Инструмент обратного проектирования и рефакторинга программного обеспечения написанного на языке Java.

Романов В.Ю.

***Аннотация* — В статье рассматриваются особенности реализации CASE-инструмента для моделирования свободно распространяемого программного обеспечения разработанного на языке Java. Данный инструмент предназначен для моделирования и визуализации такого программного обеспечения с помощью языка UML, верификации и рефакторинга его архитектуры. В качестве анализируемого исходного кода для такого CASE-инструмента предполагается использовать репозитории кода с унифицированной идентификацией распространяемых программных библиотек. В статье описывается контекст, в котором предполагается работа данного инструмента. Рассмотрены типовые ошибки в проектировании логической и физической структуры программного обеспечения, методы верификации и рефакторинга для исправления таких ошибок проектирования.**

***Ключевые слова* — artifact repository, software architecture, UML.**

I. Введение

Использование свободно распространяемых в сети Internet библиотек, разработанных на языке Java, получает все более широкое распространение. Многие такие библиотеки распространяются в исходных текстах, что дает возможность их развития независимыми группами разработчиков. Для распространения таких библиотек все чаще используются не только сайты разработчиков библиотек, но и централизованные хранилища (репозитории) со стандартизованным интерфейсом. Это позволяет ссылаться на библиотеки непосредственно по их уникальному адресу в репозитории. Разработка программного обеспечения в таком случае представляет собой сборку программного обеспечения с указанием уникального идентификатора группы (фирмы или команды разработчиков), уникального идентификатора артефакта (библиотеки) и номера версии артефакта. Заданная в проекте с помощью таких координат библиотека автоматически подгружается из глобального репозитория в интернете в локальный репозиторий проекта и используется при сборке программы. Требуемые загруженной библиотекой другие библиотеки также рекурсивно подгружаются в локальный репозиторий и используются при сборке вновь разрабатываемого программного обеспечения на языке Java. Широкое распространение получили также инструменты [1, 2] позволяющие автоматизировать с помощью таких координат описание зависимостей между библиотеками и их сборку. Разработана унифицированная структура репозиториев для хранения таких библиотек [2] и программный интерфейс [3] для доступа к репозиториям артефактов из различного рода инструментов автоматизации сборки.

Вместе с тем использование свободно распространяемых через репозитории библиотек сопряжено с определенными трудностями. Существует множество различных библиотек с аналогичной функциональностью, либо множество версий одной и той же библиотеки с различными функциональными возможностями. Некоторые библиотеки со схожими возможностями развиваются параллельно независимыми группами разработчиков по своему усмотрению. Многие получившие популярность библиотеки уже никем не сопровождаются. При несогласованной совместной разработке библиотек зачастую совершались ошибки проектирования, что теперь затрудняет их последующее развитие и использование. Как следствие, использование свободных библиотек требует существенного времени на их сравнение, на выбор подходящих для конкретного проекта, на изучение функциональных возможностей этих библиотек. Существенную помощь в этом оказывает наличие моделей этих библиотек, представленных с помощью наглядной графической нотации языка UML.

II. Моделирование архитектуры программного обеспечения

Стандартизованная структура байт-кода библиотек, разработанных на языке Java, позволяет выполнить построение их UML[4,5,6] модели даже при отсутствии исходных текстов этих библиотек. Наличие UML-модели существенно облегчает и ускоряет понимание внутренней структуры изучаемой библиотеки. Графическая нотация языка UML позволяет представить на одной диаграмме в компактной форме информацию, которая зачастую рассредоточена во многих текстовых файлах написанных на языке Java. Вместе с тем, при большом размере хранимого в репозиториях и анализируемого для возможного использования программного кода, полная визуализация кода с

Владимир Юрьевич Романов, факультет ВМК МГУ им. М.В. Ломоносова, vromanov@cs.msu.su, vladimir.romanov@gmail.com.

помощью языка UML может оказаться затруднительной. Построение диаграмм, отражающих все связи между элементами программ, может потребовать слишком много времени. Восприятие графического представления при этом по-прежнему будет затруднено из-за чрезмерно большого числа UML-диаграмм. Поэтому необходима визуализация лишь наиболее существенной информации о программной системе – ее архитектуры.

В результате совместной работы международных организаций по стандартизации ISO и IEEE был принят стандарт, определяющий, что может считаться описанием архитектуры программного обеспечения [7]. Были также изданы книги [8,9] являющиеся хорошими комментариями к данному стандарту. В стандарте и упомянутых книгах формализованным образом описана метамодель архитектуры. На рисунке 1 приведен пример формального определения ключевых понятий архитектуры.

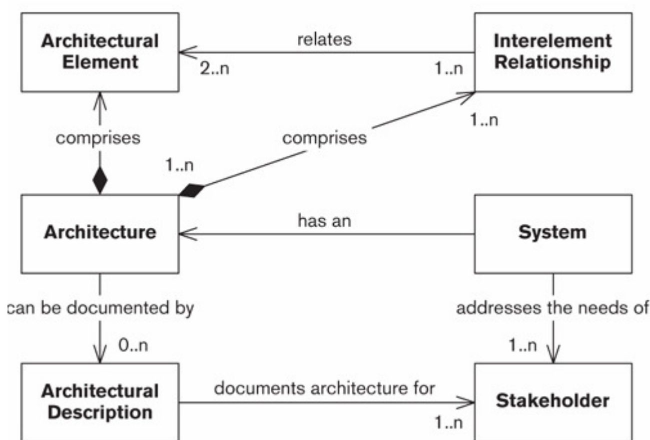

Рис. 1. Ключевые понятия архитектуры и их связи.

С помощью языка UML в модели и соответствующих им диаграммах определяются элементы, из которых строится описание архитектуры (architectural elements), категории участников заинтересованных в реализации программной системы (stakeholders). Для каждой категории участников из элементов строится описание архитектуры учитывающей специфику этой категории. В стандарте определен также каталог проекций (views) и точек зрения на модель (viewpoint) с подмножествами типов элементов архитектуры, из которых они строятся. Определен процесс построения архитектуры: соглашение о границах и контексте системы, идентификация и привлечение участников, идентификация и использование сценариев архитектуры, использование стилей и шаблонов архитектуры, порождение моделей архитектуры, документирование архитектуры. В стандарте определены семь точек зрения (viewpoints) на архитектуру: контекст, функционал, информация, параллельность, разработка, внедрение и эксплуатация. Для каждой точки зрения на архитектуру определен набор моделей, которые могут быть представлены с помощью графической нотации языка UML. Так, например, для точки зрения «контекст» строится модель прецедентов использования языка UML. Для точки зрения «функционал» строятся модель компонент со связующими их интерфейсами и соединителями, а также диаграммы деятельности, описывающие потоки управления и информации. Для точки зрения «информация» используются модели статической структуры классов, модели потока информации описываемой диаграммами классов, модели жизненного цикла информации, описываемого диаграммами переходов и состояний. Для точки зрения «параллельность» используются модели классов описывающих процессы и нити, а также модели переходов и состояний. В стандарте на архитектуру также регламентируется деятельность системного архитектора при построении им описания архитектуры.

Частично работа архитектора по построению модели архитектуры программного обеспечения (элементов модели, их проекций и различных точек зрения участников проекта) и ее визуализации, выполняемая для хранимого в репозиториях свободного программного кода, может быть автоматизирована CASE-инструментом [10,11,12].

III. Верификация и рефакторинг библиотек

Многие библиотеки, предоставляющие проекту необходимую функциональность, требуют переработки (рефакторинга) исходного текста библиотеки. Зачастую это обусловлено ошибками проектирования совершенными при не согласованной работе многих разработчиков библиотеки меняющихся на протяжении нескольких лет. Такие ошибки проектирования приводят к необходимости подключения той функциональности, которая реально в проекте не используется. Не используемый в проекте код в свою очередь может требовать подключения других библиотек. В итоге размеры подключаемого и реально используемого программного кода могут отличаться в несколько раз.

Другой распространенной причиной, которая требует рефакторинга библиотек, является отсутствие исходных текстов у части используемых ими свободно распространяемых библиотек. Для некоторых проектов критичным является отсутствие в используемом коде так называемых «закладок», выполняющих непредсказуемые действия. Например, передачу информации об окружении исполняемой программы по сети интернет. Поэтому отсечение или замена кода, использующего библиотеки без исходных текстов, для таких проектов требует понимания архитектуры библиотеки показывающей связи исключаемого или заменяемого кода с остальным кодом библиотеки.

Для выявления ошибок проектирования библиотеки CASE-инструментом выполняется верификация архитектуры библиотеки. Разработан ряд объектно-ориентированных метрик [13] позволяющих выявить критические участки в коде библиотеки. С помощью таких метрик измеряются сцепление классов, сцепление методов классов, глубина наследования и многие другие характеристики логической структуры библиотеки. Измерение качества проекта с помощью метрик

позволяет также выявить нарушение основных принципов проектирования. При проектировании классов (логической структуры) получили широкое распространение принципы, получивших сокращенное наименование SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion)[14]. Принцип единственности обязанности (Single responsibility) требует, что бы на каждый проектируемый класса была возложена единственная обязанность. Таким образом, становится меньше причин, по которым возможно потребуется изменять этот класс. Принцип открытости-закрытости (Open-closed) требует от классов возможности расширения их поведения без модификации кода классов. Принцип подстановки (Liskov substitution) требует, что бы замена экземпляров классов предков на экземпляры классов потомков не изменяла свойства программы. Принцип разделения интерфейсов (Interface segregation) требует, что бы классы-клиенты интерфейсов не зависели от методов интерфейса, которые они не используют. Принцип инверсии зависимостей требует, что бы зависимости в системе строились на основе абстракций. Классы, расположенные на верхних уровнях системы, не зависели от классов, расположенных на нижних уровнях системы. Классы всех уровней должны зависеть от абстракций (абстрактных классов и интерфейсов). Абстракции не должны зависеть от специализированных классов и интерфейсов. Специализированные классы должны зависеть от абстракций. Таким образом, объектно-ориентированные метрики позволяют выявить ошибки проектирования в логической структуре проекта (в исходных текстах программ на языке Java) и выполнить рефакторинг логической структуры с помощью CASE-инструмента.

Большое значение для использования и сопровождения свободно распространяемых библиотек имеет также их физическая структура - декомпозиция библиотек на модули. В настоящий момент в языке Java отсутствует явно понятие модуля на уровне конструкций языка. Фактически роль модуля в языке Java выполняют упакованные в файлы с расширением jar исполняемые файлы (байт-код) классов языка Java (компоненты). Отсутствие модульности на уровне самого языка приводит к часто встречающимся ошибкам проектирования, когда классы из одной компоненты имеют непосредственные связи с классами из других компонент [15]. Это существенно затрудняет сопровождение, использование и развитие библиотек состоящих из таких модулей.

Для преодоления таких ошибок проектирования связи между компонентами системы в некоторых проектах явно описываются на основе получающей все более широкое распространение спецификации OSGi[16]. При использовании спецификации OSGi описание допустимых связей между компонентами включается в файл манифеста компоненты. Такие явно описанные связи используются CASE-инструментом при построении модели компонентов и ее визуализации.

В настоящее время ведется работа по расширению самого языка Java модульными конструкциями в рамках проекта JIGSAW[17]. Эти конструкции языка можно будет использовать, начиная с версии языка Java 8. Фирмой Oracle в версии языка Java 8 для декомпозиции платформы предполагается использовать компактные профили (Compact Profiles) [18]. Полностью модульные конструкции для декомпозиции самой платформе Java предполагается использовать, начиная с версии Java 9. В CASE-инструменте модульные конструкции Java предполагается использовать при переносе библиотек на платформу Java 8.

Для верификации взаимосвязей компонент библиотек выполняется анализ структуры компонент и выявление типовых ошибок проектирования [15, 14]. Так для проектирования физической структуры компонентов, разработан ряд хорошо зарекомендовавших себя принципов проектирования компонент [19]. Принцип размещения в одной компоненте совместно используемых (common reuse) и совместно изменяемых (common closure) классов. Другим широко используемых принципом проектирования является ацикличность графа зависимости компонент. Определены методы рефакторинга с использованием конструкций языка Java[15], позволяющего удалить такие циклические зависимости между компонентами.

Для анализа качества проектирования физической структуры системы Робертом Мартином [19] разработана совокупность метрик. Метрика центробежного сцепления (Efferent Coupling) определяет количество классов внутри компоненты, которые зависят от классов вне этой компоненты. Метрика центростремительного сцепления (Afferent Coupling) определяет количество классов вне компоненты, которые зависят от классов внутри компоненты. Эта пара метрик используется для вычисления метрики нестабильности $I = Ce / (Ca + Ce)$ изменяющейся в пределах [0..1]. Значение 1 имеет компонента с максимальной нестабильностью. Исходя из значений метрики стабильности, определяется принцип стабильной зависимости (Stable Dependency). Зависимости между компонентами должны быть направлены в направлении стабильных компонент. Компонента должна зависеть от компоненты более стабильной, чем она сама. Таким образом, формируется стабильный верхний уровень (ядро) проектируемой системы. Дополнительно вводится метрика абстрактности компоненты A = (количество абстрактных классов и интерфейсов) / (общее число классов компоненты). Затем предлагается принцип стабильных абстракций (Stable Abstractions). Максимально стабильные компоненты должны быть максимально абстрактными. Максимально нестабильные компоненты должны быть максимально конкретными. Для остальных компонент должна сохраняться пропорция абстрактности и стабильности. В сумме они должны быть равны 0. Таким образом, существует основная последовательность (Main Sequence) вдоль которой должны располагаться компоненты, как показано на рисунке 2.

Компоненты, расположенные ближе к основной последовательности, менее чувствительны к изменениям

и допускают большую повторную используемость. Для измерения вводится метрика D для измерения расстояния от линии основной последовательности.

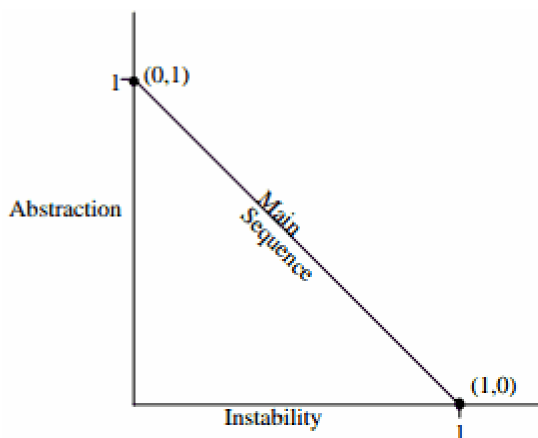

Рис. 2. Основная последовательность для компонент.

Анализ физической структуры библиотек [15] позволил выявить множество хорошо зарекомендовавших шаблонов проектирования компонент, выделив категории таких шаблонов. В частности, категории шаблонов для расширяемости библиотек, категории для уменьшения зависимости между компонентами библиотек, категории облегчающие используемость библиотек. В качестве примеров из категории расширяемости приведем замену зависимостей между классами модулей на зависимости только от абстрактных элементов структуры (интерфейсов и абстрактных классов), использование реализаций интерфейсов и абстрактных классов через фабрики реализаций. Несоответствие указанным шаблонам зачастую свидетельствует об ошибках в проектировании. К таким ошибкам относятся, например, отсутствие разбиения архитектуры компонент на уровни, несоответствие уровней логической и физической структуры, наличие сильного сцепления (cohesion) между компонентами, отсутствие разделения уровней компонент для интерфейсов, реализации и тестирования.

Для выполнения преобразования программного кода CASE-инструментом предполагается использовать программный интерфейс платформы Eclipse[19] предназначенный для рефакторинга программного текста написанного на языке Java. Данный программный интерфейс предоставляет элементарные виды рефакторинга, применяемые к исходным текстам программ, из которых возможно построение более специализированных видов рефакторинга CASE-инструмента. В частности, такой полезный вид элементарного рефакторинга Eclipse, как порождение из существующего класса интерфейса, который данный класс будет реализовать, с включением в интерфейс части методов класса. Или выделение в абстрактный базовый класс части полей и методов выбранного для рефакторинга класса. Таким образом, возможно, например, выделение в отдельные уровни (компоненты) абстрактных классов, или замены зависимостей между классами компонент на зависимости классов от интерфейсов, выделенных в отдельные компоненты.

IV. Заключение

В статье рассмотрены возможности CASE-инструмента для решения задачи обратного проектирования (reverse engineering) свободно распространяемого программного обеспечения. Описаны особенности работы этого инструмента с централизованными хранилищами свободно распространяемых библиотек. Рассмотрены возможности по моделированию и визуализации архитектуры библиотек с помощью языка UML с использованием стандартов на описание архитектуры программной системы. Описаны проблемы, осложняющие использование таких библиотек. Показаны методы верификации логической и физической структуры библиотеки для ее последующего рефакторинга.

V. Примечание редактора

Статья является продолжением цикла публикаций по программной инженерии и применению UML, начатой автором в журнале INJOIT работой [10]. Эта тема давно развивается автором ([11], [12]) и относится к числу приоритетных направлений исследований в Лаборатории ОИТ факультета ВМК МГУ [21]. Мы ожидаем увидеть продолжение публикаций автора и его учеников в нашем журнале.

Библиография

[1] Ivy - The agile dependency manager, http://ant.apache.org/ivy/
[2] Maven architecture, http://maven.apache.org/ref/3.0.5/
[3] Aether- the library for working with artifact repositories, http://www.eclipse.org/aether/
[4] Object Management Group, UML 2.4 Superstructure Specification, OMG document. http://www.omg.org/spec/UML/2.4.1/
[5] Object Management Group, UML Diagram Interchange, OMG document, http://www.omg.org/spec/UMLDI/1.0/PDF
[6] Object Management Group, XML Metadata Interchange, OMG document, http://www.omg.org/spec/XMI/2.4.1/
[7] ISO/IEC/IEEE 42010:2011, Systems and software engineering — Architecture description, http://www.iso-architecture.org/ieee-1471/
[8] Nick Rozanski, Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition, Addison-Wesley Professional, October 25, 2011, ISBN 978-0-321-71833-4
[9] Paul Clements; Felix Bachmann; Len Bass; David Garlan; James Ivers; Reed Little; Paulo Merson; Robert Nord; Judith Stafford, Documenting Software Architectures: Views and Beyond, Second Edition Publisher: Addison-Wesley Professional, October 05, 2010, ISBN 978-0-321-55268-6
[10] Романов В. Ю. Моделирование свободно-распространяемого программного обеспечения с помощью языка UML //International Journal of Open Information Technologies. – 2013. – Т. 1. – №. 7. – С. 11-15.
[11] Романов В.Ю. Реализация метамодели языка UML на основе хранилища данных фирмы Google. Сб. трудов VII Международной научно-практической конференции "Современные информационные технологии и ИТ-образование". М.:, 2012. с.605-610.
[12] Романов В.Ю. Сервис анализа и визуализации кода и текстов на языках программирования как облачное приложение Google. Сб. трудов V Международной научно-практической конференции "Современные информационные технологии и ИТ-образование". М.:, 2011. с.743-748, 12-14 декабря 2011 г.
[13] Michele Lanza, Radu Marinescu, S. Ducasse. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer 2006, ISBN 978-3540244295
[14] Robert C. Martin, Designing Object Oriented Applications using UML, 2d.ed. Prentice Hall, 1999

[15] Kirk Knoernschild , Java Application Architecture: Modularity Patterns with Examples Using OSGi, Addison-Wesley Professional, 2012, ISBN 978-0-321-24713-1
[16] OSGi - The Dynamic Module System for Java, http://www.osgi.org
[17] Standard module system for the Java SE Platform http://openjdk.java.net/projects/jigsaw/
[18] Compact Profiles, http://openjdk.java.net/jeps/161
[19] Martin, Robert C. Agile Software Development, Principles, Patterns, and Practices. Prentice Hall, 2002, ISBN 978-0135974445.
[20] The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs, http://www.eclipse.org/articles/Article-LTK/ltk.html
[21] Намиот Д., Сухомлин В. О проектах лаборатории ОИТ //International Journal of Open Information Technologies. – 2013. – Т. 1. – №. 5. – С. 18-21.

Reverse Engineering Tool for Software Developed in Java Language

Romanov V.Y.

***Abstract* — The article describes reverse engineering tool to model software architecture from free software repositories. The ISO/IEEE 42010 standard is used for modeling and visualization of the software architecture. The object oriented metrics for quality assurance of design and refactoring methods to correct the design bugs are described.**

***Keywords* — artifact repository, software architecture, object oriented metrics, refactoring, UML.**