Почему сложно оценить последствия рефакторинга

Оценить состояние кодовой базы можно по-разному. Но проведенный рефакторинг не всегда сдвигает эти метрики в положительном направлении просто потому, что они не связаны с подлежащими устранению болевыми точками. Поэтому для измерения начального состояния кодовой базы важно выбрать метрику, дающую наглядное представление о проблеме и подчеркивающую воздействие рефакторинга.

Измерить последствия рефакторинга сложно. В первую очередь потому, что успешно выполненный рефакторинг обязан быть невидим для пользователей и никак не должен менять поведение кода. Это же не новый функционал, который повысит привлекательность приложения для пользователей. Для обеспечения надежности работы приложений в них часто добавляется мониторинг за критически важными частями. Но в этом случае отслеживаются показатели, фиксирующие заметное пользователям поведение. И корректно проведенный рефакторинг никак на них не влияет. Значит, нужно найти метрики, точно характеризующие те нюансы кода, которые мы хотим улучшить, и определить точку отсчета. Только после этого можно будет двигаться дальше.

РЕФАКТОРИНГ ДЛЯ УЛУЧШЕНИЯ ПРОИЗВОДИТЕЛЬНОСТИ

Представим небольшое приложение для отслеживания заказов. Чтобы обеспечить его бесперебойную работу, мы наблюдаем за таким показателем, как время получения информации о статусе заказа, по его идентификатору. Через несколько месяцев этот показатель увеличивается, и принимается решение провести рефакторинг. Здесь у нас уже есть начальная метрика — среднее время получения ответа на запрос. Если после внедрения новой версии кода оно уменьшится, то усилия были не напрасны!

Часто простейшая метрика для оценки эффективности рефакторинга — это производительность. В таком случае у нас сразу есть надежный набор исходных показателей. Еще стоит отметить, что усилия для повышения производительности кода, в отличие от тех, что вызваны желанием сделать эффективнее труд разработчиков, относятся к одному из немногих видов рефакторинга, дающему отчетливые улучшения, видимые пользователям.

Особенно трудно поддаются количественной оценке результаты крупного рефакторинга. Он редко ограничивается несколькими неделями. Чаще всего этот процесс выходит далеко за рамки типичного цикла разработки функций. И если на время рефакторинга разработка продукта не была полностью приостановлена, будет сложно определить, какие именно действия влияют на показатели. Разумнее всего пользоваться набором разных метрик для получения более целостной картины прогресса после рефакторинга и отделить эти изменения от параллельно вносимых разработкой.

Оценка сложности кода

Для многих рефакторинг — это средство облегчения поддержки приложений и создания нового функционала и, как следствие, повышения продуктивности труда разработчиков. На практике это часто означает упрощение сложных, запутанных фрагментов кода. А раз мы ставим цель *уменьшить* сложность кода, нужно найти эффективный способ ее измерения. Количественная оценка сложности кода станет отправной точкой для оценки будущего прогресса.

Измерение сложности упрощают две вещи. Во-первых, если есть история версий, можно легко путешествовать в прошлое и вычислять сложность в любом временном интервале. Во-вторых, на многих языках программирования существует огромное число библиотек и инструментов с открытым исходным кодом, позволяющих получить отчет для всего приложения.

Давайте рассмотрим три распространенных метода расчета сложности кода.

Метрики Холстеда

В 1975 году Морис Холстед впервые предложил измерять сложность программного обеспечения путем подсчета операторов и операндов в компьютерной программе. По его мнению, поскольку программы в основном состоят из этих двух компонентов, подсчет их уникальных экземпляров может дать реальное представление о размере программы и, следовательно, о ее сложности.

Операторы — это конструкции, которые ведут себя как функции, но синтаксически или семантически отличаются от них. Можно выделить ариф-

метические и логические операторы, операторы сравнения и присваивания. Рассмотрим простую функцию, складывающую два числа.

Пример 3.1. Короткая функция сложения

```
function add(x, y) {
  return x+y;
}
```

Она содержит единственный оператор + и два операнда. Операнды — это любые объекты, с которыми мы совершаем разные действия при помощи операторов. Операнды в нашем примере — переменные x и y.

Холстед предложил использовать информацию о количестве операторов и операндов для расчета следующих характеристик.

1. Объем программы или нужное количество информации для понимания ее назначения.
2. Сложность программы или умственные затраты программиста на создание кода.
3. Количество ошибок, которые, скорее всего, будут обнаружены в системе.

Для примера возьмем функцию, вычисляющую простые множители целых чисел. Уникальные операторы и операнды и количество раз, которое они встречаются в программе, перечислены в табл. 3.1.

Пример 3.2. Факторизация целых чисел

```
function primeFactors(number) {
  function isPrime(number) {
    for (let i = 2; i <= Math.sqrt(number); i++) {
      if (number % i === 0) return false;
    }
    return true;
  }

  const result = [];
  for (let i = 2; i <= number; i++) {
    while (isPrime(i) && number % i === 0) {
      if (!result.includes(i)) result.push(i);
      number /= i;
    }
  }
  return result;
}
```

Таблица 3.1. Уникальные операторы, операнды и количество их появлений в программе

Оператор	Количество появлений	Операнд	Количество появлений
function	2	0	2
for	2	2	2
let	2	primeFactors	1
=	3	number	7
<=	2	isPrime	2
()	4	i	12
.	3	Math	1
++ (постфикс)	2	sqrt	1
if	2	FALSE	1
===	2	TRUE	1
%	2	result	4
return	3	<anonymous>	1
const	1	includes	1
[]	1	push	1
while	1		
&&	1		
! (префикс)	1		
/=	1		
Уникальных операторов: 18	Всего операторов: 35	Уникальных операндов: 14	Всего операндов: 37

Приведенный код содержит 18 уникальных операторов (n_1), 14 уникальных операндов (n_2) и всего операндов 37 (N_2). Вот предложенная Холстедом формула вычисления относительной сложности чтения программы:

$$D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}$$

Подстановка значений даст следующий результат:

$$D = \frac{18}{2} \cdot \frac{37}{14}$$

$$D = 23{,}78$$

Сама по себе эта цифра особого значения не имеет. Но работая с отдельными фрагментами кода, мы постепенно сможем понять, как эта оценка соотносится с нашим опытом. Со временем, сопоставляя вычисляемые значения с реализациями кода, мы научимся интерпретировать их в контексте приложения.

Эта метрика, как и две другие, о которых я расскажу ниже, может применяться в разных масштабах — для оценки сложности отдельной функции или целого модуля. Метрика сложности Холстеда рассчитывается даже для всего файла, например, как сумма сложностей всех входящих в него функций.

Цикломатическая сложность

Предложенная в 1976 году Томасом Маккейбом цикломатическая сложность — количество линейно независимых маршрутов через программный код. Это подсчет операторов потока управления в программе. Сюда входят операторы `if`, циклы `while` и `for` и блоки `case` в конструкциях `switch`.

Для начала возьмем простую программу без управляющих конструкций (пример 3.3). Для вычисления цикломатической сложности присвоим 1 объявлению функции. Этот параметр должен увеличиваться с каждой встречной точкой принятия решения. В нашем примере через функцию есть только один путь. Соответственно, ее цикломатическая сложность равна 1.

Пример 3.3. Функция пересчета температуры

```
function convertToFahrenheit(celsius) {
  return celsius * (9/5) + 32;
}
```

Для более сложного примера возьмем уже знакомую функцию разложения на простые множители. В примере 3.4 мы нумеруем каждую точку потока управления, получив в итоге цикломатическую сложность 6.

Пример 3.4. Факторизация целых чисел

```
function primeFactors(number) { ❶
  function isPrime(number) {
    for (let i = 2; i <= Math.sqrt(number); i++) { ❷
      if (number % i === 0) return false; ❸
    }
```

```
    return true;
  }

  const result = [];
  for (let i = 2; i <= number; i++) { ❹
    while (isPrime(i) && number % i === 0) { ❺
      if (!result.includes(i)) result.push(i); ❻
      number /= i;
    }
  }
  return result;
}
```

❶ Первая управляющая точка — объявление функции.

❷ Вторая — первый цикл `for`.

❸ Третья — первый оператор `if`.

❹ Четвертая — второй цикл `for`.

❺ Пятая — цикл `while`.

❻ Шестая — второй оператор `if`.

Каждый раз, когда при чтении кода появляется ветвление (оператор `if`, цикл `for` и т. п.), возможны несколько путей выполнения. Для понимания того, что делает код, нужно уметь удерживать в голове больше информации. Цикломатическая сложность 6 позволяет сделать вывод, что функция `primeFactors`, вероятно, не так уж сложна для чтения и понимания.

Подсчет числа точек принятия решения — это упрощение предложенного Маккейбом метода расчета сложности программы. Математически мы можем вычислить цикломатическую сложность структурированной программы, создав ориентированный граф, описывающий ее поток управления. Каждый узел — это базовый блок (прямолинейная кодовая последовательность без ветвей), связывающие узлы ребра показывают способ перехода от одного блока к другому. Сложность M определяется по формуле:

$$M = E - N + 2P,$$

где E — количество ребер, N — количество узлов, а P — количество компонентов связности. Компонентом связности называется подграф, в котором все узлы достижимы друг для друга.

На рис. 3.1 показан поток управления для функции `primeFactors`.

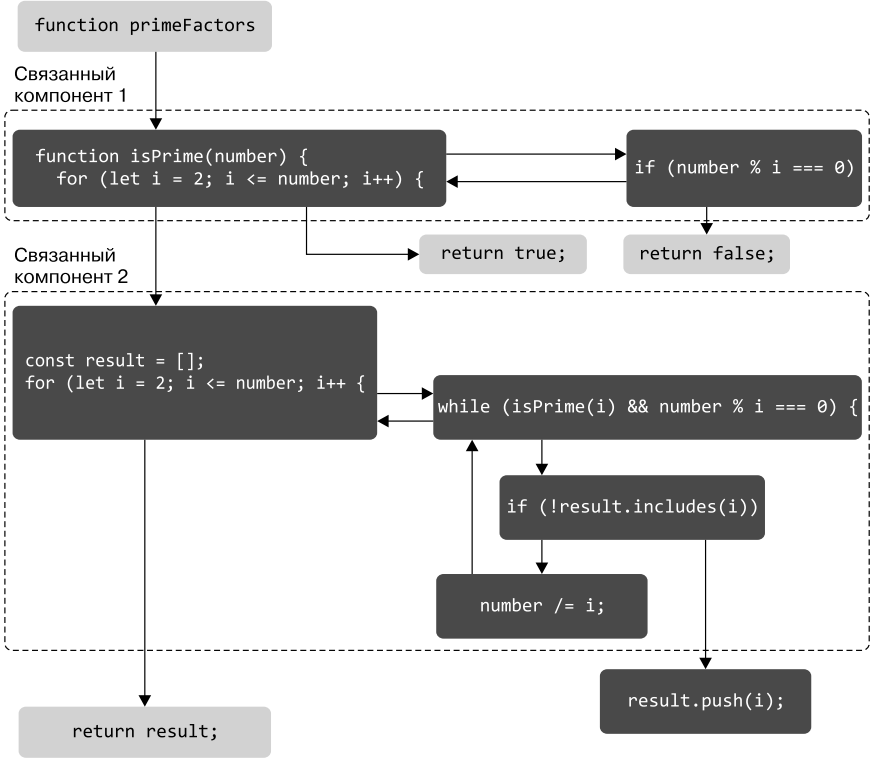

Рис. 3.1. Граф потока управления функции primeFactors, на котором синие узлы обозначают нетерминальные, а красные — терминальные состояния. Здесь 13 ребер, 11 узлов и 2 компонента связности

ДРУГИЕ ВАРИАНТЫ ПРИМЕНЕНИЯ ГРАФОВ ПОТОКА УПРАВЛЕНИЯ

Графы потока управления (CFG) помогают не только в расчете сложности программного обеспечения. Сталкиваясь с особо запутанными потоками управления, я часто рисовала эти графы вручную, чтобы выделить точки принятия решений. Конечно, есть инструменты автоматической генерации CFG. Но чтобы начертить их самостоятельно, нужно внимательно читать код, дающий представление о потоке управления.

Еще CFG позволяют эффективно определить недоступный код. Наличие подграфа, не связанного ни с одной точкой входа, позволяет предположить, что соответствующий код недоступен, и его можно удалить. С другой стороны, недоступность блока выхода из точки входа может указывать на бесконечный цикл.