений: например, с помощью автоматизации и использования FlowLight или вручную, поменяв свой статус в Slack.

12 Проектирование и усовершенствование больших систем

В этой главе:

- мы рассмотрим влияние различных проектных решений на понятность баз кода;
- рассмотрим компромиссы между разными проектными решениями;
- изменим проекты существующих баз кода для улучшения работы мозга.

До этого момента мы обсуждали то, как лучше всего читать и писать код. Мы изучили, какие когнитивные процессы участвуют при чтении и написании кода. Однако при работе с большими базами кода на понимание влияют не только мелкие фрагменты кода. Способ организации кода также играет роль в том, как легко людям будет взаимодействовать с кодом. Это особенно верно для кода библиотек, фреймворков и модулей, которые другие люди не изменяют, а просто используют.

Очень часто, когда мы говорим о библиотеках, фреймворках или модулях, мы говорим об их технических аспектах, к примеру языке, на котором они написаны. Но базы кода можно рассматривать и с когнитивной точки зрения. В этой главе мы рассмотрим концепцию когнитивных измерений — методику изучения баз кода с когнитивной точки зрения. Концепция когнитивных измерений может помочь вам ответить на вопросы о больших базах кода, например: «Этот код потом будет просто изменить?» или «Смогут ли потом другие люди найти информацию в этой базе кода?». Изучение баз кода с когнитивной точки зрения поможет вам лучше понять, как другие люди взаимодействуют с вашим кодом.

После того как мы рассмотрим концепцию когнитивных измерений и то, как она может поддержать наше понимание баз кода, мы подробнее изучим то, как эту концепцию можно использовать для улучшения структуры уже существующих баз кода. Данное улучшение проводится с помощью *когнитивного измерения базы кода*.

В предыдущей главе мы рассмотрели пять типов активностей. В этой главе мы также рассмотрим то, как свойства базы кода влияют на различные активности программирования.

12.1. Проверка свойств базы кода

Очень часто, когда мы говорим о библиотеках, фреймворках или модулях, мы говорим об их технических аспектах. Обычно мы говорим: «Эта библиотека написана на Python», «В этом фреймворке используется node.js» или «Этот модуль был заранее скомпилирован».

Когда мы говорим о языках программирования, мы также рассматриваем технические аспекты, например парадигму (объектно-ориентированная, функциональная или обе вместе), систему типов и то, есть ли компиляция языка в байт-код. Вы также можете думать о том, где именно работает язык программирования, фреймворк или библиотека: например, в браузере или в виртуальной машине? Но все эти аспекты касаются только технической части (того, что может делать язык программирования).

Но когда мы думаем о библиотеке, модуле, фреймворке или языке программирования, мы также можем задуматься о том, что они делают с нашим мозгом, а не компьютером.

УПРАЖНЕНИЕ 12.1. Подумайте о кодовой базе, на которой вы недавно работали, но которую не составляли сами. Это может быть библиотека, код которой вам нужно было прочитать для того, чтобы понять алгоритм вызова функции, или фреймворк, в котором вы исправляли ошибки.

Теперь ответьте на следующие вопросы:

- Что именно облегчило выполнение задачи (например, документация, правильные имена переменных, комментарии)?
- Что именно затруднило выполнение задачи (код был сложным, отсутствие документации)?

12.1.1. Когнитивные измерения

Концепцию когнитивных измерений (CDN, cognitive dimensions of notations) можно использовать для оценивания удобства использования баз кода. Изначально данная концепция была создана британскими учеными Томасом Грином, Аланом Блэквеллом и Мариан Петре. Концепция включает в себя несколько измерений, каждое из которых представляет собой отличный способ исследования баз кода. Изначально измерения были созданы для изучения визуализаций, например схем, однако затем их применили к языкам программирования. Языки программирования, как и схемы, можно рассматривать как нотации, т. е. способ выражения мыслей.

Измерения, описанные Грином, Блэквелом и Петре, используются только по отношению к нотациям, но в этой книге мы используем их при работе с базами кода,

а не языками программирования. Мы называем эти измерения *когнитивными измерениями баз кода* (CDCB, cognitive dimensions of codebases) и используем их для изучения баз кода. Так мы понимаем, что нужно изменить, дополнить или улучшить. Когнитивные измерения баз кода особенно полезны для таких баз кода, которые очень часто используются, но не изменяются другими пользователями.

Сначала мы подробно рассмотрим каждое из измерений по отдельности, а потом изучим то, как измерения взаимодействуют друг с другом и как их можно использовать для улучшения баз кода.

Подверженность ошибкам

Первое измерение, которое мы с вами рассмотрим, — это *подверженность ошибкам*. При работе с одними языками программирования шанс сделать ошибку выше, чем с другими. Например, JavaScript является одним из самых популярных языков программирования, но, как нам известно, при его использовании нужно обращать внимание на множество особенностей.

На JavaScript и некоторых других языках с динамической типизацией при создании переменные не инициализируются сразу с типом. Так как во время выполнения кода непонятно, какой это тип объекта, программисты могут запутаться в типах переменных и совершить ошибку. К тому же неожиданное изменение типа переменной тоже может привести к ошибкам. Считается, что языки со строгой системой типов, например Haskell, менее подвержены ошибкам, т. к. при кодировании программист понимает тип каждого элемента.

В базах кода также могут совершаться ошибки, например из-за отсутствия документации или непонятных имен переменных.

Иногда базы кода «наследуют» измерения от языка программирования, на котором они были написаны. Например, написанный на Python модуль сильнее подвержен ошибкам, чем библиотека, написанная на C, т. к. у Python нет строгой системы типов.

Системы типов, предотвращающие ошибки

Вы можете заинтересоваться, а правда ли то, что системы типов предотвращают ошибки. На основе множества экспериментов по сравнению Java и Groovy немецкий исследователь Стефан Ханенберг (Stefan Hanenberg) показал, что системы типов действительно могут помогать в нахождении и исправлении ошибок. Во многих случаях в исследовании Ханенберга место, на которое компилятор указывал как на ошибку, совпадало с местом, на котором при выполнении вылетал код. Выполнение кода занимает много времени, поэтому лучше не стоит проверять ошибки с помощью выполнения кода.

Ханенберг испробовал разные способы по улучшению кода, написанного на языке с динамической типизацией. Он пытался уменьшить его подверженность ошибкам с помощью интерактивных средств разработки и документации, но даже в таком случае в кодах, написанных на языке с динамической типизацией, наблюдалось множество ошибок.

Согласованность

Другой способ изучения того, как люди будут взаимодействовать с языком программирования или базой кода, — это *согласованность*. Насколько похожи элементы? Всегда ли имена структурированы одинаково? В них используется та же форма, которую мы обсуждали в главе? У разных классов одинаковое расположение файлов?

Пример, когда многие языки программирования демонстрируют согласованность — это определение функций. Может быть, вы никогда не задумывались об этом, но у встроенных функций тот же интерфейс, что и у пользовательских функций. Когда вы смотрите на вызов функции, например `print()` или `print_customer()`, вы не видите создателя функции, создателя языка программирования или создателя кода.

Фреймворк или язык, которые не согласованны в использовании имен или соглашений, могут привести к увеличению когнитивной нагрузки, т. к. вашему мозгу потребуется больше энергии для того, чтобы понять, что к чему относится. Вам также может потребоваться больше времени для того, чтобы найти нужную информацию.

Как мы видели в *главе 9*, согласованность связана с подверженностью ошибкам. Код, в котором встречаются лингвистические антипаттерны проектирования, подвержен ошибкам и вызывает дополнительную когнитивную нагрузку.

Размытость

Ранее мы рассмотрели запахи кода, которые могут затруднить чтение кода. Например, запах кода «Длинный метод», который заключается в том, что метод или функция состоят из множества строк. Длинный метод всегда затрудняет понимание информации.

Длинный метод может быть ошибкой программиста: например, программисту захотелось поместить слишком много функций в один метод. Но некоторым языкам программирования просто нужно больше места, чем другим языкам, потому что от этого зависит их функциональность. Это называется *размытостью*. Размытость обозначает то, сколько места занимает элемент кода.

Например, цикл `for` на Python выглядит так:

```
for i in range(10):
    print(i)
```

На C++ этот же код будет выглядеть так:

```
for (i=0; i<10; i++){
    cout << i;
}
```

Если подсчитать строки, то код на C++ состоит из трех строк, а код на Python — из двух. Однако размытость затрагивает не только количество строк кода; может иметь значение также и то, из скольких чанков состоит код. Если вы новичок, то вы

можете посчитать отдельные элементы, а затем разбить их на чанки. Код на Python состоит из семи элементов, а код на C++ состоит из девяти элементов (рис. 12.1).

Разница в количестве чанков вызвана тем, что в коде C++ есть элементы, которых нет на Python (например, `i++`).

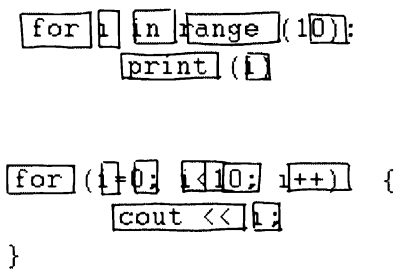


Рис. 12.1. Различные чанки в простом цикле `for` на Python (сверху) и C++ (снизу)

Но, программируя на одном языке, мы также можем встретить более размытые и менее размытые версии одного и того же кода. Мы уже рассматривали пример составления списков на Python в предыдущих главах. Вот два фрагмента кода на Python, которые делают одно и то же:

```
california_branches = []
for branch in branches:
  if branch.zipcode[0] == '9'
    california_branches.append(branch)

california_branches = [b for b in branches if b.zipcode[0] == '9']
```

Второй вариант кода менее размыт, что может улучшить читабельность и понятность кода.

Скрытые зависимости

Измерение *скрытых зависимостей* показывает на то, как зависимости видны пользователю. Примером с высоким уровнем скрытых зависимостей является HTML-страница с кнопкой, управляемой кодом на JavaScript, который находится в отдельном файле. В этом случае из файла с кодом на JavaScript может быть трудно понять, какие именно HTML-страницы вызывают функцию. Другой пример — это требования к файлам, хранящимся отдельно от файлов кода. В кодовой базе не всегда можно увидеть все библиотеки и фреймворки, которые необходимо инсталлировать для правильной работы кода.

Короче говоря, функции, которые вызываются из другой функции или класса, более заметны, чем функции или классы, вызывающие данную функцию. В первом случае мы всегда можем прочитать код функции и понять, какие функции вызываются из тела этой функции.

И хотя современные интегрированные среды разработки могут обнаруживать скрытые зависимости (рис. 12.2), для поиска зависимостей все равно нужно сделать несколько щелчков мышью или несколько нажатий клавиш.

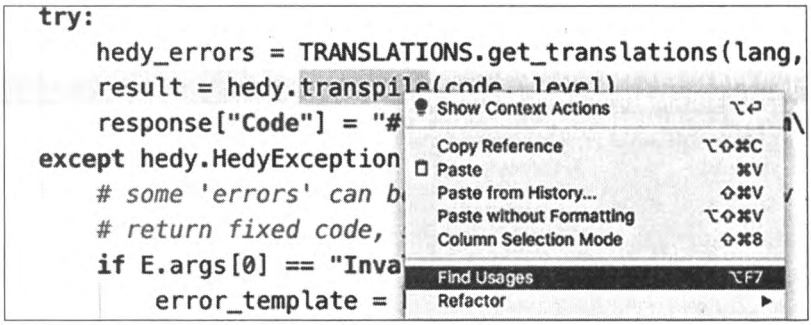


Рис. 12.2. Опция в PyCharm для поиска всех мест вызовов заданной функции

Авторы кода могут компенсировать скрытые зависимости хорошей документацией. У компаний должна быть политика по обсуждению и принятию новых зависимостей, а также внесению их в документацию.

Преждевременная фиксация решения

Измерение *преждевременная фиксация решения* описывает то, насколько легко думать во время использования инструмента. Как уже говорилось в *главе 11*, иногда вы программируете на этапе исследования, когда вы еще не совсем уверены, какой код хотите написать. При исследовании вы можете использовать ручку с бумагой или доску. Эти инструменты имеют максимальную преждевременную фиксацию решения, т. к. вы можете свободно делать наброски, записывать пояснения любого вида и писать неполный или неправильный код без всяких последствий.

Однако как только мы начинаем кодировать в кодовой базе, мы теряем часть нашей свободы. Если мы напишем код с синтаксическими ошибками, то мы не сможем проверить тип. Если мы не сможем проверить тип, то мы не сможем выполнить код. И хотя проверка кода — это дело полезное, она может помешать нам пробовать что-то новое и использовать код как средство мышления, а не модель выполнения.

Если база кода или язык программирования строгие (например, правила использования типов или условий публикации), то использовать код как модель мышления будет затруднительно. О таком коде мы говорим, что у него низкая преждевременная фиксация решения.

Преждевременная фиксация решения является важным фактором в обучении, т. к. если вы новичок в определенной системе, то вам нужно будет как-то выражать свои идеи и писать неверный код. Размышление о плане кода одновременно с размышлением о типах и синтаксисе кода может вызвать слишком сильную когнитивную нагрузку у начинающих программистов.

Вязкость

С преждевременной фиксацией решения связана *вязкость* — то, насколько сложно внести изменения в уже существующую кодовую базу. Как правило, коды, напи-

санные на языках с динамической типизацией, изменить чуть легче. Вы можете просто внести свои изменения и не менять все соответствующие типы. Код, который не является модульным и содержит большие блоки, тоже изменить легче, т. к. вы можете изменить его в одном месте, а не изменять во множестве мест в нескольких функциях или классах.

Легкость изменения кода зависит не только от языка программирования и базы кода; на вязкость также влияют и независимые от базы кода факторы. Например, если на выполнение компиляции или теста нужно слишком много времени, то вязкость каждого из изменений сильно увеличивается.

Поэтапное оценивание

Еще одно измерение, связанное с преждевременной фиксацией решения, — *поэтапное сравнение*. Это измерение характеризует то, как легко проверить или выполнить незаконченный код. Как мы уже знаем, код с высокой преждевременной фиксацией решения позволяет пользователю записывать неполный код. Код с поэтапным оцениванием тоже позволяет пользователю выполнять неполный или неправильный код.

Некоторые системы программирования позволяют программистам работать в реальном времени: программист может изменить код и снова запустить его, не прерывая уже запущенное выполнение кода. Примером такой системы программирования является язык Smalltalk.

Язык Smalltalk стал первым инструментом, поддерживающим *живое программирование* и позволяющим изменять и проверять код во время его выполнения. Scratch, язык программирования для детей, в какой-то степени был вдохновлен Smalltalk и тоже позволяет детям изменять код без прерывания выполнения.

При разработке базы кода или библиотеки вы можете разрешить пользователям выполнять неполный код. Примером проекта с поэтапным оцениванием может послужить использование дополнительных параметров. Когда у функции есть необязательные параметры, то пользователь может сначала скомпилировать и запустить код со значениями по умолчанию, а затем обновлять параметры один за другим на каждом шаге работы системы. Другой пример — система «дыр» (holes) языка Idris, позволяющая запускать неполный код. Затем компилятор предлагает, чем можно заполнить «дыру». Вы можете выполнять итерации и детализировать типы, что приводит к уменьшению «дыр». При этом компилятор становится средством исследования, а не ограничением, которое не дает проводить исследование.

Системы, где поэтапное оценивание выражено не так явно, не позволяют пользователю выполнять неполный код или код без ошибок, что противоречит преждевременной фиксации решения.

Выразительность ролей

Выразительность ролей показывает, насколько легко понять роль разных частей кода. Простым примером является тот факт, что почти во всех языках программирования вызовы функций без параметров записываются с двумя круглыми скобка-

ми в конце, например `file.open()`. И хотя разработчики языка могли решить, что пользователи могут опускать скобки, теперь скобки указывают на то, что `open()` — это функция. Скобки в конце функции представляют собой выразительность ролей.

Еще один известный пример выразительности ролей — это выделение синтаксических конструкций. Во многих интегрированных средах разработки переменные окрашиваются в один цвет, а ключевые слова — в другой. Так вы видите все роли, которые играют элементы кода в программе.

Выразительность ролей может быть достигнута и с помощью синтаксиса. Например, вызов функции, возвращающей булево значение, в виде `is_set` (а не `set`), помогает пользователю понять роль переменной.

Мы видели похожую концепцию в *главе 9*, где рассказывалось о лингвистических антипаттернах проектирования. Если в кодовой базе есть лингвистические антипаттерны, то такие конструкции, как функции и методы, будут вводить читателей в заблуждение относительно своей роли. Это означает, что у базы кода низкая выразительность ролей.

•

Близость соответствия

Измерение *близость соответствия* означает, насколько близок язык программирования или код к предметной области задачи. Некоторые языки программирования имеют хорошую близость соответствия. Мы сталкивались с языком APL с *главы 1*. В следующем листинге снова представлен код на APL (листинг 12.1). И хотя вам могло показаться, что APL — это очень запутанный язык, на самом деле он очень близок к такой предметной области, как векторное исчисление.

Листинг 12.1. Двоичное представление на APL

```
2 2 2 2 2 T n
```

Программа на APL, преобразующая число n в двоичное представление. Замешательство здесь вызвано тем, что вы можете не знать, что означает T.

Например, все переменные по умолчанию являются векторами — мы можем понять это в листинге 12.1 из того, что `T` работает со списком двоек. Если вы привыкли к векторам, а задачи, которые вы решаете, можно решить с помощью векторного исчисления, то данный код не вызовет у вас когнитивной нагрузки. COBOL часто называют языком с хорошей близостью соответствия к сфере бизнеса и финансов. Excel — это еще один пример языка программирования с хорошей близостью соответствия. Расположение данных в строках и столбцах очень точно отображает то, как проводились финансовые расчеты до появления компьютеров.

Большинство современных языков программирования, включая Java, Python и JavaScript, не имеют хорошей близости соответствия. Однако не существует таких задач, которые мы не могли бы решить с помощью этих языков. Конечно, это не всегда плохо. Хорошо всегда иметь возможность решить любую задачу с помощью Python или Java, а не учить новый язык программирования для каждого нового проекта или заказчика.

Базы кода тоже могут иметь хорошую близость соответствия своей предметной области. Базы кода, в которых используются концепции и слова из предметной области, легче для понимания заказчиком, чем базы кода, в которых используются общие термины. Например, метод с именем `executeQuery()` имеет более низкую степень близости соответствия, чем функция `findCustomers()`.

В последнее время в программировании наблюдается растущая заинтересованность в том, чтобы как можно лучше отобразить предметную область в коде. Например, философия предметно-ориентированного проектирования устанавливает, что структура и идентификаторы кода должны представлять предметную область. Это прогресс на пути к хорошей близости соответствия кодовых баз.

УПРАЖНЕНИЕ 12.2. Составьте список всех имен переменных, функций и классов вашей базы кода. Для каждого из имен исследуйте близость соответствия. Задайте себе вопрос для каждого из имен:

- ❒ Имя переменной выражено на языке предметной области?
- ❒ Вне контекста кода понятно ли, к какому процессу относится это имя?
- ❒ Вне контекста кода понятно ли, к какому объекту относится это имя?

Трудность мыслительных операций

Некоторые системы программирования требуют от пользователя выполнения *трудных мыслительных операций* вне самой системы. Например, язык Haskell требует от пользователя думать обо всех типах функций и параметров. У вас не получится игнорировать сигнатуры типа функций, т. к. иначе вы не сможете написать рабочий код. Такая же ситуация и в языке C++, где необходимо использовать указатели во множестве ситуаций, а также рассуждать о них, а не объектах.

Конечно, трудные мыслительные операции — это не всегда плохо. Умственные затраты, которых вы требуете от программистов, потом могут с лихвой окупиться, например, небольшим количеством ошибок в коде со строгой системой типов, высокой производительностью или эффективным использованием памяти.

Однако, когда вы требуете от пользователей проведения трудных мыслительных операций, вы должны это понимать и как следует обдумать эти операции.

Примерами трудных мыслительных операций в кодовых базах являются ситуации, в которых нужно задействовать большую часть памяти. Например, вы просите пользователей запомнить большое количество параметров в нужном порядке для вызова функции — это очень трудная мыслительная операция, потому что кратковременная память испытывает сильную нагрузку.

Мы знаем, что непонятные имена функций имеют низкую близость соответствия. Такие имена тоже являются причиной трудной мыслительной работы. Пользователю приходится запоминать неинформативные имена функций, например `execute()` или `control()`, а также сохранять их в долговременной памяти, что тоже приводит к трудным мыслительным операциям.

И наконец, некоторые операции являются трудными только из-за того, что они негативно влияют на рабочую память. Например, нужные вам данные загружены

с разных источников в разных форматах и были преобразованы в совсем другой формат! Пользователю придется отслеживать различные потоки данных и соответствующие им типы.

Вторичные обозначения

Измерение *вторичные обозначения* указывает на возможность добавить в код дополнительный смысл, которого нет в формальном описании. Наиболее частым примером данного измерения является возможность делать комментарии в исходном коде. Формально комментарии не являются частью языка, во всяком случае в том смысле, что они не меняют поведение кода. Однако комментарии могут помочь читателям лучше понять код. Другой пример вторичных обозначений — это именованные аргументы языка Python. Как показано в листинге 12.2, аргументы могут передаваться вместе с именем, и в таком случае порядок параметров на месте вызова может отличаться от порядка параметров в функции.

Листинг 12.2. Ключевые (именованные) параметры на Python

```
def move_arm(angle, power):
    robotapi.move(angle,power)
# три способа вызова move_arm
move(90, 100)
move(angle = 90, power = 100)
move(power = 100, angle = 90)
```

Программа на Python, демонстрирующая три способа вызова функции с аргументами по порядку, с именами по порядку и с именами в произвольном порядке

Добавление ключевого параметра к вызову функции на Python не меняет поведение кода, но позволяет интегрированной среде разработки отобразить роль каждого из параметров при вызове функции.

Градиент абстракции

Измерение *градиент абстракции* обозначает, может ли пользователь вашей системы создавать собственные абстракции, по мощности не уступающие встроенным абстракциям. Примером абстракции, которая допускается большинством языков программирования, является создание функций, объектов и классов. Программисты могут создавать собственные функции, которые во многом могут быть похожи на встроенные функции. Пользовательские функции могут иметь входные и выходные параметры и работать точно так же, как и обычные функции. Сам факт того, что пользователи могут создавать функции, означает то, что пользователи могут подстраивать язык программирования под себя и добавлять собственные абстракции. И хотя возможность создавать свои собственные абстракции доступна теперь почти во всех языках программирования, многие программисты никогда не работали в предструктурированных системах программирования, таких как ассемблер или некоторые диалекты BASIC, где таких возможностей не было.

Библиотеки и фреймворки могут также предлагать пользователям возможность создавать собственные абстракции. Например, разрешение в библиотеке создавать

подкласс, в который могут быть добавлены дополнительные функции, даст больше свободы для создания абстракции, чем библиотека, в которой только разрешены вызовы API.

Наглядность

Наглядность указывает на то, насколько легко увидеть разные части системы. В кодовой базе иногда сложно увидеть, из каких классов она состоит, особенно если код располагается в нескольких файлах.

Библиотеки и фреймворки могут предлагать пользователям разные уровни наглядности. К примеру, API, извлекающий данные, может вернуть строку, файл JSON или объект. Каждый из них имеет разную наглядность. Если возвращается строка, то форму данных увидеть сложнее, т. е. фреймворк предлагает пользователю низкий уровень наглядности.

12.1.2. Использование когнитивных измерений базы кода для улучшения базы кода

Мы рассмотрели разные измерения, которые могут иметь программы. Эти различия сильно влияют на то, как люди взаимодействуют с базой кода. Например, если у базы кода высокая вязкость, то другие разработчики, которые будут работать с вашей базой кода, не захотят вносить в код изменения. Это может привести к трудным и долгим исправлениям, а не изменениям в структуре базы кода. Если для работы с базой кода с открытым исходным кодом нужно выполнять трудные мыслительные операции, то у людей будет меньше шансов стать сопроводителем. Следовательно, важно понимать то, как ваша база кода работает с различными измерениями.

Список когнитивных измерений можно использовать в качестве списка для проверки кодовой базы. Не все измерения одинаково влияют на все базы кода, но ознакомление с каждым из измерений позволит вам создать понятную и удобную кодовую базу. В идеале нужно регулярно анализировать измерения своей базы кода (к примеру, раз в год).

УПРАЖНЕНИЕ 12.3. Заполните следующую таблицу, чтобы закрепить понимание измерений. Какие измерения важны для вашей базы кода? Какие измерения можно улучшить?

Измерение	Релевантное?	Можно улучшить?
Подверженность ошибкам		
Согласованность		
Размытость		
Скрытые зависимости		
Преждевременная фиксация решения		

(окончание)

Измерение	Релевантное?	Можно улучшить?
Вязкость		
Поэтапное оценивание		
Выразительность ролей		
Близость соответствия		
Трудность мыслительных операций		
Вторичные обозначения		
Градиент абстракции		
Наглядность		

12.1.3. Проектные маневры и их плюсы и минусы

Внесение изменений в кодовую базу для улучшения одного из измерений называется *проектным маневром*. Например, добавление типов в кодовую базу — это проектный маневр, повышающий вероятность совершения ошибок, а изменение имен функций, чтобы так они больше соответствовали предметной области кода, — это проектный маневр, увеличивающий близость соответствия.

УПРАЖНЕНИЕ 12.4. Изучите список, созданный в упражнении 12.3. Видите ли вы, какие проектные маневры можно применить? А как эти маневры повлияют на остальные измерения?

Измерение	Проектный маневр	Влияние измерений позитивное?	Влияние измерений негативное?

Очень часто проектный маневр (т. е. изменение одного из измерений) вызывает изменения в других измерениях. То, насколько сильно связаны измерения, зависит от вашей базы кода. Но есть несколько противоречий, которые конфликтуют друг с другом.

Подверженность ошибкам и вязкость

Если вы хотите предотвратить ошибки пользователя вашей библиотеки или фреймфорка, то пользователю очень часто придется писать дополнительные фрагменты кода. Самый известный пример снижения подверженности ошибкам — это разрешение пользователям добавлять типы к классам-сущностям. Если компилятору известен тип элемента, то эту информацию можно будет использовать для предотвращения ошибок, например случайное добавление списка к строке.

Но иногда, когда в системе уже все есть, пользователю приходится делать лишнюю работу. Например, вам нужно преобразовать переменные в другой тип, чтобы использовать переменные в своих целях. Людям могут не нравиться системы типов из-за того, что они добавляют в код дополнительную вязкость.

Преждевременная фиксация решения и поэтапное оценивание против подверженности ошибкам

Система с поэтапным оцениванием и высокой преждевременной фиксацией решения позволяет пользователю выполнять неполный и неполноценный код. И хотя эти измерения в какой-то степени могут помочь обдумать задачу, может так случиться, что неполные программы не будут удалены, а неполноценный код никогда не будет улучшен. Это приведет к тому, что код будет трудно понять и отладить, что повлияет на подверженность ошибкам.

Выразительность ролей и размытость

Мы уже видели, что выразительность ролей может быть достигнута с помощью добавления синтаксических элементов, например именованных параметров. Однако эти дополнительные идентификаторы удлиняют код. Это верно и для сигнатур типов, которые также выражают роли переменных, но при этом увеличивают размер базы кода.

12.2. Измерения и активности

В предыдущей главе мы рассмотрели пять активностей: поиск, осмысление, переписывание, наращивание и исследование. Каждая из этих активностей накладывает собственные ограничения на когнитивные измерения, для которых нужно изменить кодовую базу. Связь между измерениями и активностями показана в табл. 12.1.

12.2.1. Влияние измерений на разные активности

В *главе 11* мы рассмотрели пять типов активностей, которые люди выполняют во время программирования. По правде говоря, эти активности происходят из оригинальной концепции когнитивных измерений. Блэквелл, Петре и Грин описали эти активности, т. к. эти активности взаимодействуют с измерениями. Некоторые типы активности требуют того, чтобы измерение было высоким, в то время как для работы других нужно низкое измерение. Это показано в табл. 12.1.

Поиск

При поиске некоторые измерения играют большую роль. Например, скрытые зависимости нанесут вред поиску, т. к. в том случае, если вы не знаете код и что откуда вызывается, то вы можете не знать, что нужно читать дальше. Следовательно, вы потратите много времени на поиск. Размытость приводит к тому, что код становится очень длинным, что тоже негативно сказывается на поиске — вам нужно искать в больших фрагментах кода.

Таблица 12.1. Измерения и активности, которые они поддерживают или которым вредят

Измерение	Помощь	Вред
Подверженность ошибкам		Наращивание
Согласованность	Поиск, осмысление	Переписывание
Размытость	Поиск	
Скрытые зависимости		Поиск
Преждевременная фиксация решения	Исследование	
Вязкость		Переписывание, наращивание
Поэтапное оценивание	Исследование	
Выразительность ролей	Осмысление	
Близость соответствия	Наращивание	
Трудные мыслительные операции		Переписывание, наращивание, исследование
Вторичные обозначения	Поиск	
Градиент абстракции	Осмысление	Исследование
Наглядность		Осмысление

С другой стороны, вторичные обозначения могут помочь при поиске, т. к. комментарии и имена переменных могут указывать на то, где искать нужную информацию.

Осмысление

Некоторые характеристики особенно важны при осмыслении кода. Например, низкая наглядность может повредить осмыслению, т. к. из-за этого очень сложно увидеть и, следовательно, понять, как различные классы и функции связаны между собой.

С другой стороны, выразительность ролей особенно полезна для осмысления. Если тип и роль переменных и остальных элементов ясны, то осмысление кода будет проходить легче.

Переписывание

При переписывании (т. е. реализации функции на основе плана) некоторые измерения могут вредить, например согласованность. И хотя согласованная база кода очень подходит для осмысления, при реализации новой функции вам придется изменять новый код под существующую кодовую базу. Как вы понимаете, это приведет к трудным мыслительным операциям. Конечно, в итоге все ваши усилия могут окупиться, однако на это нужно тратить время и силы.

Наращивание

Добавление новых функций в кодовую базу поддерживается близостью соответствия к предметной области кода. Если при работе с базой кода вы думаете о смысле

и цели кода, а не о концепциях программирования, то вам будет легче наращивать код. С другой стороны, базы кода с высокой вязкостью затрудняют добавление нового кода.

Исследование

Изучение новых идей проектирования в кодовой базе (т. е. исследование) поддерживается системами с поэтапным оцениванием и высокой преждевременной фиксацией решения. Трудные мыслительные операции и абстракции могут нанести вред данному типу активности, т. к. они создают высокую когнитивную нагрузку, ограничивая нагрузку, которую программист может использовать на исследование проблемы и поиск решения.

12.2.2. Изменение базы кода под ожидаемые активности

Мы уже видели, что разные активности накладывают различные ограничения на систему. Соответственно, вы должны понимать, какие действия будут выполнять люди при работе с вашей базой кода. В относительно старых и стабильных библиотеках поиск будет проводиться чаще, чем в наращиваемых библиотеках; при это более новые приложения чаще будут наращиваться или переписываться. Это означает, что со временем кодовой базе могут потребоваться проектные маневры, которые сделают базу соответствующей действиям, которые выполняют пользователи.

УПРАЖНЕНИЕ 12.5. Подумайте о своей базе кода. Какие активности вероятны больше всего? За последние несколько месяцев эти активности были стабильными? Какие измерения играют роль в этих активностях и как ваша база кода использует эти измерения?

Выводы

- Концепция когнитивных измерений — это модель, помогающая программистам определить когнитивный эффект, который окажут на пользователей языки программирования.
- Когнитивные измерения базы кода — это расширение концепции когнитивных измерений, которая помогает программистам понять, какое влияние их библиотеки, кодовые базы и фреймворки окажут на пользователей.
- Во многих случаях нужно идти на компромисс между измерениями базы кода. Улучшение одного измерения может привести к ухудшению другого.
- Улучшение структуры существующих баз кода с точки зрения концепции когнитивных измерений можно выполнить с помощью проектного маневра.
- Разные активности обладают разными требованиями к измерениям, под которые подстраивается база кода.

13 Как ввести новых программистов в курс дела

В этой главе:

- сравним образы мышления опытных программистов и начинающих;
- узнаем, как упростить адаптацию новых программистов в кодовой базе;
- научимся поддерживать новых программистов при обучении работе с новым языком программирования или фреймворком.

До сих пор мы изучали то, как нужно читать и структурировать код. Однако, будучи старшим и опытным программистом, вы будете часто сталкиваться с собственным замешательством и замешательством людей, с которыми вы будете работать. Во многих случаях вы захотите регулировать когнитивную нагрузку, которую будут испытывать ваши подопечные, чтобы убедиться в том, что они учатся с наибольшей эффективностью.

В этой главе мы рассмотрим, как можно улучшить процесс адаптации как для опытных программистов, например в незнакомой кодовой базе, так и для программистов-новичков.

Сначала мы рассмотрим, как думают и действуют профессиональные и начинающие программисты. Затем мы рассмотрим различные активности, которые может делать команда для адаптации новых коллег. К концу данной главы вы познакомитесь с тремя стратегиями и упражнениями, которые помогут вам поддерживать новичков эффективнее.

13.1. Проблемы процесса адаптации

Будучи опытным программистом, вы, скорее всего, сталкивались с ситуациями, в которых вам нужно было привлечь новичков. Это может быть новичок в команде

или в проекте с открытым исходным кодом. Не все программисты умеют обучать и наставлять других, поэтому процесс адаптации может оказаться разочаровывающим для обеих сторон. В этой главе мы подробно рассмотрим то, что происходит во время адаптации в мозгу новичка, а также то, что вы можете делать для того, чтобы управлять адаптацией.

Процессы адаптации, которые я наблюдала лично сама, были примерно вот такими:

- старший программист заваливает новичка большим количеством информации. Информации настолько много, что новичок испытывает чрезмерно высокую когнитивную нагрузку. Например, наставник знакомит сотрудника с новыми людьми, предметной областью базы кода, трудовым процессом и базой кода;
- затем старший программист задает новому коллеге вопросы или дает ему задание, например исправление маленькой ошибки или добавление небольшой функции. Наставнику данное задание кажется очень простым;
- новый сотрудник не справляется, потому что он испытывает слишком сильную когнитивную нагрузку. Недостаток знания предметной области в сочетании с незнакомым языком программирования и отсутствием автоматизированных навыков негативно сказываются на нагрузке новичка.

В чем же заключается проблема взаимодействия между наставником и новым сотрудником? Самая серьезная проблема состоит в том, что наставник с легкостью перегружает рабочую память новичка, ожидая от него слишком многого. Давайте слегка освежим нашу память и вспомним основные концепции, которые мы рассматривали в предыдущих главах. В *главе 3* мы рассмотрели когнитивную нагрузку, т. е. усилие, которое делает мозг для решения поставленной задачи. Мы видели, что если мозг испытывает слишком сильную когнитивную нагрузку, то продуктивность человека резко снижается. В *главе 10* мы узнали, что когда человек испытывает слишком сильную внешнюю и внутреннюю когнитивную нагрузку, то у него не хватает места для соответствующей нагрузки, а это значит, что вы не сможете *запоминать* новую информацию.

Так как рабочая память новичков перегружена, они не смогут быть продуктивными при программировании в новой базе кода, а также не смогут нормально запоминать новую информацию. Я не один раз видела, как это приводило к разочарованию с обеих сторон. Руководитель может посчитать, что новичок не на многое способен, а у новичка складывается предположение, что проект будет слишком сложным. Не самый хороший старт для дальнейшего сотрудничества.

Одной из причин, почему вышестоящие сотрудники не могут нормально обучать и объяснять, является «проклятие опыта». Как только вы овладеваете каким-либо навыком или знанием, вы забываете, как трудно было этого добиться. Следовательно, вы переоцениваете объем новой информации, которую новичок может одновременно обработать.

Я уверена, что за последние несколько месяцев вы не один раз сказали что-то вроде: «Да это было не так уж и сложно», «Это было легко» или «Пустяк». Думаю, что в большинстве случаев вы так говорили о знаниях, на изучение которых у вас ушла не одна неделя. Ситуации, когда вы говорите: «Да это легко!», могут быть теми

моментами, когда вы находитесь под «проклятием опыта». Первое, что вы можете сделать для того, чтобы упростить процесс адаптации, — это осознать то, что для человека, который только учится, все эти вещи не кажутся легкими.

13.2. Различия между профессионалами и новичками

Очень часто профессионалы считают, что новички рассуждают так же, как и они, только чуть медленнее. Самый важный вывод, который можно сделать из этой главы состоит в том, что профессионалы и новички думают и действуют совсем поразному.

Ранее в этой книге мы уже рассматривали причины, по которым профессионалы могут думать по-другому. Во-первых, в мозге профессионала намного больше сформированных схем, которые рабочая память может взять из долговременной памяти. Хранящиеся воспоминания могут быть как стратегиями, например написание теста для решения проблемы, или событийной памятью о том, что они пытались сделать в прошлом, например перезагрузку сервера. Профессионалы не всегда знают ответы на все вопросы. Им тоже нужно время на то, чтобы сравнить несколько решений, однако обычно они уже имеют опыт решения задач.

Во-вторых, профессиональный программист может очень эффективно разбивать код на чанки, а также разбираться в артефактах кода, например сообщениях об ошибках, тестах, задачах и их решениях. Чаще всего профессионал может посмотреть на фрагмент кода и понять, что это, допустим, сбрасывание очереди. Новичок же должен читать код строку за строкой. Простое сообщение об ошибке, например «Индекс массива выходит за пределы допустимого диапазона», для эксперта выглядит как одна концепция. Но новичок видит три отдельных элемента, что сильно увеличивает когнитивную нагрузку. Многие ситуации, в которых более опытные сотрудники считают, что их новый коллега «не такой уж и хороший программист», на самом деле являются моментами «проклятия опыта», в которых новичок всего лишь чрезмерно загружен.

13.2.1. Поведение новичка более подробно

Чтобы лучше понять поведение новичков, давайте рассмотрим понятие неопиажизма, которое объясняет поведение людей при встрече с новой информацией. Неопиажизм основан на работах Жана Пиаже (Jean Piaget), влиятельного психолога, который сосредоточился на четырех стадиях развития маленьких детей. Концепция неопиажизма очень подходит под поведение программистов, когда они только начинают знакомиться с языком программирования, парадигмой или базой кода.

Оригинальная концепция Пиаже

Перед тем как я расскажу вам, как программисты ведут себя при обучении, давайте рассмотрим этапы поведения, которые мы все проходили в детстве. Сначала я

должна объяснить вам оригинальную концепцию Пиаже о развитии детей, показанную в табл. 13.1. На первом этапе, на котором находятся дети в возрасте от 0 до 2 лет, дети не могут строить планы или что-либо контролировать. Они ощущают вещи (сенсорные функции) и трогают (моторные функции) без особого смысла. На втором этапе, когда детям от 2 до 7 лет, они начинают формировать простые представления. Например, четырехлетний ребенок решит, что дождь идет из-за того, что облакам грустно. Это не совсем правда, но вы видите, что они пытаются найти логическое объяснение своим мыслям.

На третьем этапе, когда дети в возрасте от 7 до 11 лет, дети формируют представления, о которых они могут рассуждать, но лишь в определенных ситуациях. Например, они могут сделать хороший ход в настольной игре, но им будет трудно объяснить, всегда ли это будет хороший ход в разных играх. Подобные заключения складываются уже на заключительном этапе — этапе конкретных ситуаций, — когда детям больше 11 лет.

Таблица 13.1. *Краткое описание этапов когнитивного развития Пиаже*

Этап	Описание	Возраст ребенка
Сенсомоторный этап	Дети не имеют понятия о плане или стратегии, они просто чувствуют и трогают вещи	0–2 года
Дооперациональный этап	Дети начинают учиться строить представления и планы, но не полагаются на них при мышлении	2–7 лет
Этап конкретных ситуаций	Дети могут рассуждать о вещах, которые они видят, но им сложно сделать общий вывод	7–11 лет
Этап формальных операций	Дети могут совершать формальные операции	11 лет и старше

Концепция неопиажизма для программирования

Концепция Пиаже встретила некоторую критику из-за того, что при создании концепции Пиаже использовал собственных детей. Однако его работа стала основой концепции неопиажизма, которая имеет большое значение для понимания мышления начинающего программиста. Основная идея состоит в том, что уровни Пиаже являются предметными. Люди могут быть на этапе формальных операций в одной предметной области, например программировать на Java, но при этом при программировании на Python они могут находиться на сенсомоторном этапе. Может быть и такое, что на одной базе кода программист находится на этапе формальных операций, но как только он начинает работу с другой базой кода, он переходит на более низкий этап развития. В табл. 13.2 описана концепция неопиажизма и ее применение для программирования, предложенное австралийским профессором Раймондом Листером (Raymond Lister)[1].

[1] «Toward a Developmental Epistemology of Computer Programming», Раймонд Листер, **https://dl.acm.org/doi/10.1145/2978249.2978251**.

Таблица 13.2. *Обзор этапов концепции неопиажизма и соответствующего поведения при программировании*

Этап	Описание	Поведение
Сенсомоторный этап	Дети не имеют понятия о плане или стратегии, они просто чувствуют и трогают вещи	У программиста нет четкого понимания работы программы На данном этапе программист не может корректно трассировать программу
Дооперациональный этап	Дети начинают учиться строить представления и планы, но не полагаются на них при мышлении	Программист может достоверно рассчитать результат работы нескольких строк кода вручную, например составив таблицу трассировки На этом этапе программист часто пытается догадаться, что делает фрагмент кода
Этап конкретных ситуаций	Дети могут рассуждать о вещах, которые они видят, но им сложно сделать общий вывод	Программист рассуждает о коде дедуктивно, основываясь на чтении самого кода, а не на дооперациональном индуктивном подходе
Этап формальных операций	Дети могут совершать формальные операции	Программист может рассуждать логически, последовательно и систематически. Рассуждения на этом этапе включают в себя размышления над собственными действиями, что особенно полезно при отладке

На первом этапе, который показан в виде программиста слева на рис. 13.1, программисты не могут правильно трассировать программу. Поведение на данном этапе типично для людей, у которых нет или мало опыта программирования. Программисты также могут оказаться на этом этапе в том случае, если они переключаются между двумя совсем разными языками (например, с JavaScript на Haskell). Так как выполнение программ на этих языках сильно отличается, то опытный программист на JavaScript столкнется с трудностями при трассировке программы на Haskell. Так как на этом этапе программисты фокусируют все свое внимание на коде, который для них все еще труден для понимания, объяснение основных концепций в отрыве от кода не пойдет им на пользу. Это пример плохого обучения. Например, вы не поможете программисту на сенсомоторном этапе, который по шагам пробирается через код базы данных, если начнете объяснять, как база данных сконфигурирована по всему коду. Сначала ему нужно понять модель выполнения.

Второй этап — это дооперациональный этап, на котором программист может трассировать небольшие фрагменты кода. Однако это единственный способ, которым программист может рассуждать о коде, — использовать только что приобретенный навык трассировки. Программисты на этом этапе с трудом могут объяснить смысл кода. Такой программист сосредоточен на самом коде и не может переключать внимание на другие объекты, особенно диаграммы. Бесполезно пытаться помочь им читать и писать код, предоставляя схемы и диаграммы. Так как они придерживаются индуктивного умозаключения при работе с кодом, они часто пытаются угадать поведение кода, полагаясь на трассировку.

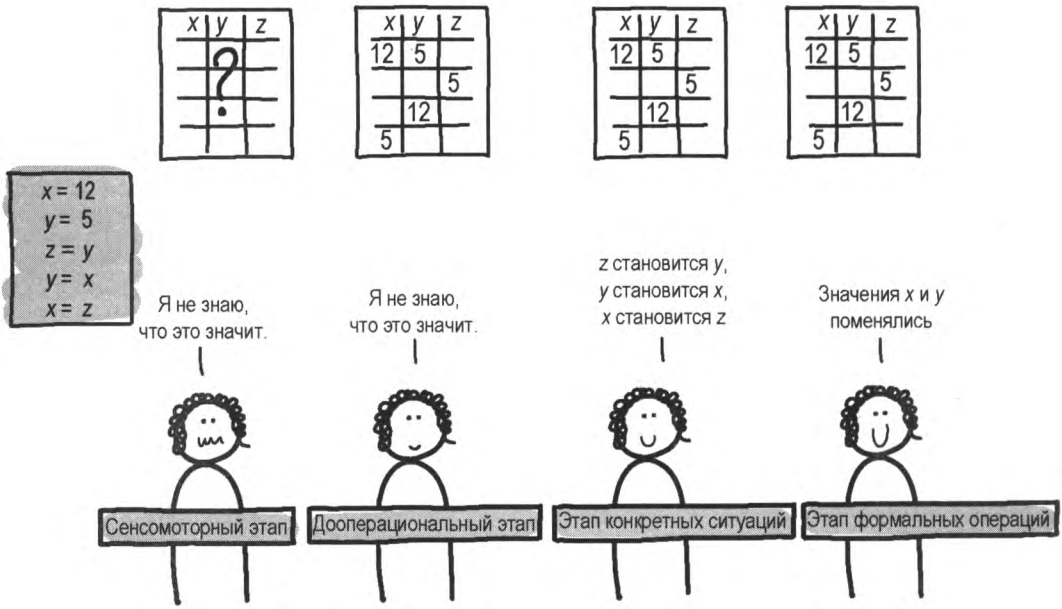


Рис. 13.1. Обзор четырех этапов программирования в концепции неопиажизма

Мне кажется, что именно второй этап больше всего расстраивает не только программистов, но и их наставников. Так как программистам, которые находятся на дооперациональном этапе, сложно понять смысл кода, они часто делают предположения. Это делает их непредсказуемыми. Иногда их догадки, основанные на имеющихся знаниях (или удаче), оказываются верными, однако спустя какое-то время они могут сказать какую-нибудь глупость. Может произойти такая ситуация, когда наставник разочаруется в новичке и будет считать его недостаточно способным или не старающимся как следует. Однако дооперациональный этап — это этап, который необходимо пройти, чтобы перейти на следующий. Обучение новичков путем расширения их словаря элементов кода с помощью дидактических карточек может помочь им продвинуться вперед.

На третьем этапе программисты могу рассуждать о коде, при этом не следя за каждой строкой кода. Они делают это с помощью имеющихся знаний: узнают знакомые чанки кода, вчитываются в комментарии и имена элементов и отслеживают процесс выполнения кода только тогда, когда это необходимо (к примеру, при отладке). В своем исследовании Листер говорит, что только на этой стадии программисты могут использовать в качестве поддержки мышления диаграммы. Программисты приобретают черты «настоящих» программистов: рассуждают о коде, составляют план и придерживаются его по мере написания кода. Однако они еще не полностью понимают кодовую базу, а также не совсем уверены в том, каких стратегий нужно придерживаться в той или иной ситуации. Например, это может проявляться в том, что программист будет придерживаться только одной концепции (например, начинающий программист тратит целый день на исправление одной ошибки, у него ничего не получается, и он пытается снова, вместо того, чтобы отступить и пересмотреть выбранную стратегию).

Последний этап — это этап формальных операций. Это уже опытные программисты, которые свободно рассуждают о коде, их поведение вполне логично, и такие программисты не представляют особого интереса для адаптации. Такие программисты будут свободно изучать детали кодовых баз самостоятельно и могут попросить о помощи, когда это необходимо.

При изучении новой информации вы можете временно забывать некоторые вещи

Четыре этапа представлены как разные, отдельные друг от друга этапы, хотя на самом деле это не так. При изучении новой концепции программирования или базы кода обучающиеся могут временно откатиться на предыдущий этап развития. Если кто-то спокойно читает функции на Python без трассировки, а затем сталкивается с функциями с переменным количеством аргументов, в которых используется `*args`, то этому человеку может потребоваться трассировка некоторых вызовов функций; только после того, как они поймут, что происходит, они снова смогут спокойно читать функции без их трассировки.

УПРАЖНЕНИЕ 13.1. При обучении в компаниях очень часто встречаются эти четыре вида поведения. Изучите этапы неопиажизма, затем приведите пример, который вы видели на практике. Заполните следующую таблицу.

Этап	Поведение	Пример
Сенсомоторный этап	У программиста нет четкого понимания работы программы На данном этапе программист не может корректно трассировать программу	
Дооперациональный этап	Программист может достоверно рассчитать результат работы нескольких строк кода вручную, например составив таблицу трассировки На этом этапе программист часто пытается догадаться, что делает фрагмент кода	
Этап конкретных ситуаций	Программист рассуждает о коде дедуктивно, основываясь на чтении самого кода, а не на дооперациональном индуктивном подходе	
Этап формальных операций	Программист может рассуждать логически, последовательно и систематически Рассуждения на этом этапе включают в себя размышления над собственными действиями, что особенно полезно при отладке	

13.2.2. Разница между вещественным и абстрактным видением концепций

Мы уже знаем, что новички и профессиональные программисты думают и действуют по-разному. Исследования также подтверждают то, что профессиональные программисты говорят о концепциях как о чем-то абстрактном. Например, при

объяснении вариативной функции на Python тому, кто плохо знаком с данной концепцией, опытные программисты могут сказать, что это функция с переменным числом аргументов. Однако они оставят без ответа множество вопросов: например, как получить доступ к этим аргументам, какие имена присваивать аргументам или существует ли ограничение на число аргументов.

Однако новичкам полезны обе формы объяснения. В идеальном случае понимание новичка соответствует *семантической волне* — концепции, описанной австралийским ученым Карлом Мэтоном (Karl Maton). Семантическая волна показана на рис. 13.2[2].

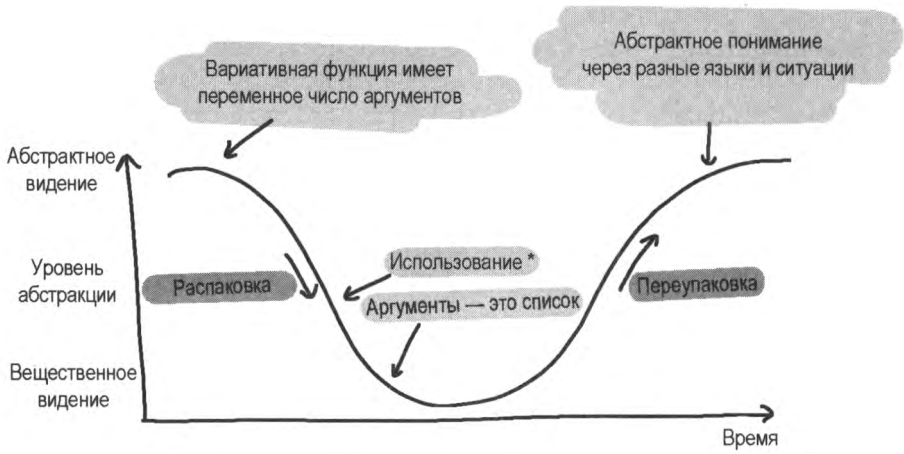


Рис. 13.2. Семантическая волна, являющаяся прекрасным объяснением того, что все начинается с абстрактного видения, потом учащийся распаковывает знания, смотрит на конкретные вещи, а затем учащийся переупаковывает знания в своей долговременной памяти

Согласно семантической волне сначала новичкам нужно понять общую концепцию: для чего она нужна и зачем ее нужно знать. Например, функция с переменным числом аргументов полезна тем, что она позволяет использовать столько аргументов, сколько потребуется.

После того как новички узнали в общем, что делает концепция, кривая идет вниз — это процесс, известный как *распаковка*. Затем новичок готов узнать концепцию в подробностях. Например, он узнает, что символ * используется для обозначения функции с переменным числом аргументов в языке Python, а также то, что Python реализует список аргументов в виде списка. Новичок понимает, что на самом деле нет множества аргументов; есть только один аргумент, в котором могут быть все аргументы функции в качестве элементов кода.

Наконец, новичок возвращается на абстрактный уровень. Он отходит от деталей и чувствует себя намного увереннее, зная общий принцип работы концепции. Этот этап называется *переупаковкой*. Когда концепция правильно переупакована, то

[2] «Making Semantic Waves: A Key to Cumulative Knowledge-Building», *Linguistics and Education*, ч. 26, № 1, стр. 8–22, 2013, Карл Мэтон, **https://www.sciencedirect.com/science/article/pii/S0898589812000678**.

учащийся может думать о ней как о чем-то общем, не заостряя внимания на деталях. Переупаковка также включает в себя внедрение новых знаний в долговременную память, а также отвечает за образование новых схем между новыми и старыми знаниями, например «на C++ поддерживаются функции с переменным числом аргументов, а не Erlang — нет».

У новичков есть три антипаттерна, которые они часто используют, а также различные объяснения, показанные на рис. 13.3. Первый антипаттерн называется высокой линией выхода на плато застоя и означает, что вы пользуетесь только абстрактными терминами. Новичок в Python может узнать, что в данном языке программирования есть функции с переменным числом аргументов и почему они полезны, но если он никогда не видел синтаксис такой функции, то ему придется заняться его изучением позже.

Второй антипаттерн — это низкая линия выхода на плато. Некоторые опытные программисты загружают новых коллег информацией, но при этом они не объясняют, почему та или иная концепция полезна и важна. Предложение типа «Функция с переменным числом аргументов начинается с *, а затем Python видит все аргументы как список» не будет иметь никакой пользы для новичка, который не знает, когда и для чего нужно использовать функцию с переменным числом аргументов.

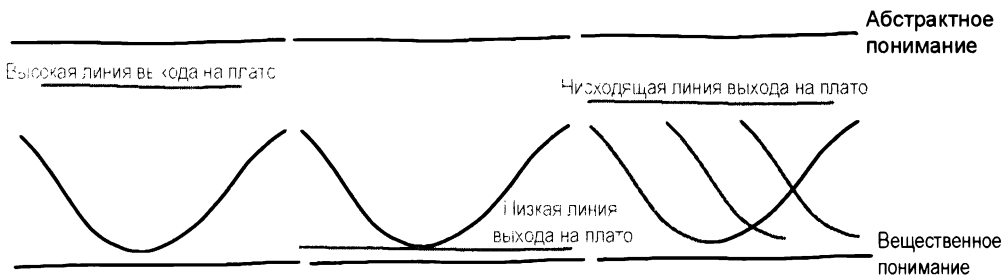


Рис. 13.3. Три антипаттерна новичка высокая линия выхода на плато (только абстрактные объяснения), низкая линия выхода на плато (только вещественные объяснения) и нисходящая линия выхода на плато (начиная от высокой линии к низкой, но при этом места для переупаковки нет)

Последний антипаттерн начинается с абстрактного понимания, а потом, повторяя движение семантической волны, спускается к вещественному пониманию. Однако после объяснения деталей наставник забывает позволить ученикам переупаковать новую информацию. Говоря другими словами, опытный программист показывает «почему» и «как», но не дает времени на создание новых схем для новой информации в долговременной памяти. Вы можете возобновить переупаковку, расспросив наставника о том, что общего между новой информацией и уже имеющимися знаниями.

УПРАЖНЕНИЕ 13.2. Выберите хорошо знакомую вам концепцию, а затем найдите ее объяснения, соответствующие всем трем положениям семантической волны, показанной на рис. 13.1.

13.3. Активности для улучшения процесса адаптации

В оставшейся части главы мы рассмотрим, как можно улучшить процесс адаптации. Первое и одновременно самое важное, что вы можете сделать, — это следить за когнитивной нагрузкой людей, которые начинают у вас работать. Понятно, что будет очень хорошо, если ваш новый коллега сам будет управлять своей когнитивной нагрузкой. Знакомство команды с такими терминами, как типы памяти (к примеру, долговременная, кратковременная и рабочая), когнитивная нагрузка и разделение на чанки поможет вам настроить коммуникацию. Будет намного проще, если новичок скажет: «Я испытываю слишком высокую когнитивную нагрузку при чтении этого кода» или «Думаю, мне не хватает чанков для Python», а не «Я ничего не понимаю». Давайте теперь подробно рассмотрим три способа адаптации.

13.3.1. Ограничение заданий до одной активности

В *главе 11* мы описали пять активностей, которые программисты могут выполнять в кодовой базе: переписывание, исследование, осмысление, поиск и наращивание. Одна из проблем адаптации заключается в том, что от новичков требуют выполнения как минимум четырех активностей: поиск подходящего места для реализации функции или релевантной информации, понимание незнакомого исходного кода, изучение базы кода для улучшенного понимания и расширение базы кода новой функцией.

Как мы видели в *главе 11*, для разных типов активности есть разные когнитивные требования как к программисту, так и к системе. Новичкам трудно переключаться между разными типами активности. Несмотря на то что они знают язык программирования и даже предметные области, выполнение разных задач вызовет высокую когнитивную нагрузку.

В процессе адаптации лучше всего специально подбирать задания для всех пяти активностей и предлагать новичку выполнять их одно за другим. Давайте рассмотрим пять активностей и примеры того, как каждая из них может помочь начинающему программисту (см. табл. 13.3).

Таблица 13.3. *Обзор активностей и того, как их можно использовать для поддержки новых сотрудников*

Активность	Пример использования для поддержки адаптации
Исследование	Просмотр кодовой базы для получения общего представления о кодовой базе
Поиск	Поиск класса, реализующего определенный интерфейс
Переписывание	Предоставление новичку четкого плана реализации метода на естественном языке
Осмысление	Понимание деталей кода, например описание определенного метода на естественном языке
Наращивание	Добавление свойства к существующему классу Включает в себя составление плана

Различные активности могут быть связаны друг с другом и работать со связанными частями кода. Например, первой задачей может быть поиск класса, второй задачей — написание кода метода из этого класса, третьей задачей — расширение класса способом посложнее. Вы можете чередовать задачи, в которых основное внимание уделяется именно изучению концепций программирования, с задачами, в которых основной упор идет на изучение предметной области (зависит от имеющихся у новичка знаний).

УПРАЖНЕНИЕ 13.3. Подумайте о конкретном задании, которое новичок мог бы выполнять во всех пяти категориях. При этом он не знаком с вашей базой кода.

Можно также представить, что для помощи новичкам команда может создавать и сохранять в актуальном состоянии документацию (например, для поддержки исследования делать полезные комментарии и описания архитектуры, объясняющие модули, подсистемы, структуры данных и алгоритмы, которые используются в системе).

13.3.2. Поддержка памяти новичка

Как мы уже говорили, первое, что важно при адаптации, — это понимать то, что для новичка обыденные вещи не кажутся простыми. Как мы говорили в *главе 2*, новички и опытные программисты запоминают и действуют по-разному. Возможно, это даже не надо говорить вслух, но важно проявлять терпение и внимание!

Существуют три этапа, на которых вы можете улучшить процесс адаптации. Все они связаны с тремя формами замешательства, которые мы рассматривали в *главе 1*.

Поддержка долговременной памяти: объяснение релевантной информации

Во-первых, вы можете подготовиться к процессу адаптации новичков, понимая, какая именно информация играет роль при работе с базой кода. Это можно сделать с командой еще до прихода нового сотрудника.

Например, вы можете создать документацию с важными концепциями предметной области, с которыми новичок столкнется в коде. Другая важная релевантная информация, о которой не стоит забывать, — это все библиотеки, фреймворки, базы данных и другие внешние инструменты, использующиеся в коде. Предложение типа «Мы используем Laravel для вот этого веб-приложения, которое мы внедряем на Heroku с Jenkins» окажется совершенно бессмысленным, если вы не знаете хотя бы один из этих инструментов. Конечно, новичок может знать, что такое веб-фреймворк или сервер автоматизации на уровне абстракций, но если он не знает этих конкретных имен, то будет трудно понять смысл и запомнить.

Изучение предметной области отдельно от кода

Изучение релевантных понятий без обращений к коду намного облегчит изучение кода. Кажется мелочью, но это может иметь большое значение. Вы можете даже сде-

лать набор дидактических карточек на тему этой предметной области и концепций программирования, чтобы новичок мог постепенно запоминать новую информацию.

К слову сказать, вне зависимости от процесса адаптации существование актуального списка, в котором указаны все концепции предметной области и программирования, может помочь и коллегам, с которыми вы работаете долгое время.

УПРАЖНЕНИЕ 13.4. Выберите проект, над которым вы чаще всего работаете. Создайте два списка, которые могут быть полезны новичкам: в одном соберите все важные концепции предметной области и их описание, а во втором — все важные библиотеки, фреймворки и концепции программирования, которые задействованы в кодовой базе. Заполните следующую таблицу.

Концепции предметной области	Концепции программирования/библиотеки
Концепт	Определение
Концепт/модель/библиотека	Использование/определение

Поддержка кратковременной памяти: ставьте небольшие конкретные задачи

Еще одна вещь, которая часто в процессе адаптации идет не так, как надо, — это то, как новичок объясняет код. Руководитель показывает код и его релевантные части. Затем он в качестве «знакомства» с базой кода предлагает новичку реализовать простую функцию. То же самое может произойти и с проектом с открытым исходным кодом, где простые функции могут быть отмечены как подходящие для новичков. Хотя звучит классно, это может представлять когнитивную проблему.

Новичка просят выполнить пару действий по программированию, и теперь его мозг выполняет несколько дел: знакомится с кодом, что-то ищет в нем, а также реализует функцию. Это может перегрузить кратковременную память новичка, т. к. он пока что не может ориентироваться в кодовой базе. Много времени он тратить на поиск по коду, а чтение кода отвлекает от поставленной задачи. Поэтому большую задачу лучше всего разбить на несколько маленьких.

От задачи на понимание больше пользы

Если вы хотите, чтобы новичок понял определенный фрагмент кода, то попросите его понять этот фрагмент, а не давайте ему задание на реализацию функции. Например, попросите нового коллегу сделать краткое описание существующего класса или попросите его выписать все классы, которые участвуют в выполнении определенной функции.

Знакомство новичков с целенаправленными задачами будет менее тяжелым процессом для кратковременной памяти. Следовательно, новичок с большей вероятно-

стью сможет запомнить важные сведения о коде. Не дополнительные функции, а хорошее описание кода поможет новичку адаптироваться, а также послужит полезной документацией другим новым сотрудникам.

Если вы захотите, чтобы новичок реализовал небольшую функцию, то он, может быть, справится с этой задачей. Однако в данном случае лучше всего убрать из кода отвлекающие аспекты, которые вызовут дополнительную когнитивную нагрузку, например поиск в коде. Вы можете заранее приготовить нужный код. Вы можете использовать описанные в *главе 4* стратегии, например рефакторинг кода в нужный класс, чтобы исключить поиск.

Поддержка рабочей памяти: используйте диаграммы

В *главе 4* предлагается несколько стратегий по поддержанию рабочей памяти, в том числе и с использованием диаграмм. Новому сотруднику, незнакомому с кодовой базой, может быть трудно создавать подобные материалы. В процессе адаптации наставник может решить создать таблицы, которые будут поддерживать рабочую память.

Однако, как мы писали ранее, диаграммы не всегда полезны для новичков, которые могут не захотеть отойти от кода и посмотреть на всю картину целиком. Отслеживайте пользу диаграмм и перестаньте их использовать в том случае, если данная стратегия не помогает.

13.3.3. Совместное чтение кода

Еще одна стратегия, которой вы можете воспользоваться для адаптации новых работников, — это чтение кода всей командой. В *главе 5* мы рассмотрели семь стратегий для чтения текстов на естественном языке, которые можно использовать для чтения кода:

1. *Активация* — активное размышление о связанных вещах для активации уже имеющихся знаний.
2. *Определение важности* — выделение наиболее важных частей текста.
3. *Постановка предположений* — работа с фактами, которые упомянуты в тексте вскользь.
4. *Наблюдение* — отслеживание вашего понимания текста.
5. *Визуализация* — создание диаграмм или рисунков для углубленного понимания текста.
6. *Постановка вопросов* — подготовка вопросов о тексте.
7. *Резюмирование* — создание краткого пересказа текста.

В *главе 5* мы рассматривали эти стратегии с той точки зрения, когда вы читаете код как отдельный программист. Однако эти стратегии могут быть полезны и в том случае, когда вы знакомите нового программиста с базой кода. Когда команда

выполняет эти действия совместно, то у новичка снижается когнитивная нагрузка, и он может полностью сосредоточиться на коде.

Далее мы подробно рассмотрим, как каждая из этих семи стратегий может использоваться при совместном чтении кода.

Активация

Перед началом чтения ознакомьтесь с нужными концепциями кода. Если вы выполняли упражнение 13.1, то вы можете подготовить список концепций заранее. За некоторое время до чтения кода напомните новичку о нужных концепциях, чтобы он мог разобраться с ними и не путаться в них. Лучше всего сейчас обсудить все детали, а не на этапе, когда новичок будет пытаться понять код. Например, если вы пользуетесь моделью «Модель — Представление — Контроллер», то лучше ознакомить новичка с этой моделью до того, как он сам узнает об этом при просмотре файлов кода.

После этапа активации может начаться совместное чтение кода. Обратите внимание на то, что данная активность — это понимание. Это означает, что в данный момент мы ограничиваем остальные четыре типа активности.

Определение важности

Если у вас недостаточно знаний о чем-либо, вам может быть сложно отличить ненужную информацию от нужной. Выделение наиболее важных частей кода помогает новичкам адаптироваться. Это можно сделать и при совместном чтении кода, например когда все члены указывают на то, что они считают самым важным. В качестве альтернативы у вас может быть создан документ, в котором собрана информация со всех предыдущих совместных чтений кода, в котором уже выделены все важные моменты.

Постановка предположений

Бывает очень сложно понять то, что не лежит на поверхности. Например, у команды программистов может быть четкое понимание концепций предметной области, например: в одной поставке всегда содержится как минимум один заказ, однако такое решение могло не быть задокументировано в коде. Как и с определением важности, вы можете рассказать о принятых решениях на этапе чтения кода. Вы можете также сообщить об этих решениях, чтобы новички могли с легкостью познакомиться с ними.

Наблюдение

Как мы уже писали, самое важное, что вы можете сделать в процессе адаптации новичка, — это отслеживать уровень его понимания в данном моменте времени. Регулярно просите его пересказать прочитанное, определить основные концепции предметной области или вспомнить концепцию программирования, которая использовалась в коде.

Визуализация

Как было описано ранее, диаграммы могут быть полезны в двух случаях. Во-первых, они могут поддерживать рабочую память *(см. главу 4)*, а во-вторых, они могут способствовать пониманию *(см. главу 5)*. В зависимости от того, на каком уровне находится новый работник, наставник может или создать диаграмму и использовать ее для того, чтобы новичку было проще читать код, или попросить его нарисовать диаграмму, чтобы новичок мог лучше понять код.

Постановка вопросов

При совместном чтении кода постоянно задавайте вопросы и отвечайте на вопросы о нем. В зависимости от уровня, на котором находится новый работник, вы можете задавать вопросы, на которые он должен ответить, или вы можете просить его задавать вам вопросы, на которые вы будете отвечать. Новичкам с опытом может быть полезно задавать вопросы, на которые они будут искать ответы вместе с вами; однако всегда следите за их когнитивной нагрузкой при решении тех задач, с которыми вы им помогаете меньше. Когда человек начинает гадать или делает выводы, которые не имеют никакого смысла, то это значит, что он испытывает слишком высокую когнитивную нагрузку.

Резюмирование

В качестве последнего этапа совместного чтения кода вы можете резюмировать код. Пример показан на рис. 13.4. В зависимости от состояния документации в кодовой базе этот краткий пересказ впоследствии может стать частью документации кода — отличная задача для новичка после окончания чтения кода. В процессе этого новички познакомятся с рабочими потоками и документооборотом кодовой базы (например, создав запрос на принятие изменений или запросив обзор, причем не перегружая при этом свою рабочую память).

Транспиляция Hedy — это поэтапный процесс. Сначала код разбирается с помощью Lark, в результате чего получается абстрактное синтаксическое дерево. Затем абстрактное синтаксическое дерево проверяется на предмет некорректных правил. Если они появляются в коде, то программа Hedy становится недействительной, после чего появляется сообщение об ошибке. Затем из абстрактного синтаксического дерева берется таблица поиска с именами всех переменных, находящихся в коде. Наконец, абстрактное синтаксическое дерево преобразуется на языке Python путем добавления необходимого синтаксиса, например скобок.

Рис. 13.4. Пример резюмирования кода

Выводы

- Профессиональные программисты думают и работают не так, как начинающие. Профессионалы могут абстрактно рассуждать о коде и думать о нем, не работая с самим кодом. Начинающие же, как правило, сильно сосредоточиваются на деталях кода.

- Когда среднего уровня программисты узнают новую информацию, то они могут вернуться к мышлению, которое у них было, когда они были начинающими.
- Люди, которые учат новую концепцию, должны изучать ее не только как абстрактное понятие, но и на конкретных примерах.
- Людям, которые учат новую концепцию, необходимо время на то, чтобы в мозгу образовались схемы между новой информацией и имеющимися знаниями.
- При адаптации новичка ограничьте количество его задач программирования до одной в отдельно взятый момент времени.
- При адаптации новичка приготовьте нужную информацию для поддержки его долговременной, кратковременной и рабочей памяти.

Эпилог. Пара слов перед прощанием

Вне зависимости от того, прочитали вы эту книгу полностью или выборочно некоторые главы, я рада, что вы сейчас здесь. Работа над этой книгой была для меня очень полезным опытом. В ходе работы над книгой я узнала много нового о когнитивистике и программировании. Но еще больше я узнала о себе. Работа над книгой позволила мне осознать кое-что важное: чувства замешательства и когнитивной подавленности — это нормально, это часть нашей жизни и обучения. До того как я узнала все, что знаю о когнитивистике теперь, я расстраивалась из-за того, что считала себя недостаточно умной для чтения сложных статей или работы над незнакомым кодом; теперь же я с уверенностью могу сказать себе: «Не волнуйся, возможно, ты просто испытываешь слишком высокую когнитивную нагрузку».

Я также начала работу над проектом, который в конце концов стал языком программирования Hedy. По правде говоря, я никому бы не рекомендовала совмещать создание нового языка программирования и написание книги, но у меня эти две активности были тесно связаны между собой и хорошо сочетались тематически. Многие упражнения из этой книги, например по снижению когнитивной нагрузки, работе с неверными представлениями, семантическим волнам или интервальному повторению, также реализованы в языке Hedy. Для меня будет огромной честью, если вы, обучая своих детей программированию, познакомите их с Hedy. Сделать это можно на сайте **www.hedycode.com**. Это бесплатно, а еще все исходные коды открыты!

В заключение я хочу подчеркнуть, что мне очень понравилось искать, изучать и рассказывать о работе великих ученых в области программирования и когнитивистики. Хотя мне очень интересно проводить собственные исследования, гораздо важнее было познакомить разработчиков с существующими исследованиями, которые могут сильнее повлиять на среду программирования, чем мои собственные проекты. Если вы хотите почитать больше на эту тему, то ниже я приведу несколько книг, которые я рекомендую, и назову некоторых ученых, с работами которых вы, возможно, захотите ознакомиться.

Если вы хотите узнать больше о своем мозге, то я смело советую вам книгу «*Думай медленно... Решай быстро*» (АСТ, 2014) Даниэля Канемана, потому что из его книги вы узнаете намного больше информации о мозге, чем из моей. Тема мозга также хорошо раскрывается в книге «*How We Learn*» (Penguin, 2015) Бенедикта Керри (Benedict Carey): в книге подробно рассматривается интервальное повторение и па-

мять. Читателям, интересующимся изучением математики, я могу посоветовать книгу «*How the Brain Learns Mathematics*» (Corwin, 2014) Давида Соуза (David Sousa), в которой вы найдете множество интересных исследований по изучению математики и абстракций. По программированию я советую книгу «*A Philosophy of Software Design*» (Yaknyam Press, 2018) Джона Оустерхаута (John Ousterhout). Книга тяжело читается, однако в ней дается подробное описание того, как разрабатывается ПО. Мне также очень нравится книга «*Software Design Decoded*» (MIT Press, 2016) Андре ван де Хука (Andre van der Hoek) и Мариан Петре, в которой собраны 66 способов мышления опытных программистов и которую можно рассматривать в качестве набора дидактических карточек и использовать в различных ситуациях. Я рассказывала об этой книге в одном их выпусков SE Radio.

Если вы хотите почитать научные статьи по темам, освещаемым в моей книге, то я могу рекомендовать вам две статьи, которые сильно повлияли на меня: «Why Minimal Guidance During Instruction Does Not Work» Киршнера (Kirschner) и др.[1], в которой ставится под сомнение все, что я знала об обучении и преподавании, и «Toward a Developmental Epistemology of Computer Programming» Раймонда Листера, в которой точь-в-точь описывается мой опыт преподавания программирования; эта работа научила меня преподавать лучше[2].

В этой книге собрано множество работ выдающихся ученых, но даже такого количества страниц недостаточно для того, чтобы осветить всю их работу. Если вы хотите узнать подробнее об исследованиях понимания программ, то вы можете подписаться на аккаунты этих прекрасных и выдающихся людей в «Твиттере»: Сара Фахури (@fakhourysm), Александр Серебреник (@aserebrenik), Крис Парнин (@chrisparnin), Джанет Зигмунд (@janetsiegmund), Британни Джонсон (@DrBrittJay), Титус Бэрик (@barik), Дэвид Шеперд (@davidcshepherd) и Эми Ко (@amyjko).

[1] «Making Semantic Waves: A Key to Cumulative Knowledge-Building», *Linguistics and Education*, ч. 26, № 1, стр. 8–22, 2013, Карл Мэтон, **https://www.sciencedirect.com/science/article/pii/S0898589812000678**.

[2] «Toward a Developmental Epistemology of Computer Programming», Раймонд Листер (2010), **https://dl.acm.org/doi/10.1145/2978249.2978251**.

Предметный указатель

A

Anki 64
Apache 166
APL 28, 29, 36, 126, 202, 240

B

BASIC 28, 30, 37, 76, 89, 148, 202, 242
BCPL 104

C

C 143, 166
C# 147, 148, 176
C++ 70, 166, 176
CDCB 235
CDN 234
Cerego 64
COBOL 240
CSS 62

E

Eclipse 163, 166, 187
Eclipse Checkstyle 191
Excel 104, 105, 240

F

F# 149
FindBugs 177
FlowLight 230
fNIRS 194–196, 224
Fortran 166

G

gcc 166
GitHub 51, 109, 167, 173, 215, 227
Groovy 235

H

Haskell 153, 235, 241
Hedy 265
Hibernate 177
HTML 237

I

Idris 239
IntelliSense 104

J

Java 28, 29, 32, 34, 36, 40, 43, 61, 76, 82, 91, 101, 111, 126, 137, 144, 148, 152, 165, 166, 168, 173, 176, 177, 187, 191, 196, 202, 207, 209, 235, 240
JavaScript 62, 84, 117, 119, 145, 146, 150, 173, 235, 237, 240
JUnit 168

K

Kinect 114

L

LAPD 191
Logo 136

M

Microsoft 104
MySQL 166

N

Naturalize 168

P

PEP8 165
Perl 173
PHP 173
Prolog 154
PyCharm 238
Python 55, 63, 73, 83, 84, 90, 98, 102, 112, 143, 144, 152, 165, 207, 235, 236, 240, 242, 256
Python Tutor 90

Q

Quizlet 64

S

Samba 166
Scala 147
Scratch 140, 239
Slack 223, 226, 231
Smalltalk 239
SQL 150
STEM 113
StringBuldier 202

T

Tomcat 177

V

Visual Basic 104
Visual Studio 227

W

Windows 105
Word 104

А

Абстрактная модель 131
Автоматизация 206, 210
Алгол 48, 52
Анонимная функция 82
Анонимный оператор 85
Антипаттерны проектирования
◊ лингвистические 190, 191
◊ семантические 190
◊ структурные 190
Ассемблер 242
Ассоциативный этап 208

Б

Биометрика 193

В

Венгерская нотация 104, 165
◊ прикладная 105
◊ системная 104
Внутренняя сложность 78
Встраивание кода 81
Вычислительная мощность 30, 75

Г

Гемоглобин 194
Генератор списков 73, 83
Граф
◊ зависимостей 86, 91, 124, 132
◊ неориентированный 134
◊ ориентированный 134

Д

Делокализованный код 81
Диаграмма состояний 124
Диадическое кодирование 32
Дидактическая карточка 63, 64, 68, 85, 103, 133, 204, 207, 211

З

Запахи кода 184
Зрачок 193

И

Интервальное повторение 67
Исходное состояние 200

К

Кислород 194
Класс всемогущий 186
Ключевое слово 52
Когнитивная компиляция 90
Когнитивная нагрузка 77, 86, 107, 108, 125, 129, 132, 168, 178, 183, 188, 191, 192, 194, 195, 207, 210, 214, 224, 228, 230, 236, 238, 247, 250, 258, 261, 262
◊ внешняя 79, 81, 82, 84, 179
◊ внутренняя 78
◊ измерение 191, 193
Когнитивные измерения 234
Когнитивный процесс 27, 31–34, 39, 75, 170
Когнитивный этап 208
Кодирование 66
Кожа 193
Компиляция 148
Концептуальная замена 152
Крестики-нолики 200
Кривая забывания 67, 68, 71

Л

Лексика проекта 181
Лингвистические антипаттерны 195, 240
◊ проектирования 236
Локальная модель 132
Лондонский Тауэр 149
Лямбда-функция 82

М

Маячки 55
Ментальная модель 126, 1281–133, 135, 138, 148, 152
Модель 124

Н

Нейровизуализация 110
Неопиажизм 251
Нотации 234
НТИМ 113

О

Оксигенация 194

П

Память
◊ декларативная 204
◊ долговременная 31, 33, 39, 43, 46, 65, 68–70, 116, 118, 125, 132, 133, 135, 141, 144, 148, 165, 170, 179, 189, 202, 203, 214, 221, 241, 251, 257, 259
◊ иконическая 50, 51
◊ имплицитная 203, 205, 207, 208, 211
◊ кратковременная 31, 32, 39, 40, 44, 47, 51, 65, 77, 144, 168, 169, 173, 213, 220, 241, 260
◊ мышечная 31
◊ проспективная 226
◊ процедурная 203, 205
◊ рабочая 31, 33, 39, 72, 75, 77, 81–83, 85, 86, 111–113, 116, 118, 125, 130, 132, 133, 135, 144, 168, 169, 179, 188, 189, 203, 213, 221, 222, 241, 250, 251, 261
◊ семантическая 204
◊ сенсорная 49, 144, 169
◊ событийная 147, 251
◊ эксплицитная 204, 207
◊ эпизодическая 204, 205, 210
Паттерн проектирования 53
Переменная 98
Перенос 144
◊ модели обучения 145
Плагин 227
Полиндром 202
Поля Бродмана 110
Проверяемые исключения 148
Проектный маневр 244
Проработка 71, 73, 144
Пространство состояний 200

Р

Регистр
◊ верблюжий 176
◊ змеиный 176
Резюмирование 215
Рефакторинг 80, 88, 184, 221, 222
◊ когнитивный 81
◊ обратный 81
Роль переменных 98

С

C 104, 173, 176, 195, 224, 235
C# 172
C++ 63, 85, 236, 241
Семантика 142
Семантическая волна 256
Сеть Петри 134
Синтаксис 61, 63, 165, 240
Соответствующая нагрузка 214
Сортировка вставками 40
Срез кода 108
Статический анализ 177
Стратегия понимания текста 114
Структурные антипаттерны 196
Схема 72, 98, 133, 141, 208

Т

Таблица
◊ операций 119
◊ состояний 89, 91, 132
Теория
◊ чанков 45, 47
◊ экземпляров 210
Тернарный оператор 82, 84
Точка фокуса 106–108
Трансференция 144, 196
◊ близкая 147
◊ виды 147
◊ дальняя 147, 150
◊ негативная 150, 151, 155, 205
◊ неосознанная 147
◊ обучения 144, 145
◊ осознанная 147
◊ отрицательная 148
◊ положительная 148
Трассировка 33, 90

У

Уровень
◊ воспроизведения 69, 71, 72
◊ хранения 69
Условная машина 135, 138, 141

Ф

Формула забывания 67
Функциональная МРТ (фМРТ) 111, 194

Х

Ханойская башня 149

Ц

Целевое состояние 200

Ч

Чанк 47, 52, 55, 58, 62, 64, 69, 77, 168, 177, 188
Черепашья графика 136

Ш

Шаблон вызова 108
Шаблоны имен 178
Шахматы 45, 46, 149
Шкала Пааса 192

Э

Электроэнцефалограмма 194
ЭЭГ 112, 113, 194

УМ ПРОГРАММИСТА

КАК ПОНЯТЬ И ОСМЫСЛИТЬ ЛЮБОЙ КОД

Понимая, как именно работает мозг, можно значительно улучшить навыки программирования. Умения, подкрепленные достижениями когнитивистики, помогут вам быстрее изучать новые языки программирования, повысить продуктивность, значительно сократить количество итераций при написании кода. Перед вами — уникальная книга, рассказывающая, как именно это сделать, помогающая понять, как именно программный код укладывается в голове. Здесь описаны научно обоснованные приемы, которые пригодятся вам, чтобы эффективно осваивать новые технологии, осмысливать код и запоминать синтаксис. Вы узнаете, что и из рабочих пробуксовок можно извлечь пользу, а недопонимание тоже одна из граней обучения.
Попутно вы узнаете, как по ходу работы накапливать учебные материалы, станете экспертом в самообучении и профессиональном наставничестве.

Прочитав эту книгу, вы

- узнаете, как именно ваш мозг видит код;
- научитесь бегло читать и быстро усваивать код;
- убедитесь, что есть простые приемы, позволяющие распутать самый сложный код;
- сможете навести порядок в любой, даже самой запущенной базе кода.

Эта книга особенно заинтересует программистов, работающих сразу с несколькими языками.

Доктор Фелин Херманс — приглашенный профессор в Лейденском университете в Нидерландах. Вот уже более 10 лет она исследует механизмы программирования, выясняя, как лучше учиться этому искусству и обучать других.

"Великолепная и глубоко информативная книга, помогает перекинуть мостик между программированием и мышлением."
— Майк Тейлор, компания CGI Inc.

"Книга о том, как именно работает мозг, как эффективнее читать, писать и совместно разрабатывать код."
— Бен Мак-Намара, консалтинговое агентство DataGeek

"В книге изложен научный подход, помогающий программисту разгрузить голову при работе и самому себе проторить путь к вершинам мастерства!"
— Даниэла Запата Риеско, стартап M1 Finance

"Если вы когда-нибудь задумывались, как брать умом, а не только усердием — прочтите эту книгу. Я каждый день убеждаюсь, насколько она помогла мне в работе."
— Чжицзюнь Лю, рекламный холдинг Mediaocean

bhv®
191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

MANNING

ISBN 978-5-9775-1176-6