Кузнецова Яна ИСП-31 (11 вариант)

Мутационное тестирование — это метод тестирования ПО, основанный на всевозможных изменениях исходного кода и проверке реакции на эти изменения набора автоматических тестов. Если тесты после изменения кода успешно выполняются, значит либо код не покрыт тестами, либо написанные тесты неэффективны. Критерий, определяющий эффективность набора автоматических тестов, называется Mutation Score Indicator (MSI).

Для применения этой технологии у нас, очевидно, должен быть исходный код (source code), некоторый набор тестов (для простоты будем говорить о модульных — unit tests).

После этого можно начинать изменять отдельные части исходного кода и смотреть, как реагируют на это тесты.

Одно изменение исходного кода будем называть Мутацией (Mutation). Например, изменение бинарного оператора "+" на бинарный "-" является мутацией кода.

Результатом мутации является Мутант (Mutant) — то есть это новый мутированный исходный код.

Каждая мутация любого оператора в вашем коде (а их сотни) приводит к новому мутанту, для которого должны быть запущены тесты.

Кроме изменения "+" на "-", существует множество других мутационных операторов (Mutation Operator, Mutator) — отрицание условий, изменение возвращаемого значения функции, удаление строк кода и т.д.

Итак, мутационное тестирование создает множество мутантов из вашего кода, для каждого из них запускает тесты и проверяет, выполнились они успешно или нет. Если тесты упали — значит всё хорошо, они отреагировали на изменение в коде и поймали ошибку. Такой мутант считается убитым (Killed mutant). Если тесты выполнились успешно после мутирования — это говорит о том, что либо ваш код не покрыт в этом месте тестами вовсе, либо тесты, покрывающие мутированную строку, неэффективны и в недостаточной степени тестируют данный участок кода. Такой мутант называется выжившим (Survived, Escaped Mutant).

Важно понимать, что мутационное тестирование это не хаотичное преобразование кода, а абсолютно предсказуемый и понятный процесс, который, при наличии одинаковых входных мутационных операторов, всегда выдает один и тот же список мутаций и результирующие метрики на одинаковом тестируемом исходном коде.

Рассмотрим пример. Будем использовать мутационный фреймворк (МФ) для PHP — [Infection](https://infection.github.io/guide/).

Пусть у нас есть какой-то фильтр, умеющий отфильтровывать коллекцию пользователей по признаку совершеннолетия, написанный в объектно-ориентированном стиле:

class UserFilterAge

{

const AGE\_THRESHOLD = 18;

public function \_\_invoke(array $collection)

{

return array\_filter(

$collection,

function (array $item) {

return $item['age'] >= self::AGE\_THRESHOLD;

}

);

}

}

И для этого фильтра есть юнит тест:

public function test\_it\_filters\_adults()

{

$filter = new UserFilterAge();

$users = [

['age' => 20],

['age' => 15],

];

$this->assertCount(1, $filter($users));

}

Сгенерированные мутации.

Первая мутация:

class UserFilterAge

{

const AGE\_THRESHOLD = 18;

public function \_\_invoke(array $collection)

{

return array\_filter(

$collection,

function (array $item) {

- return $item['age'] >= self::AGE\_THRESHOLD;

+ return $item['age'] > self::AGE\_THRESHOLD;

}

);

}

}

Запущенные для нее тесты выполняются успешно. То есть изменение исходного кода абсолютно никак не отразилось на результатах теста. Это не то, что нам надо.

Мутационное тестирование сказало нам, что мы можем взять и заменить условие с ">=" на ">", и программа будет работать точно также. Помните, юнит тесты гарантируют нам, что программа работает так, как мы этого хотим? А раз тесты выполнились успешно с таким мутированным кодом, значит мы и ожидаем такое поведение.

Из этой мутации видно, что, тестируя код с условиями на интервалы, всегда надо проверять граничные значения.

Давайте исправим ситуацию и убьём мутанта:

/\*\*

\* @dataProvider usersProvider

\*/

public function test\_it\_filters\_adults(array $users, int $expectedCount)

{

$filter = new UserFilterAge();

$this->assertCount($expectedCount, $filter($users));

}

public function usersProvider()

{

return [

[

[

['age' => 15],

['age' => 20],

],

1

],

[

[

['age' => 18],

],

1

]

];

}

Мы добавили один тест на граничное значение — 18. Теперь, если опять запустить тесты с мутированным кодом, они упадут, так как все значения отфильтруются и вернется пустая коллекция, что, естественно, неверно.

Вторая мутация:

class UserFilterAge

{

const AGE\_THRESHOLD = 18;

public function \_\_invoke(array $collection)

{

- return array\_filter(

+ array\_filter(

$collection,

function (array $item) {

return $item['age'] >= self::AGE\_THRESHOLD;

}

);

+ return null;

}

}

Опции

Из самых интересных опций, с которыми запускается Infection, можно выделить следующие:

--threads

Это количество потоков, работающих параллельно для запуска всего набора сгенерированных мутантов. Значительно ускоряет время выполнения. Но тут есть оговорка: если ваши тесты каким-то образом зависят друг от друга или используют базу данных, использование этой опции может привести к многочисленным упавшим тестам, что крайне негативно скажется на результирующих метриках. Поэтому просматривать лог хотябы на начальных стадиях внедрения все-таки стоит.

--show-mutations

Сразу выводит diff с неубитыми мутантами на консоль, что позволяет моментально анализировать результат и исправлять тест при его написании.

--mutators

Перечисление мутационных операторов, мутирующих код. Удобно, если вы например хотите проверить только PublicVisibility и ProtectedVisibility операторы.

./infection.phar --mutators=PublicVisibility,ProtectedVisibility

--min-msi и --min-covered-msi

Эти две опции полезны, если вы запускаете Infection как один из шагов процесса сборки вашего проекта на Continious Integration сервере.

--min-msi позволяет указать минимальное значение (в процентах) Mutation Score Indicator. Если указанное значение будет меньше фактического, то билд упадет. Данная опция заставляет при каждом билде покрывать большее количество строк кода.

--min-covered-msi соответственно позволяет указать минимальное значение Covered Code MSI. Данная опция при каждом билде заставляет писать более эффективные и надежные тесты.

Обе опции могут использоваться как по отдельности, так и вместе.

./infection.phar --min-msi=80 --min-covered-msi=95

Использование с Travis CI

before\_script:

- wget https://github.com/infection/infection/releases/download/0.5.0/infection.phar

- wget https://github.com/infection/infection/releases/download/0.5.0/infection.phar.pubkey

- chmod +x infection.phar

script:

- ./infection.phar --min-covered-msi=90 --threads=4

Каждый релиз (Phar архив) подписывается приватным openssl ключом, поэтому кроме самого архива вам необходимо скачивать и публичный ключ.

Как использовать мутационное тестирование?

Чем может быть полезно мутационное тестирование вам, как разработчику в ваших рабочих или персональных проектах? Как внедрить его в уже имеющийся проект?

Ежедневное использование для разработчика

Мутационное тестирование может быть полезно в ежедневной работе при написании новых тестов. Схема работы выглядит примерно так:

вы написали новый функционал, например тот же UserFilterAge из примера выше

этот код уже покрыт тестами

для проверки тестов, вы запускаете мутационное тестирование только для этого файла

./infection.phar --threads=4 --filter=UserFilterAge.php --show-mutations

Анализируете выжившие мутанты и пытаетесь добиться хорошего показателя Covered Code MSI — т.е. чтобы процент убитых мутантов из всех сгенерированных для покрытого тестами кода стремился к 100. Это позволит максимально эффективно писать тесты.

При использовании МТ вы заметите, что пишите более лаконичный код с большим количеством тестов. При этом будет использовано покрытие путей (branch coverage), когда все пути вашего кода протестированы, вместо обычного покрытия строк кода (line coverage).

Ежедневное использование в проекте

Мутационное тестирование может быть использовано на Continious Integration сервере. В зависимости от величины проекта, запускать его можно либо при каждом билде, либо реже, как вариант раз в сутки ночью. Тут главное анализировать полученный результат и постоянно улучшать качество тестов.

На мой взгляд, генерируя лишь только отчет, хороших показателей не добиться, поэтому лучше использовать опции --min-msi и/или --min-covered-msi.

Например, мутационный фреймворк Infection мутационно тестирует сам себя при каждом билде. И если показатели падают, билд тоже падает.

При постоянном использовании МТ, показатели MSI в проекте будут расти и вы сможете постепенно увеличивать значения опций --min-msi и --min-covered-msi.