УПРАВЛЯЕМАЯ ТЕСТАМИ РАЗРАБОТКА ИМИТАЦИОННЫХ МОДЕЛЕЙ

Д. В. Щетинин (Тверь)

Аннотация

В статье на простом примере рассматривается метод тестовой имитации (управляемой тестами разработки имитационных моделей). Данный подход может успешно применяться как для создания несложных имитационных моделей, например, в процессе обучения студентов, так и для разработки сложных промышленных моделей и имитационных программных комплексов.

Введение

В настоящее время растет интерес к т.н. *гибким методологиям* разработки программного обеспечения [1], [2], основной целью которых является снижение затрат на внесение изменений в программные системы на всех стадиях их жизненного цикла. Использование гибких методологий может обеспечить значительные преимущества при разработке имитационных моделей, поскольку необходимость постепенного, итеративного приближения к искомому решению (и, следовательно, обеспечения гибкости системы) здесь проявляется особенно сильно.

Метод *управляемой тестами разработки (Test-Driven Development, TDD)* [3] можно рассматривать как синтез нескольких принципов гибких методологий, основными из которых в данном случае являются: разработка тестов до кодирования **(«сначала тесты»)** и постоянная переработка кода системы (**«рефакторинг»**).

Имитационное моделирование и метод управляемой тестами разработки

TDD предлагает строить программные системы, последовательно формулируя и решая небольшие задачи, каждая из которых постепенно приближает разработчиков к поставленной цели. Решение очередной задачи выполняется за три шага.

1. **Создание теста.** Формализованная постановка задачи выполняется в виде *автоматического (программного) теста*. Тест состоит из *тела* и *условия* и считается выполненным успешно только в том случае, если в результате выполнения тела теста его условие принимает значение «истина».

2. **Реализация теста.** Разработчик должен как можно быстрее (наиболее простым способом) обеспечить успешное выполнение всех (в том числе созданных ранее) тестов.

3. **Рефакторинг.** Целью данного этапа является улучшение программного кода при сохранении внешнего поведения системы (последнее выражается как успешное выполнение всех тестов).

По мере развития системы в нее добавляются новые тесты. При этом уже существующие тесты сохраняются, при необходимости подвергаясь модификации.

Таким образом, хотя «традиционная» задача тестов – проверка правильности (адекватности) системы – сохраняется, основной целью их создания в TDD является формирование гибкой структуры системы на различных уровнях абстракции («проектирование в коде»).

Использование TDD в контексте имитационного моделирования имеет ряд особенностей, в первую очередь связанных с проблемой формализованного описания поведения динамических систем в достаточно компактной и понятной форме. Некоторые аспекты продемонстрированы на простом практическом примере ниже.

Пример разработки имитационной модели через тесты

Использование TDD при разработке имитационных моделей продемонстрируем на простом (учебном) примере. Допустим, нужно промоделировать работу системы массового обслуживания (для определенности «банка»). Детали будем уточнять по ходу разработки.

«Пустая» имитация

Начиная «с чистого листа», можно начать с реализации «пустой» имитации:

```
BankTests >> testEmptySimulation
   | simulation |
   simulation := BankSimulation new.
   simulation start.
   self assert: simulation log
           = (OrderedCollection with: 'started' with: 'finished')
```

Листинг 1. «Пустая» имитация

Здесь переменная ***start*** указывает на объект класса ***BankSimulation***, обеспечивающий управление имитацией (запуск через посылку сообщения ***start***) и доступ к ее результатам – протоколу (метод ***log***). В протоколе должны содержаться только записи о начале и окончании имитации. В условии теста проверяется совпадение содержимого протокола в ***simulation*** и *эталонного протокола*.

Для реализации данного теста необходимо создать класс ***BankSimulation*** с методами экземпляра ***start*** и ***log***. Реализация этих методов на данной итерации может быть максимально простой: ***log*** просто возвращает коллекцию, в которую в методе ***start*** добавляется нужные строки.

Обслуживание единственного клиента

Реализуем теперь имитацию чуть более сложного случая – обслуживание единственного клиента:

```
BankTests >> testOnlyClient
   | simulation client name arrTime |
   client := BankClient named: (name := 'client1') serviceTime: 3.
   (simulation := BankSimulation new)
   acceptClient: client at: ( arrTime := 1 );
      start.
   self assert: simulation log = ((OrderedCollection new)
            addString: 'Simulation started';
            addEvent: 'arrived' at: arrTime printString for: name;
            addEvent: 'left' at: (arrTime + client serviceTime) for: name;
            addString: 'Simulation finished';
            yourself)
```

Листинг 2. Обслуживание единственного клиента

Реализация требует введения понятия *клиента* (класс ***BankClient***), при создании которого указываются его имя и время обслуживания. Поскольку (для простоты) приход клиента планируется заранее (сообщение ***acceptClient:at:***), экземпляры ***BankSimulation*** должны хранить информацию о будущем *событии*. Для удобства можно ввести класс ***BankEvent***, экземпляры которого (пока) будут только хранить информацию о клиенте и времени его появления. Метод ***start*** обеспечивает *обработку события*, которая (на данном этапе) состоит в занесении соответствующей информации (**о приходе и уходе клиента сразу**) в протокол. Можно заметить, что объек-

ты, представляющие события, сейчас просто хранят информацию, используемую внутри ***BankSimulation*** при обработке событий – логично перенести обработку событий в класс ***BankEvent***. Но этого можно на данной итерации и не делать – код «сам потребует» такой модификации в дальнейшем.

Последовательное обслуживание двух клиентов

Следующий шаг – имитация последовательного обслуживания двух клиентов.

```
BankTests >> testClientAfterAnother
   | simulation arrivalTime1 arrivalTime2 |
   arrivalTime1 := 1.
   arrivalTime2 := arrivalTime1 + clients first serviceTime + 1.
   (simulation := BankSimulation new)
      acceptClient: clients first at: arrivalTime1;
      acceptClient: clients first at: arrivalTime2;
      start.
   self
      assert: simulation log = (self
                  logNoDelays: arrivalTime1 forClient: clients first;
                  logNoDelays: arrivalTime2 forClient: clients first;
                  log)
```

Листинг 3. Два клиента обслуживаются последовательно

Метод ***logNoDelays:forClient:*** обеспечивает занесение в эталонный протокол информации о приходе и уходе клиента.

Теперь в ***BankSimulation*** нужно хранить уже несколько событий (связанных с приходом клиентов), что требует введения *очереди* – сортированной по времени коллекции событий. В процессе имитации теперь необходимо обработать каждое событие из очереди. Сама обработка остается прежней – в протокол имитации заносится соответствующая информация.

Одновременное обслуживание двух клиентов

Создадим новый тест, который описывает посещение банка двумя клиентами, но второй приходит в тот момент, когда первый еще обслуживается. На данном этапе будем для простоты считать, что система обладает необходимым количеством обслуживающих устройств.

```
testClientWhileAnother
   | simulation |
   (simulation := BankSimulation new)
      acceptClients: clients at: arrivalTimes;
      start.
   self
      assert: simulation log = (self
               logNoDelays: arrivalTimes first forClient: clients first;
               logNoDelays: arrivalTimes second forClient: clients second;
               sort;
               log)
```

Листинг 4. Два клиента обслуживаются одновременно

Эталонный протокол теперь подвергается сортировке (метод ***sort***) по времени наступления событий, так как здесь второй клиент приходит до того, как систему покидает первый. Реализация такого поведения модели приводит к необходимости разделения событий *«приход клиента»* и *«уход клиента»* (после окончания обслуживания). Обработчик события *«приход клиента»* теперь должен не только занести информацию

в протокол имитации, но и запланировать событие *«уход клиента»*. Соответственно, обработку событий разного типа удобнее вынести в соответствующие классы (т.е. реализовать ее полиморфно).

Единственное обслуживающее устройство

Вернемся к вопросу о количестве обслуживающих устройств в системе. Рассмотрим случай, когда количество заявок (клиентов банка) в некоторый момент времени превышает количество свободных обслуживающих устройств (*банкоматов*). Введем в систему соответствующий тест:

```
BankTests >> testOneATM
   | simulation eventTimes2 |
   (eventTimes2 := OrderedCollection new)
      add: arrivalTimes second;
      add: arrivalTimes first + self clients first serviceTime;
      add: eventTimes2 second + self clients second serviceTime.
   (simulation := BankSimulation new)
      atmCount: 1;
      acceptClients: self clients at: arrivalTimes;
      start.
   self
      assert: simulation log = (self
            logNoDelays: arrivalTimes first forClient: self clients first;
            logTimes: eventTimes2 forClient: self clients second;
            sort;
            log)
```

Листинг 5. Обслуживание двух клиентов на одном банкомате

Теперь приход клиента может не совпадать с началом обслуживания, если все имеющиеся в наличии банкоматы заняты. (Количество банкоматов задается с помощью метода ***BankSimulation >> atmCount:***.) Следовательно, прежде всего, этот факт нужно отразить в эталонном протоколе. Для этого используется метод ***logTimes:forClient:***, который заносит в эталонный протокол записи о трех событиях (приход, начало обслуживания, уход), время которых задается в первом аргументе (коллекция ***eventTimes2***). Для реализации требуемого поведения необходимо ввести новый тип события – «*начало обслуживания*». Эти события можно трактовать как *условные*. Соответственно разработчик встает перед дилеммой: вводить поддержку условных событий «в чистом виде» или реализовать обнаружение события *«начало обслуживания»* с помощью событий, привязанных ко времени. В любом случае в систему вводится класс для данного типа событий (***BankServingEvent***), чья обработка заключается в занесении информации в протокол и планированию ухода клиента.

В системе необходимо сохранять информацию о банкоматах. Для этого удобно ввести соответствующий класс (***BankATM***), который отвечает за отслеживание состояния банкомата (занят / свободен) и (при необходимости) вычисления времени его освобождения.

После внесения изменений, обеспечивающих выполнение данного теста, необходимо будет обеспечить выполнение старых тестов. Наиболее простой способ сделать это – задать необходимое количество банкоматов в системе как значение «по умолчанию». Интересным альтернативным решением является создание обрабатывающего устройства неограниченной емкости. Программная реализация такого класса оказывается довольно простой.

В следующей таблице обобщены результаты проведенных итераций:

Тест	Изменения в системе
«Пустая» имитация	Управление имитацией (класс ***`BankSimulation`***); протокол событий
Обслуживание единственного клиента	Клиент (***`BankClient`***); обработка событий (***`BankEvent`***)
Последовательное обслуживание двух клиентов	Очередь событий, цикл обработки событий
Одновременное обслуживание двух клиентов	Разделение событий по типам (*«приход клиента»* и *«уход клиента»*)
Обслуживание двух клиентов на одном банкомате	Событие *«начало обслуживания»*, банкоматы (класс ***`BankATM`***), отслеживание состояния банкоматов

Естественно, рассмотренные шаги не исчерпывают возможности развития данной имитационной модели. Тесты позволяют сохранить контроль над ходом разработки и обеспечивают эволюцию системы от моделирования простейших ситуаций до сложных имитаторов с высокой степенью реалистичности.

Заключение

Метод тестовой имитации может применяться как при реализации (относительно) простых учебных примеров при обучении студентов, так и для реальных, сложных имитационных моделей и программных комплексов [4], [5].

В первом случае он обеспечивает успешное освоение принципов построения имитационных моделей и внутренних механизмов их реализации за счет наглядной демонстрации требований к имитационным моделям. В случае промышленной разработки сложных имитационных моделей и построения программных комплексов на их основе метод управляемой тестами разработки обеспечивает необходимую степень гибкости программного кода, позволяя без лишних усилий организовать итеративное построение адекватных моделей.

Литература

1. Manifesto for Agile Software Development. http://agilemanifesto.org/.
2. **Бек К.** Экстремальное программирование. – СПб.: Питер, 2002.
3. **Beck K**. Test-Driven Development: By Example. – Addison-Wesley, 2003.
4. **Местецкий Л. М., Щетинин Д. В**. Имитационная модель наземного движения воздушных судов в аэропорту//Сборник докладов 1-й Всероссийской конференции ИММОД-2003. – Т. II. – СПб., 2003.
5. **Щетинин Д.В.** Управляемая тестами разработка в применении к имитационному моделированию//Сложные системы: обработка информации, моделирование и оптимизация: Сборник научных трудов. – Тверь, Тверской Государственный Университет, 2005.