Часть I
Очистка

В обучении я предпочитаю переходить от конкретного к абстрактному. По этой причине мы начнем с каталога небольших «приемов» дизайна, которые можно применить к проблемному коду, который требуется изменить.

Читатели, знакомые с рефакторингом, увидят значительное сходство между рефакторингом, определяемым как изменения в структуре, не изменяющие поведения, и очисткой. Методы очистки составляют подмножество методов рефакторинга. Это своего рода маленькие пушистые рефакторинги, перед которыми никто не устоит.

Термин «рефакторинг» фатально пострадал, когда люди стали применять его для обозначения длинных перерывов в разработке функциональности. Затем они даже убрали уточнение «не изменяющие поведения», так что такой «рефакторинг» может легко сломать систему. Что получается? Никакой новой функциональности, возможный

ущерб, и в итоге ничего, что оправдывало бы неудобства. Нет уж, спасибо.

В части II речь пойдет об интеграции очистки в процесс разработки. А пока читайте, учитесь и применяйте на практике приемы, благодаря которым заниматься разработкой будет приятнее.

ГЛАВА 1

Охранные выражения (guard clauses)

Вам встретился код, который выглядит так:

```
if (условие)
    ...некий код...
```

Или даже так:

```
if (условие)
    if (not другое условие)
        ...код...
```

При чтении такого кода легко запутаться во вложенных условиях. Почистите его и приведите к следующему виду:

```
if (not условие) return
if (другое условие) return
...код...
```

Такой код проще читать. По сути, он говорит: «Прежде чем переходить к деталям, зададим предварительные условия, которые необходимо учитывать».

```
(Но как быть с наличием нескольких операторов return?
«Правило» о единой точке возврата из функции появилось
в эпоху FORTRAN, когда одна процедура могла иметь
несколько точек входа и выхода. Отладить такой код
было невозможно. Просто невозможно определить, какие
инструкции были выполнены. Код с охранными выражениями
проще анализировать, потому что предусловия в них явно
выражены.)
```

Не переусердствуйте с охранными выражениями. Функцию с семью или восемью охранными выражениями (а я встречал такие на практике) читать *ничуть* не проще. Она определенно нуждается в секционировании сложности.

Выделяйте проверки в охранные выражения только в том случае, если условие четко формулируется:

```
if (условие)
    ...остальной код процедуры...
```

Иногда бывает код, который я хотел бы очистить, но не могу:

```
if (условие)
    ...код...
...другой код...
```

Возможно, первые две строки можно выделить во вспомогательный метод (метод-хелпер), и *после этого* охранное выражение можно будет почистить, но очистку кода *всегда* и *без исключений* следует делать только небольшими шагами.

Пример: https://github.com/Bogdanp/dramatiq/pull/470

ГЛАВА 2

Мертвый код

Удалите его. Да, это весь совет. Если код не выполняется, просто удалите его.

Для кого-то это может показаться странным. В конце концов, автор кода не пожалел времени и усилий, чтобы написать его. Организация за него заплатила. И вот он здесь. Чтобы код снова стал полезным, нужно совсем немного: снова вызвать его. Если этот код когда-нибудь нам опять понадобится, мы пожалеем, что удалили его.

Возлагаю на вас, мой читатель, задачу определить все когнитивные искажения, которые я только что продемонстрировал.

Иногда найти мертвый код легко. А иногда — если активно применяется отражение (reflection) — не очень. Если вы подозреваете, что какой-то код не используется, подготовьте его к очистке, регистрируя случаи его употребления в журнале. Добавьте эту подготовку на продакшен и подождите, пока не будете полностью уверены.

Кто-то спросит: «А если потом он нам понадобится?» Для этого существуют системы контроля версий. На самом деле ничего не удаляется — вы просто не будете видеть этот код. Если (и это довольно длинная цепочка условий): 1) у вас есть большой объем кода, который 2) не используется прямо сейчас и который 3) вы планируете использовать в будущем 4) точно так же, как он был написан, и 5) он все еще работает — да, вы всегда сможете его вернуть. А можно написать его заново и лучше. Но в крайнем случае вернуть.

Как обычно, на каждом шаге очистки следует удалять только небольшие фрагменты. Если вдруг окажется, что вы допустили ошибку, изменение можно будет относительно легко отменить (см. главу 28). «Небольшие» — когнитивная метрика, а не количество строк. Это может быть одно условие в условной конструкции (например, вы видите, что условие всегда преобразуется в `true`), одна процедура, один файл, один каталог.

ГЛАВА 3

Нормализация симметрий

Код растет органично. Некоторые используют определение «органично» как ругательство. Мне это кажется глупым. Невозможно сразу написать весь код, который вам когда-либо понадобится. Можно, но только если бы мы ничему не учились.

При органичном росте одна задача может по-разному решаться в разное время или разными людьми. И это нормально, но программы становится сложнее читать. Читатели ценят последовательность. Заметив закономерность, они могут быть уверены, что поняли, что происходит.

Для примера возьмем отложенную инициализацию переменных. Ее можно записать по-разному:

```
foo()
    return foo if foo not nil
    foo := ...
    return foo
```

```
foo()
    if foo is nil
        foo := ...
    return foo
```

```
# Неочевидно
foo()
    return foo not nil
        ? foo
        : foo := ...

# Вдвойне неочевидно; предполагается, что присваивание
является выражением
foo()
    return foo := foo not nil
        ? foo
        : ...

# И еще хитрее со скрытой проверкой условия
foo()
    return foo := foo || ...
```

(Попробуйте найти или придумать другие варианты.)

По сути, все эти варианты означают: вычислить и кэшировать значение для `foo`, если его еще нет. У каждого из них есть достоинства и недостатки. Вы как читатель быстро привыкнете к любому. Сложнее, если в коде используются два и более паттерна. Читатель ожидает, что разная запись подразумевает разный смысл. Здесь же различия только скрывают тот факт, что на самом деле суть одна и та же.

Выберите один вариант. Преобразуйте к нему все остальные. Вычищайте излишнюю вариативность по одной за раз — начните, например, с отложенной инициализации.

Нередко общее скрывается за лишними деталями. Ищите процедуры, которые похожи друг на друга, но не идентичны. Отделяйте разные части от одинаковых.

ГЛАВА 4

Новый интерфейс, старая реализация

Итак, вам нужно вызвать процедуру, а из-за интерфейса это слишком сложно/неочевидно/запутанно/утомительно. Реализуйте интерфейс, который вам удобен, и вызывайте его. Реализуйте новый интерфейс простым вызовом старого (реализацию можно будет подставить позже, после миграции всех остальных вызывающих сторон).

Создание сквозного интерфейса — квинтэссенция программного дизайна. Вам нужно изменить поведение. Если бы дизайн был таким и сяким, то внести изменение было бы просто (проще). Значит, сделайте дизайн именно таким.

Этот же принцип справедлив и в других случаях.

- Подход «начать с конца» — начните с последней строки процедуры, так как у вас уже есть все промежуточные результаты.

- Подход «сначала тесты» — начните с теста, который должен проходить.
- Проектирование хелперов — если бы у меня была процедура/объект/сервис, который делает *XXX*, все остальное было бы просто.