Nanomaterials ontology model

Irene Leonidovna Artem'eva, Full Professor, Dr. Sc. (Engineering)
Head of Applied Mathematics, Mechanics, Management and Computer Software Department,
Far Eastern Federal University

Natalya Valentinovna Reshtanenko, Cand. Sc. (Engineering)
Associate Professor of Applied Mathematics, Mechanics, Management and Computer Software Department,
Far Eastern Federal University

Using intelligent modeling systems for explaining the results of calculations in terms of a particular domain provides more options by contrast with program systems of other classes. A formally described domain ontology that defines the unambiguous interpretation of the terms makes it possible to develop intelligent modeling systems combining knowledge and data of different sections of this domain and program systems for solving application issues. Nanoscience is of a cross-disciplinary nature which implies the necessity to use knowledge of various disciplines, i.e. there are ontologies of other domains in this ontology. The paper proposes a mathematical ontology model of the nanomaterials domain with complicated structures and terms from ontologies of organic chemistry and physical chemistry.

Keywords:Mathematical modeling; ontology model; development of knowledge-based systems.

УДК 004.053, 004.054

АКТУАЛЬНЫЕ ПРОБЛЕМЫ ОРГАНИЗАЦИИ МОДУЛЬНОГО ТЕСТИРОВАНИЯ КЛАССОВ ПРОГРАММНОГО КОДА

***Дмитрий Олегович Кожевников**,аспирант*
кафедра информационных технологий
Тел.: +7 953 5942082, email:d.o.kozhevnikov@gmail.com
Сибирский государственный технологический университет
http://www.kit-sibstu.ru/

Данная работа посвящена описанию и анализу некоторых проблем разработки модульных тестов, которые связаны с недостатками тестируемого кода. Различая достижение пригодность к модульному тестированию и получение кратких и понятных модульных тестов, материал работы концентрируется на описании проектных решений, которые приводят к появлению громоздких и сложных для чтения модульных тестов. Анализируются симптомы и признаки таких решений в тестируемом коде программы.

Ключевые слова: модульное тестирование, рефакторинг, слабая связность, внедрение зависимостей, тестовый двойник, конструктор, контейнер инверсии управления.

Введение

Данная работа основана на моделях, методах и алгоритмах, описанных в книгах и статьях Джерарда Мессароша, Роя Ошерова, Мартина Фаулера, Кента Бека, Марка Симана и Роберта Мартина. Целью данной работы является обращение внимания на актуальность конкретизации, формализации и каталогизации моделей и методов проектирования и рефакторинга [5] программного кода, относительного которого разрабатываются и применяются модульные тесты. Работа призвана обозначить предметную область, проблематику и основные направления дальнейшего исследования.


Д.О. Кожевников

Модульное тестирование [1] классов, нагруженных реализацией критически важных правил и ограничений предметной области, является неотъемлемой частью процесса разработки корпоративных приложений. В современных мето-

дологиях разработки ему отводится очень существенная роль на всех этапах жизненного цикла программного обеспечения. Создание и поддержка в проекте обширных наборов модульных тестов является нормой для проектов различной направленности и размера.

Модульные тесты представляют собой компактные методы, проверяющие правильность работы отдельных аспектов классов приложения изолировано и максимально просто. Очень важно, чтобы тесты проверяли значительную часть аспектов работы класса, то есть, чтобы достигалась высокая степень покрытия [1]. Широкое применение модульного тестирования предъявляет ряд концептуальных требований к устройству тестируемых классов и характеру связей между классами. Основными из них является достижение слабой связности [7] между классами и программирование на основе интерфейсов [3]. Данные концепции полезны и должны применяться вне зависимости от того, будут ли разрабатываться модульные тесты вообще [3], однако именно они делают код программы пригодным для модульного тестирования в принципе.

Достижение слабой связности классов и модулей программы всегда является одной из основных задач при проектировании. В рамках объектно-ориентированной парадигмы это означает, что классы как минимум должны проектироваться в соответствии с принципами объектно-ориентированного программирования, такими как инкапсуляция, полиморфизм и наследование. Больше выгод можно получить, уместно применяя принципы SOLID [7] и паттерны проектирования [8]. Однако написание и поддержка большого количества коротких и полезных модульных тестов остаётся нетривиальной задачей проектировщиков и разработчиков [9]. Особенно это касается покрытия сложных правил предметной области корпоративных приложений [4].

Модульные тесты должны оперативно извещать разработчиков об ошибках и несогласованных изменениях, снижая риск как при рефакторинге, так и в случае приращения новых функций. В методологии разработки через тестирование [6] модульные тестирование позволяет развивать программу итеративно, формируя устройство классов и модулей, в то время как все изменения в коде направляют модульные тесты. Но помимо всего прочего применение модульного тестирования также явно обнажает проблемы сильной связности и может указать на недостатки в устройстве классов. При этом возможны две ситуации: модульные тесты невозможно написать в принципе и модульные тесты для существующих классов получается громоздким и нечитаемым.

Разноплановость влияния модульного тестирования на устройство классов, архитектуру программы в целом и сам процесс разработки обусловлена глубоким и всесторонним проникновением логики тестов в логику каждого тестируемого класса. Таким образом, если все классы должны быть покрыты модульными тестами, все классы должны быть спроектированы слабосвязанными.

Если модульное тестирование классов невозможно из-за сильной связности, то это может послужить поводом для проведения рефакторинга. Это явная проблема, которую нельзя игнорировать иначе, как отказаться от модульного тестирования вовсе. Однако проблема больших и трудных для понимания тестов гораздо тоньше, так как её проще не замечать. В этом случае недостатки формально слабосвязных классов будут накапливаться, постепенно затрудняя дальнейшую работу с программой.

Таким образом, организация процесса разработки с применением модульного тестирования и стремление получить все преимущества стимулирует разработчиков серьёзнее относиться к проектированию классов и модулей программы, осознавать принципы слабой связности как основу проектирования. Поэтому разработка новых моделей и методов проектированию для получения кратких и полезных модульных тестов для классов программы является актуальной задачей для специалистов отрасли разработки программного обеспечения.

Паттерн Arrange-Act-Assert. Качественные модульные тесты отличаются простотой и доходчивостью, они автоматизированы, выполняются быстро и доступны ка-

ждому разработчику в команде[1]. Тесты не только помогают рано диагностировать различные виды программных ошибок, но и позволяют безопасно проводить рефакторинг на регулярной основе, легко автоматизируются различными инструментами выполнения, могут быть интегрированы в сценарии автоматической сборки продукта, представляют собой всегда актуальную документацию для разработчиков, несут информацию о том, как правильно использовать программный интерфейс тестируемых классов. Все эти преимущества не появляются у разработчика автоматически с написанием модульных тестов.

Даже если тестовый метод укладывается в определение модульного теста[1], его наличие в проекте может стать проблемой. Некачественные модульные тесты – сложные для чтения и поддержки с точностью до наоборот существенно снижают возможности разработчиков по развитию программы. Это происходит потому, что тест трудно использовать по назначению. Если громоздкий, трудно читаемый тест сообщает об ошибке, трудно понять, что именно привело к ней при добавлении новой функциональности, где рефакторинг привёл к рассогласованию работы, каким образом нужно правильно использовать класс в данной ситуации.

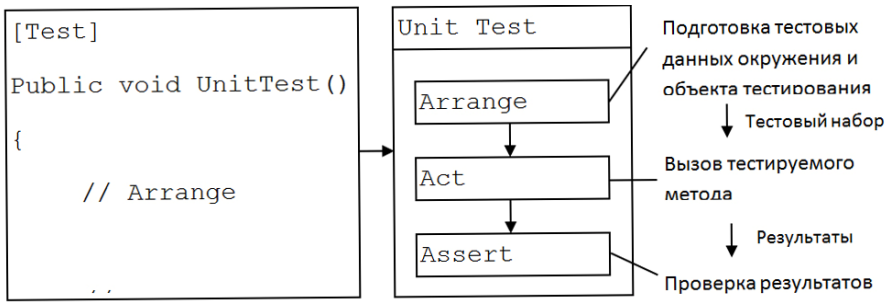

Рис. Структура модульного теста. Паттерн AAA.

Рассматривая проблемы написания простых модульных тестов и подходы к их решению необходимо обратиться к наработанным методам организации модульного тестирования. Существуют каталоги моделей (паттернов), предназначенных для получения классов, удобных для модульного тестирования, а также для получения простых и полезных тестов и тестовых наборов[2, 6, 1].

Структура типичного модульного теста включает три секции, описываемые паттерном AAA(Arrange-Act-Assert)[1]. Схема паттерна представлена на рисунке. В секции Arrange находится подготовка тестовых данных и конфигурация объекта тестирования, формируется тестовый набор, в секции Act выполняется тестируемое действие, в секции Assert находятся проверки ожидаемых результатов действия. Код в каждой секции AAA имеет различную специфику, и трудности с чтением кода секций имеют различный характер.

Код к секции Arrange должен следовать правилу: конфигурируй только необходимые данные в самом простом виде. Во многом Arrange является ключевой секцией теста и в общем случае наиболее ёмкой и подверженной излишнему разрастанию и усложнению. В Arrange задаются тестовые данные, определяются зависимости[3] и их поведение, создаётся и конфигурируется объект тестируемого класса. При этом объект тестируемого класса должен быть только один, а все его зависимости должны быть подменены тестовыми двойниками[2]. Если подмена невозможна, это указывает на сильную связность класса и его зависимостей и требует рефакторинга.

Код в секции Act должен следовать правилу: тестируй только один вызов метода. В Act находится вызов метода, аспект работы которого проверяется модульным тестом. Не допускаются вызовы нескольких методов. Это гарантирует, что никакие другие вызовы не повлияют на внутреннее состояние объекта и возвращаемые результаты.

Код в секции Assert должен следовать правилу: проверяй только один аспект работы тестируемого класса. Другими словами проверка должна быть только одна. Ограничение числа проверок позволяет быстрее локализовать и диагностировать ошибку при запуске тестов. Каждый тест точно определяет уязвимый аспект работы. Несколько

проверок всегда вносят путаницу в причины появления нарушения работы теста[6] и лишают разработчика возможности оперативно, глядя лишь на имя теста, понять, какой аспект работает неправильно.

В целом существуют довольно чёткие устоявшиеся рекомендации относительно содержания секций Act и Assert. Секция Act должна состоять из одного вызова метода тестируемого класса. Секция Assert должна содержать одну проверку одного аспекта работы класса – это и есть цель данного модульного теста. Данные рекомендации нетрудно соблюдать, если распределять тестирование различных аспектов по отдельным тестам. Секция Arrange в общем случае должна быть как можно проще и доступнее для понимания. Фактически вся логика теста должна полностью отражаться в его имени [1], что само по себе уже является существенным ограничением.

Секция Arrange. Однако если организация Act и Assert по большей части определяется следованием или игнорированием простых рекомендаций, то код в Arrange сильно зависит от дизайна класса и его зависимостей. Это часто делает Arrange наиболее сложной секцией, как для написания, так и для чтения, понимания впоследствии. В секции Arrange задаются условия, в которых будет выполняться тестирование объекта. Непонимание условий, в которых запускается тест или, что то же самое, невозможность написания понятного имени теста существенно его снижает полезность. У проблемы есть три основных аспекта:

1) Из имени теста непонятно, у какого объекта что именно проверяется и при каких условиях. Информативные имена позволяют работать с тестами, даже не заглядывая в их код, используя только инструменты автоматического запуска, что существенно экономит время. Неинформативные имена затягивают сеансы рефакторинга просмотром кода тестов.

2) Затруднено чтение теста. В случае если тест не прошёл, необходимо оперативно определить причину ошибки. Первое на что разработчик обращает внимание, это имя теста и вывод ошибки. Если этого недостаточно, чтобы понять что случилось, разработчик обращается к коду теста. Код должен ясно и недвусмысленно говорить о том, какие входные данные привели к ошибке.

3) Затруднён рефакторинг теста. Каким бы подходом к рефакторингу не пользовался разработчик – от кода к тесту или от теста к коду – сложный код Arrange будет серьёзным препятствием к внесению изменений.

Существует ряд разработанных методов упрощения кода секций Arrange, относящихся к устройству самого модульного теста:

1) Не конфигурировать больше деталей, чем необходимо для прохождения теста. Любые посторонние данные, которые не влияют на конечный исход выполнения теста, только затрудняют его чтение и снижают полезность. Даже если в рабочем коде класс никогда не используется без этих данных, это не повод засорять ими модульный тест.

2) Не конфигурировать больший объём тестовых данных, чем необходимо для проверки целевого аспекта работы тестируемого класса. Если для проверки можно ограничиться небольшим набором тестовых данных, всегда следует предпочесть именно его. Большой объём тестовых данных говорит о том, что тест проверяет более одного аспекта работы. Часто такие тесты также содержат избыточные проверки в секции Assert.

3) Конфигурация Arrange должна целиком укладываться в тест, без обращения к сторонним методам, классам и тем более внешним ресурсам. Сторонние методы, специальные методы тестовых классов [2], внешние ресурсы конфигурации это места, в которые можно перенести часть логики Arrange, формально упростив код секции. Однако проблема остаётся. Более того, теперь быстро разобраться в причинах ошибки, назначении теста или тестируемого класса стало ещё сложнее.

4) Применение тестовых двойников [2] для подмены зависимостей тестируемого класса с использованием каркаса изоляции [1]. Можно создавать объекты двойники

вручную с помощью специально написанных классов, которые наследуют абстракциям [3] зависимостей. Однако каркас изоляции позволит компактно сконфигурировать объект двойник прямо в коде теста и исключить необходимость просматривать другие классы при чтении теста.

Несомненно, перегруженные секции Arrange являются симптомом (запахом [5, 6]) неудачного устройства модульного теста. Такой тест теряет свою полезность из-за трудностей с чтением, сопровождением и рефакторингом. Со временем он начнёт приносить больше вреда, чем пользы. Однако, как уже было сказано ранее, проблемы секции Arrange зачастую невозможно решить какими-либо ухищрениями при написании кода теста. Важно отметить, что код Arrange говорит о внутреннем устройстве класса и специфике его работы гораздо больше, чем остальные секции. Поэтому перегруженные секции Arrange как симптом могут также указывать на недостатки в коде тестируемого класса.

Если попытаться классифицировать распространённые недостатки, которыми страдают секции Arrange модульных тестов, получится следующий список (важно отметить, что данная классификация включает в себя симптомы, которые появляются в коде тестов вопреки осознанному стремлению разработчиков следовать основополагающим методам организации модульного тестирования, в том числе описанным выше):

1) Большое количество тестовых двойников, которые подменяют зависимости тестируемого класса. Данный симптом может свидетельствовать о нескольких негативных явлениях в устройстве тестируемого класса:

   a. Класс имеет избыточное количество зависимостей, то есть нарушает базовый принцип единственности ответственности [7].

   b. Класс получает приемлемое количество зависимостей через конструктор, однако в каждом тестируемом вызове пользуется только частью из них. Возможно, выбран неверная модель (паттерн) внедрения зависимостей [3].

2) Сложная, составная конфигурация тестовых данных и\или тестовых двойников зависимостей. Тестируемый класс требует от своих зависимостей сложного поведения, а от данных сложного строения и детальной подготовки, то есть обращается к большому количеству полей и методов. Этот симптом также может указывать на разные негативные явления:

   a. Неверное распределение ответственности между тестируемым классом и его зависимостями. Тестируемый класс слишком сильно вовлечён в манипуляции с данными своих зависимостей. Как правило, в этом случае нарушается принцип единственности ответственности и страдает инкапсуляция в классах зависимостях или тестовых данных.

   b. Многоходовый, сложный алгоритм атомарно выполняется во время вызова в Act, от тестовых двойников зависимостей требуется в каком-то виде поддерживать существенную часть функциональности подменяемых зависимостей. Количество подменяемых зависимостей также возрастает.

3) Применение нескольких тестовых двойников типа мок [1]. Моком называется тестовый двойник способный запоминать аргументы вызовов своих методов, количество вызовов, часто также способный самостоятельно проводить проверки. Как правило, каркасы изоляции позволяют наделять моки дополнительным настраиваемым поведением вплоть до полной имитации работы класса с подменой одного небольшого аспекта. Мок даже может выступать как тестируемый объект. Наличие нескольких подобных объектов в модульном тесте чрезмерно усложняет понимание его работы. Во многом проблематика использования моков выходит за границы секции Arrange и распространяется на весь тест.

4) Явно определённый защищённый конструктор по умолчанию у класса зависимости. Данный симптом указывает на пренебрежение концепцией программирования на основе интерфейсов. Тестируемый класс принимает черезконструктор объект конкретного класса, а не интерфейса или абстрактного класса. В свою очередь класс зависимости тоже имеет свои зависимости, принимаемые через конструктор. Не каждый

каркас изоляции сможет [1] сгенерировать тестовый двойник для такой зависимости, так как двойники, как правило, являются объектами генерируемых классов наследников от классов подменяемых зависимостей.

Необходимо понять, что появление описанных проблем тесно связано не только с устройством самого тестируемого класса, но и с задачей управления и внедрения зависимостей. Концепция внедрения зависимостей является необходимым условием организации модульного тестирования, но правильное применение моделей и методов внедрения зависимостей также остаётся нетривиальной задачей для разработчиков, особенно когда они не пишут модульных тестов. В ином случае тесты могут направлять эволюцию их решений в правильное русло. Основные идеи на этом направлении изложены в работах из списка использованных источников.

Секция Act. Не только код секции Arrange способен многое прояснить в устройстве тестируемого класса и содержащего его модуля. Если быть точным, то можно сказать, что на стыке секций Arrange и Act может залегать непростая проблема. Как уже было сказано, хорошей практикой является вызов одного метода тестируемого класса в секции Arrange. Однако, как правило, тест вызывает ещё как минимум один метод тестируемого класса, а именно конструктор. Вызов конструктора, а он неизбежен, может не представлять никаких проблем, а может существенно осложнить поиск проблемы в ситуации, когда тест не проходит.

Назначение конструктора – конструировать объект. Логика конструктора может включать в себя компоновку объекта с его зависимостями, выставление значений полей по умолчанию, обращение к внешним ресурсам для конфигурации объекта и т.д. Все эти действия могут квалифицироваться как конструирование объекта, однако при этом ответственность конструктора может стать чрезмерной.

Проверка аспектов работы конструктора, как и любого метода класса, это нормальная задача для модульного теста. Если логика конструктора сложна, тестов может быть много, это тоже нормальная ситуация для любого метода. Однако в случае сложного конструктора изолированное тестирование других методов класса затрудняется. Ведь для того, чтобы создать объект класса необходимо вызвать его конструктор и выполнить всю его логику. Другими словами концептуально секция Act будет состоять уже из двух вызовов, что не является приемлемым для модульного теста.

У этого скрытого недостатка кода класса существует несколько специфических симптомов, которые проявляются, как в коде класса, так и в коде тестов:

1) Перегруженный конструктор, который используется только в коде тестов. Наиболее явный симптом, указывающий на то, что класс трудно тестировать. Часто при написании тестов после написания класса в нём появляется явно определённый конструктор по умолчанию для создания объекта без инициализации. У такого класса рабочий конструктор выполняет слишком много действий, что приводит к трудностям в понимании тестов.

2) Зависимости, внедряемые черезконструктор, которые нигде далее в коде класса не используются. Правильным приёмом использования модели внедрения черезконструктор является сохранение (компоновка) объектов зависимостей для последующего использования в закрытые или защищённые поля класса только для чтения. Если этого не происходит и зависимость используется только для инициализации объекта в конструкторе, он теряет простоту и ограниченность ответственности. Тестировать такой класс становится сложно.

3) Вызовы методов объектов зависимостей. Данный симптом (помимо того, что он включает в себя оба предыдущих) может указывать на то, что класс использует свои зависимости не по назначению. Как следствие любой тест для данного класса, независимо от проверяемого аспекта работы будет вынужден дополнительно конфигурировать соответствующее поведение двойников зависимостей.

4) Передача в конструктор объектов значений или сущностей. Объекты-значения это аргументы примитивных типов, плоские объекты или объекты других классов, следующих модели (паттерну) Объект-значение [4]. Сущность – это класс, представляющий персистентные данные, значимые для предметной области своей уникальностью и идентичностью. Типичным примером являются конструкторы классов обёрток для объектов-сущностей [4], принимающие

саму сущность либо её уникальный идентификатор. Подобные конструкторы могут быть нагружены логикой инициализации на основе передаваемых объектов значений и являются примером ошибочного проектирования согласно модели (антипаттерну) Ограниченное конструирование [3]. Ограниченное конструирование усложняет внедрение зависимостей и вместе с ним модульные тесты.

Беглый взгляд на приведённые симптомы и проблемы, которые за ними скрываются, позволяет сказать, что все они связаны с нарушением принципа единственности ответственности. Конструктор выполняет много работы неявно, не предупреждая, что будет занят чем-то кроме компоновки объекта зависимостями. Компоновка объекта – это единственная обязанность конструктора, которая явно следует из его сигнатуры. Именно этой работы от него всегда можно ожидать наряду с проверкой аргументов с помощью защитных выражений [3]. Того же ожидает от него и модульный тест, цель которого проверить отдельный аспект работы другого метода класса.

Решение проблемы правильного конструирования объекта лежит в концепции внедрения зависимостей, которая, как уже было сказано, тесно связана с организацией модульного тестирования. Однако задача остаётся нетривиальной во многих случаях на практике проектирования и разработки.

Управление зависимостями. Внедрение зависимостей черезконструктор – это метод внедрения зависимостей тесно связанный с проблемой перегруженных логикой конструкторов. Данный паттерн является предпочтительным [3] среди остальных паттернов внедрения, так как он гарантирует, что после вызова конструктора клиентский код получит готовый к работе объект в согласованном состоянии. Тем не менее, эта гарантия распространяется на наличие у объекта всех необходимых зависимостей, однако не подразумевает дополнительной логики связанной с использованием этих зависимостей. Тезис об ограниченной ответственности для конструкторов часто упускается из виду.

Распространение идей инверсии управления и внедрения зависимостями породило класс служебных библиотек и целых каркасов приложений (фреймворков), назначение, которых заключается в облегчении управления зависимостями в крупных программных комплексах. Контейнеры инверсии управления (Inversion of Control Container, IoC-контейнер) позволяют не только автоматизировать сборку слабосвязнной композиции объектов, но и управлять временем их жизни и осуществлять внедрение сквозных аспектов функциональности (Аспектно-ориентированное программирование). В современных проектах контейнер инверсии управления помогает собирать рабочую структуру программы из объектов слабосвязных классов на основе заранее заданной конфигурации.

Контейнер инверсии управления может скомпоновать целое дерево объектов и предоставить его вызывающему коду всего одним вызовом метода. Как правило, этот метод имеет имя Resolve() – «Преобразовать». Возможность контейнера автоматизировать запуск множества конструкторов в одном вызове метода Resolve() может сыграть негативную роль в свете описанной проблемы с нарушением границ ответственности конструктора. Здесь есть два аспекта:

1) Вызов метода Resolve() может успешно скрывать работу конструкторов любой сложности и любым количеством передаваемых в аргументы зависимостей. Большое число аргументов и сложная логика инициализации неизбежно приведут к проблемам при написании модульных тестов. Подобные ситуации характерны для разработки тестов после написания кода модуля.

2) С ошибками, возникающими в конструкторах сложнее работать. Любая необработанная ошибка в ходе вызова Resolve() сорвёт процесс компоновки и будет транслирована в составе исключения. Как правило, библиотеки контейнеров управления зависимостей предоставляют собственные классы (или вовсе один класс) исключений для обработки таких ошибок. При этом информация об исходном исключении, которое сгенерировал конструктор может быть завёрнута в одну или несколько обёрток. Если границы ответственности конструк-

торов не нарушены, при работе с контейнером можно ожидать, что вызов Resolve() сорвётся из-за ошибочной или незавершённой конфигурации контейнера. В этом случае ошибку будет легко обнаружить. А вот ошибку инициализации будет найти гораздо сложнее, она может произойти в любом из конструкторов, которые вызываются при компоновке дерева объектов.

Применяется ли контейнер для управления зависимостями или нет, оно будет осложнено последствиями Ограниченного Конструирования, если конструкторы занимаются чем-либо помимо компоновки объекта зависимостями. Однако при использовании контейнера инверсии управления в коде фабрик [8], которые к нему обращаются, может проявиться характерный симптом, могущий указывать на нарушения границ ответственности конструкторов. Напротив, при управлении зависимостями без использования контейнеры, код фабрик никак не будет указывать на наличие проблемы.

Если конструктор принимает, только объекты классов зависимостей, ранее зарегистрированных в контейнере, любой необходимый объект можно получить вызовом метода Resolve(). Для передачи черезконструктор дополнительных параметров, значения которых заранее не известны контейнеру придётся прибегать к различным ухищрениям. Большинство широко используемых библиотек контейнеров содержат функциональность позволяющую внедрять зависимости, используя имена, типы или позиционирование параметров конструктора.

Применение подобных механизмов создаёт отложенные проблемы с рефакторингом кода в будущем, так как корректность внедрения параметров зависит от изменчивой сигнатуры конструктора, а то и вовсе от строкового представления имени аргумента. Кроме того становится невозможным применение паттерна Register-Resolve-Release[3] так как невозможно ограничиться всего одним вызовом метода Resolve(). Проблема становится особенно остро, когда обращение к Resolve() может происходить в течение работы приложения множество раз.

Заключение

Разработанные модели и методы организации модульного тестирования и управления зависимостями хорошо зарекомендовали себя в отрасли. С их помощью разработчики приходят к пониманию принципов проектирования слабосвязных классов и модулей, раннего обнаружения ошибок и безопасного рефакторинга. Однако на практике часто возникают ситуации, когда формально слабосвязный дизайн порождает трудные для понимания тесты и большие сложности в управлении зависимостями. Другими словами, пригодность для модульного тестирования[3] это только необходимое условие, при котором тесты будут полезны.

Как было показано, нередко желание разработчиков следовать предлагаемым моделям порождает неверные проектные решения. Возникают ситуации, в которых формального описания и поверхностного понимания концепции недостаточно для принятия правильного решения. Более того, негативные последствия неправильных проектных решений объясняются недостатками самой концепции. Модульное тестирование, внедрение зависимостей, слабая связность объявляются ненужными усложнениями и лишней работой. Порог входа для этих концепция остаётся довольно высоким.

Поэтому разработка более конкретных, хорошо формализованных и каталогизированных моделей и методов, которые помогли бы справиться с описанными проблемами в этих областях является актуальной задачей отрасли и сегодня. Существует потребность в детализации предлагаемых на сегодняшний день общих принципов. Часто неверному применения теории способствуют широкие возможности технологий и инструментов (в частности контейнеров инверсии управления и каркасов изоляции), которые легко использовать в ущерб устройству тестируемых классов. Углублённые, более специализированные модели и методы проектирования, могли бы помочь в правильном толковании основополагающих идей и принципов, а перечисленные симптомы

неудачных проектных решений заметить проблему раньше, чем она даст о себе знать в качестве отказа от инверсии управления или модульного тестирования.

Литература

1. *Osherove R.* The Art of Unit Testing with Examples in .NET. – Greenwich: Manning, 2009. – 324 с.
2. *Meszaros G.* xUnit Test Patterns. Refactoring Test Patterns. – Boston: Addison-Wesley, 2007. – 948 с.
3. *Симан М.* Внедрение зависимостей в .NET. – Спб.: Питер, 2013. – 464 с.
4. *Фаулер М.* Шаблоны корпоративных приложений. – М.: Вильямс, 2011. – 544 с.
5. *Фаулер М.* Рефакторинг. Улучшение существующего кода. – СПб.: Символ-плюс, 2010. – 432 с.
6. *Бек К.* Экстремальное программирование. Библиотека программиста. – СПб.: Питер, 2003. – 212 с.
7. *Мартин Р.С.* Принципы, паттерны и методики гибкой разработки на языке C#. – СПб.: Символ, 2012. – 768 с.
8. Приёмы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон и др. – СПб.: Питер, 2012. – 368 с.
9. *Кожевников Д.О., Рудакова Г.М.* Обобщение инфраструктурного дизайна корпоративных приложений на основе классов доменной модели на платформе .Net // Информатизация и связь. 2013. № 2. С. 31-35.

Actual problems of the organization of unit testing of classes.

Dmitry Olegovich Kozhevnikov, postgraduate
Siberian State Technological University, Chair of Informational Technologies

This work is devoted to description and the analysis of some problems of development of unit tests which are connected with shortcomings of a tested code. Distinguishing attainment suitability to unit testing and obtaining short and clear unit tests, the material of work concentrates on the description of design decisions which lead to emergence of unit tests bulky and difficult for reading. Symptoms and signs of such decisions in a tested code of the program are analyzed.

Key words: unit testing, refactoring, low coupling, dependency injection, test double, constructor, inversion of control container.

УДК 519.8

РЕШЕНИЯ ЗАДАЧ ОРТОГОНАЛЬНОГО РАСКРОЯ-УПАКОВКИ НА ОСНОВЕ КОНСТРУКТИВНЫХ И НЕЙРОСЕТЕВЫХ ПОДХОДОВ

***Оксана Валерьевна Корчевская**, к.т.н, доцент*
Тел 8 906 9713576,e-mail: okfait@gmail.com
ФГБОУ ВПО «Сибирский государственный технологический университет»
www.kit-stu

Представлена постановка задачи n-мерной упаковки(n = 1, 2, 3). Приведено описание метода плоскостей для решения задачи трёхмерной упаковки. Приведена композиция этого метода для задачи двухмерной упаковки. Для определения нижних границ решения задач раскроя-упаковки применён аппарат нейронных сетей. Разработаны высокоэффективные алгоритмы решения задач двух и трёхмерного раскроя-упаковки, позволяющие быстро строить карты раскроя с коэффициентом раскроя, в среднем, от 85%. Достоверность полученных результатов диссертации подтверждается сравнительным анализом существующих подходов к решению поставленной задачи и результатами экспериментальных данных.