ТЕСТИРОВАНИЕ ИМИТАЦИОННЫХ МОДЕЛЕЙ В ANYLOGIC С ПОМОЩЬЮ JUNIT

В.Е. Черненко (Ульяновск)

Введение

В коммерческих проектах время на создание имитационной модели (ИМ) сильно ограничено, а тестирование зачастую проводится только перед передачей ИМ заказчику. Такой подход кажется разумным, так как по ходу проекта выявляются особенности решаемой задачи, о которых не было известно в начале проекта, и требования к модели постоянно меняются. Если в таких условиях создавать тесты параллельно с разработкой, то они быстро потеряют свою актуальность. Возможно, именно этот факт привел к тому, что в популярных средах разработки ИМ нет инструментария для тестирования.

С другой стороны, при поддержке и развитии ИМ в нее нередко приходится вносить изменения, которые могут нарушить проверенную ранее функциональность. Это особенно актуально для сложных моделей, которые состоят из десятков взаимодействующих логических элементов (например, единиц оборудования различного типа).

Трудозатраты на поддержку корректности сложных ИМ в длительных проектах становятся значительными. Тестирование, выполняемое параллельно с разработкой, является основным способом контроля и снижения этих затрат. Автоматизация тестирования способна еще более снизить затраты, делая продолжительные проекты с ИМ экономически выгодными для исполнителя.

О том, как реализовать тестирование в коммерческой разработке, написано немало книг [2], и в области ИМ эта задача тоже не нова. Так, например, в публикациях конференции Wintersim 1995 года [3] затрагиваются аспекты автоматизированного тестирования ИМ и предлагаются некоторые решения.

К решению задачи тестирования ИМ в среде AnyLogic [4] уже прибегали, например, в публикации [5], но удовлетворительного решения для автоматизации тестирования предложено не было. Данная статья предлагает конфигурацию ИМ для организации ее тестирования в AnyLogic.

Структура проекта в AnyLogic

Предлагаемая структура проекта содержит четыре модуля, каждый из которых реализован в виде отдельного файла модели AnyLogic:

1. Модель данных (DataModel) – набор классов, описывающих предметную область. Корневой класс модели данных можно назвать «Scenario» – сценарий, содержащий всю необходимую информацию о предметной области. Также этот модуль может содержать классы, отвечающие за считывание / сохранение модели данных из файла / базы данных, а также за проверку корректности этих данных.

2. Модель (ModelUnderTest) – имитационная модель, которую необходимо протестировать. Эта модель не должна содержать простой эксперимент (Simulation) для запуска. По возможности она также не должна содержать анимацию, связанную со сбором данных для визуализации и статистики. Иными словами, важно как можно более строго соблюдать принцип отделения логики модели от ее внешнего представления. Эта имитационная модель будет запускаться с помощью JUnit-тестов без анимации и не будет содержать логику анимации.

3. Модель с анимацией (AnimationModel) – имитационная модель, содержащая анимацию и статистику. Этой моделью можно пользоваться для запуска конечного приложения и для запуска анимации при анализе корректности работы модели при тестировании.

4. Среда тестирования (UnitTests) – проект, содержащий все необходимое для организации процесса тестирования. Такая среда тестирования может использоваться как шаблон и для других проектов. Например, если создается тест для транспортной модели, к названию среды тестирования можно добавить префикс «TransportModel» (TransportModelUnitTests). Этот проект содержит классы с JUnit-тестами, а также ссылки на Jar-файлы (a.jar и b.jar), необходимые для запуска тестов.

Структура проекта в списке проектов AnyLogic имеет следующий вид (рис. 1).

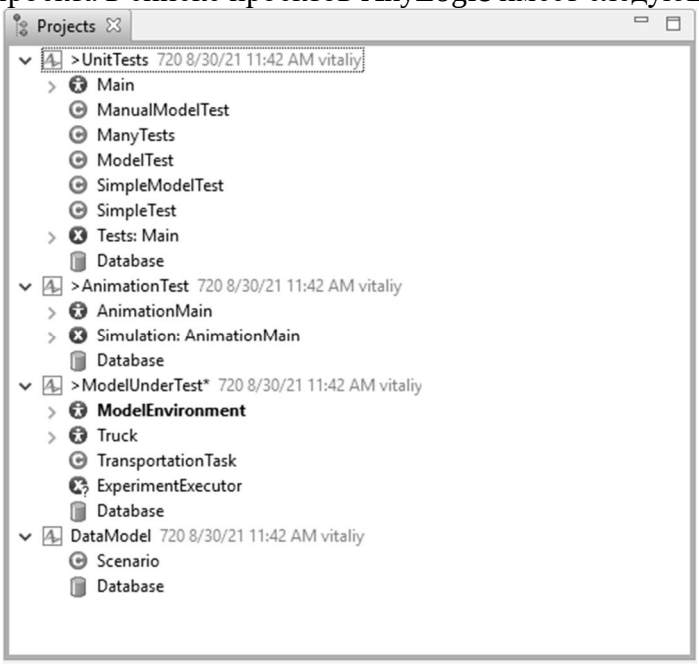

Рис. 1. Структура проекта в списке проектов AnyLogic

Технические аспекты реализация тестирования

Класс ModelEnvironment является корневым классом агента тестируемой модели. Для реализации предлагаемой конфигурации в этом классе должны быть определены:

· переменная *scenario* – для доступа ко всем значениям сценария из модели;

· динамическое событие *AuxDynamicEvent* – для возможности выполнять код теста в заданные момент времени;

· переменные *beforeSimulationStart* и *afterSimulationFinished* с типом Consumer<ModelEnvironment>, с помощью которых выполняется код теста при начале и окончании моделирования.

Абстрактный класс ModelTest является базовым классом для всех классов тестов. Класс предоставляет следующую функциональность:

· метод *run* позволяет запустить модель в визуальном режиме (как при нажатии кнопки «Запустить» в интерфейсе AnyLogic);

· метод *executeTest* позволяет запустить модель без анимации; этот метод предназначен для вызова из JUnit-тестов;

· метод *doAtTime* – позволяет выполнить код в указанные моменты времени.

В процессе разработки и использования модели она может запускаться тремя различными способами:

1. Запуск ИМ из интерфейса AnyLogic;

2. Запуск визуального теста из среды тестирования; для этого создается экземпляр класса-наследника *ModelTest* и вызывается его метод run;

3. Запуск JUnit без анимации. Для этого в коде JUnit-теста создается экземпляр класса-наследника *ModelTest* и вызывается его метод методом *executeTest*.

Пример

В качестве примера имитационной модели предлагается модель транспортировки нефтепродуктов из резервуара A (sourceTank) в резервуар B (destTank) с помощью грузовиков. Модель в AnyLogic может иметь следующий вид (рис. 2).

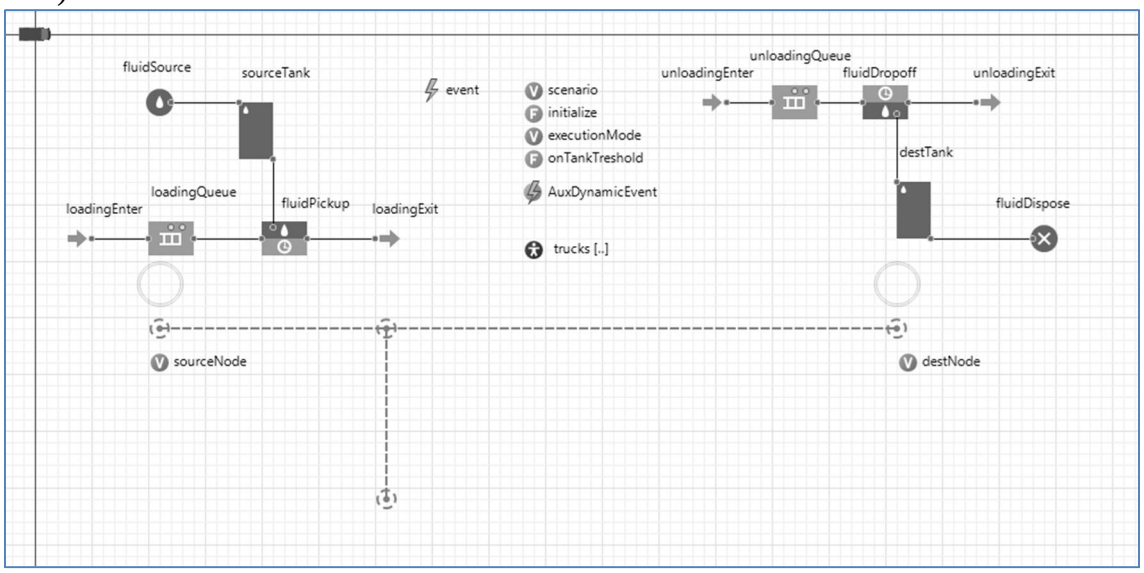

Рис. 2. Внешний вид агента верхнего уровня тестируемой ИМ

В резервуар A поступает фиксированное количество нефтепродукта. Время между поступлениями нефтепродукта является случайной величиной, распределенной по закону треугольного распределения. Грузовики базируются в гараже, их количество и вместимость являются параметрами и задаются в модели данных. Транспортировка нефтепродукта начинается в момент заполнения резервуара на 50% при наличии свободных грузовиков. В этот момент планируется по три рейса для каждого свободного грузовика.

В целом, описание программной реализации ИМ для автоматизированного тестирования не является целью данной статьи и приведено исключительно для демонстрации предлагаемой структуры всего проекта в AnyLogic и возможных типов тестов.

Стандартный подход к реализации такой модели в AnyLogic предполагает создание схемы процесса транспортировки с помощью библиотеки моделирования процессов (Process modeling library), которая может выглядеть так, как показано на рис. 2.

Такая реализация не требует выделения задания на транспортировку в отдельную сущность. Напротив, логика выполнения рейса грузовиком «размазывается» между двумя модулями – основным классом модели Main и классом агента-грузовика Truck. Таким образом, тестирование агента-грузовика становится невозможным без использования класса Main.

Альтернативным вариантом реализации логики модели, обеспечивающим большую модульность, является выделение сущности «задание на транспортировку» в отдельный класс и реализация логики выполнения таких заданий в классе Truck. Этот вариант позволяет, с одной стороны, легко отделить логику формирования заданий на транспортировку от логики их исполнения, а с другой – отделить класс агента Truck от «среды обитания» агентов, реализованной в классе Main. Это, например, позволит

использовать класс Truck в моделях с другой конфигурацией транспортной сети. Именно такая декомпозиция обеспечивает то, что каждый класс «занимается свои делом»: класс TruckTask содержит информацию о задании на транспортировку, класс Truck выполняет задания, а класс Main предоставляет необходимый контекст, задающий среду функционирования грузовиков.

Такая модульность будет полезной и при развитии модели. Например, добавление моделирования заправки и ремонтов грузовиков с точки зрения архитектуры модели сводится к добавлению двух классов заданий – «задание на заправку» и «задание на выполнение ремонта». Кроме того, может оказаться, что класс «задание на выполнение ремонта» применим не только для грузовиков, но и для оборудования другого типа.

Предложенная архитектура позволяет реализовать тест, в котором:

1. одно задание на транспортировку создается в начале теста и передается грузовику на исполнение;
2. в момент времени, указанный в задании, грузовик выполняет задание;
3. по окончанию теста проверяются контрольные значения, например:
    a. количество выполненных грузовиком рейсов;
    b. количество нефтепродукта, находящегося в dest.

Кроме того, появляется возможность проверки самого алгоритма формирования таких заданий в отрыве от их исполнения.

Тестирование

Для демонстрации работы предлагаемого подхода рассмотрим три теста:

1. тест выполнения единичного задания;
2. запуск сценария;
3. множественные тесты в одном классе.

Для выполнения единичного задания в определенный момент времени работы ИМ необходимо отключить логику диспетчеризации, то есть автоматическое создание заданий. Для этого в модели предусмотрено поле *executionMode*, которое нужно установить в состояние *Mode.TEST*. Для назначения задания грузовику можно воспользоваться функцией *doAtTime* класса *ModelTest*. Код назначения такого здания приведен на листинге 1:

Листинг 1

```
doAtTime( 10, () -> {
    truck = modelEnvironment.trucks.get(0);                                  // первый грузовик
    truck.startTask( new TransportationTask( modelEnvironment.sourceNode,    // позиция загрузки
                                             modelEnvironment.loadingEnter,  // блок Enter загрузки
                                             modelEnvironment.destNode,      // позиция разгрузки
                                             modelEnvironment.unloadingEnter, // блок Enter разгрузки
                                             1                               // количество рейсов
                                             ));
});
```

В этом примере в момент времени 10 минут первый грузовик должен начать выполнять рейс из точки погрузки в точку разгрузки. Код самого теста пояснен комментариями. Стоит отметить, что сравнение нужно производить с переменными создаваемого в тесте экземпляра класса *test* (выделен на листинге 2).

Листинг 2

```
@Test
public void runTest(){
    SingleRunTest test = new SingleRunTest();
    test.executeTest();
    // Проверка корректности даты начала моделирования
    assertEquals("startScenarioDate", scenario.getBeginDate(), test.startScenarioDate);
    // Проверка корректности начальной инициализации бункера
    assertEquals( "sourceStockAtTime0", scenario.getSourceBunkerInitStock(), test.sourceStockAtTime0, 0);
    // Принимая во внимание что в момент 20 минут грузовик должен быть в пути на разгрузку
    // Проверка состояний sourceTank и destTank
    assertEquals( "Source tank at 20", 100, test.sourceStockAtTime20, 1 );
    assertEquals( "Dest tank at 20", 0, test.destStockAtTime20, 1 );
    // Проверка факта разгрузки грузовика в destTank в момент 25 минут
    assertEquals( "Разгурзился", 100, test.destAmountPassedInAt25, 1 );
    // Проверка нахождения грузовика в гараже в 30 минут модельного времени
    assertEquals( "Idle", true, test.truckIsIdleAt30 );
}
```

Заполнение переменных для проверки выполняется в методе beforeSimulationStart, который показан на листинге 3:

Листинг 3

```
doAtTime( 20, () -> {
    sourceStockAtTime20 = modelEnvironment.sourceTank.amount();
    destStockAtTime20 = modelEnvironment.destTank.amount();
});
```

Приведем пример теста, позволяющего убедиться, что общие показатели работы модели на специально подготовленным наборе данных соответствуют ожидаемым. При этом проверка показателей выполняется в момент окончания моделирования. Таким образом, тест сводится к подготовке сценария (набора данных) и сравнению агрегированных показателей. Пример такого теста приведен на листинге 4:

Листинг 4

```
@Test
public void runTest(){
    SimpleModelTest test = new SimpleModelTest();
    test.executeTest();

    assertEquals(   "Материальынй баланс",
                    test.modelEnvironment.sourceTank.amountPassedOut(),
                    test.modelEnvironment.destTank.amountPassedIn(),
                    1 );
}
```

В примере проверяется, что весь объем, вывезенный из резервуара А, равен всему объему, привезенному в резервуар В. Этот тест в ходе подготовки статьи непреднамеренно не выполнился. Автор получил следующий результат:

Ожидаемое значение = 3599.999999999959, смоделированное значение = 7200.000000000038.

Быстрая проверка показала, что было некорректно заполнено значение поля fluidDropoff – оно осталось по умолчанию равным 100, когда было добавлено на холст. Корректное значение – scenario.getTruckCapacity().

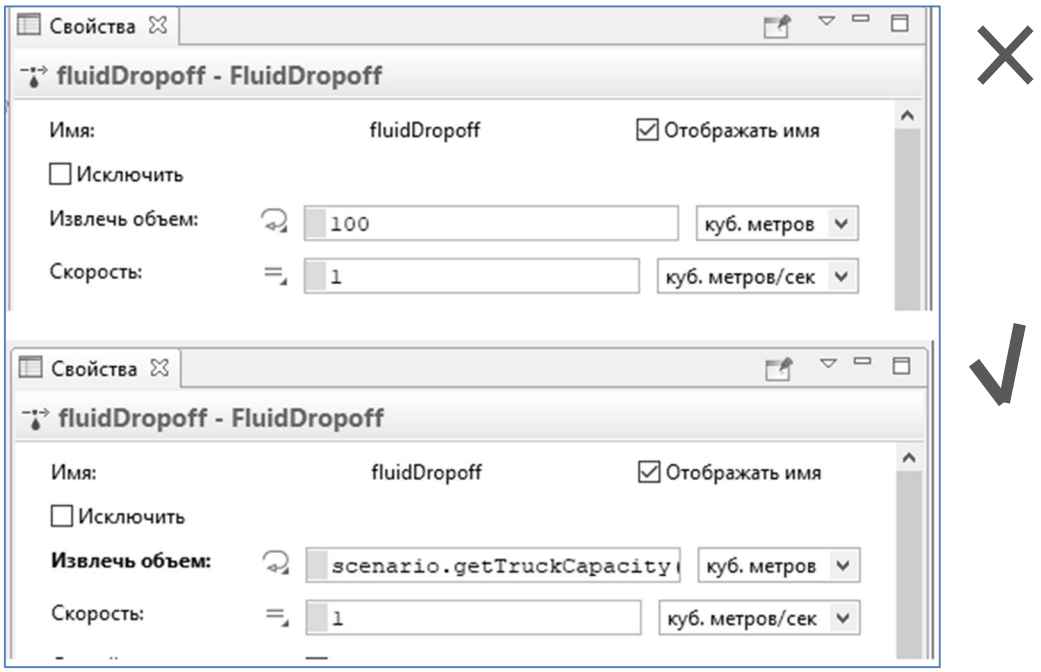

Рис. 3. Свойства объекта fluidDropoff с ошибкой и после ее исправления

В одном классе можно описать несколько тестов. Такой пример приведен на листинге 5.

Листинг 5

```
@Test
public void runTest1(){
    ManyTests test = new ManyTests();
    test.scenario.setTruckCount( 1 );
    test.executeTest();
    assertEquals( "Truck count", 1, test.modelEnvironment.trucks.size() );
}

@Test
public void runTest2(){
    ManyTests test = new ManyTests();
    test.scenario.setTruckCount(2);
    test.executeTest();
    assertEquals( "Truck count", 2, test.modelEnvironment.trucks.size() );
}

@Test
public void runTest3(){
    ManyTests test = new ManyTests();
    test.scenario.setTruckCount(3);
    test.executeTest();
    assertEquals( "Truck count", 3, test.modelEnvironment.trucks.size() );
}
```

В этом примере один сценарий запускается три раза с различным количеством грузовиков. Количество грузовиков задается с помощью метода *setTruckCount* перед вызовом метода *executeTest*, запускающего выполнение сценария.

Выводы

Предложенная структура проекта ИМ позволяет разработчику использовать для тестирования в AnyLogic технологию Junit, признанную стандартом в мире разработки программного обеспечения. При этом обеспечивается возможность запуска созданных тестов с визуализацией, что позволяет лучше понимать логику исполнения модели. Тестирование является основным способом контроля и снижения затрат на поддержку и развитие имитационных моделей.

Опыт применения предложенного подхода показывает, что создание функциональных тестов параллельно с разработкой имитационной модели побуждает разработчика к повышению модульности модели. Таким образом, сложность переносится на композицию объектов, позволяя избегать создания объектов-«монстров» с чрезмерно сложной логикой.

Литература

1. https://github.com/chernenkove/anylogic-testing-environment.
2. **Гленфорд Майерс, Том Баджетт, Кори Сандлер.** Искусство тестирования программ, 3-е издание, М.: «Диалектика», 2012. 272 с. ISBN 978-5-8459-1796-6.
3. **Osman Balchi.** Principles and techniques of simulation validation, verification, and testing // Proceedings of the 1995 Winter Simulation Conference, P. 147-154.
4. https://www.anylogic.ru/.
5. **James T. Sawyer, David M. Brann.** How to test your models more effectively: applying agile and automated techniques to simulation testing // Proceedings of the 2009 Winter Simulation Conference, P. 968-978.