Использование шаблонов пакетов для анализа архитектуры программной системы

Романов В.Ю.

Аннотация. **Первый шаг при решении задачи обратного проектирования программной системы — восстановление архитектуры системы. Существует множество инструментов, которые для решения задачи обратного проектирования используют подход сверху/вниз, рассматривая иерархию компонент программной системы. Некоторые инструменты при этом предоставляют возможность выполнять эту работу в интерактивном режиме, оперативно предоставляя архитектурно важные виды (точки зрения на систему) на рассматриваемые элементы программной системы. Эти виды позволяют визуально оценить структуру рассматриваемого элемента и его связи с другими элементами. Каждый шаг в навигации по иерархической структуре потребляет вычислительные ресурсы для построения визуального предоставления архитектуры для рассматриваемого элемента модели. В статье рассматривается использование шаблонов пакетов для упрощения решения такой задачи.**

Ключевые слова **— software visualization, package visualization, architecture recovery, package patterns, reverse engineering.**

I. Введение

Разработка программных систем с помощью библиотек с исходными кодами получает все более распространение. Вместе с тем такой подход к разработке имеет и ряд недостатков. Большой объем исходных текстов этих библиотек, наличие большого числа библиотек со схожей функциональностью, наличие большого количества версий библиотеки, зачастую сопровождаемых различными группами разработчиков, существенно увеличивает время на изучение таких библиотек. Это делает актуальной задачу обратного проектирования (reverse engineering) для построения и визуализации модели такого программного обеспечения [1, 2]. После решения задачи обратного проектирования важным становится извлечение из модели архитектуры программной системы и ее визуализация [3]. В предыдущих работах автора особо рассматривались способы визуализации архитектуры системы [4] и визуализации результатов измерения качества программной системы с помощью объектно-ориентированных метрик [5]. В последующей работе [6] сделан обзор и анализ объектно-ориентированных метрик. Рассмотрены простейшие объектно-ориентированные метрики для анализа проектирования отдельных классов. Затем рассмотрены метрики связанности класса, позволяющие оценить качество проектирования структуры класса. Рассмотрены метрики сцепления классов, позволяющие оценить качество проектирования взаимосвязей классов. После этого рассмотрены метрики для оценки связанности и сцепления пакетов. В завершение работы рассматриваются метрики для оценки проекта ряду хорошо зарекомендовавших себя принципов проектирования.

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

Существует множество подходов к извлечению описания архитектуры программной системы из ее кода. При одном из таких из таких подходов модель системы строится снизу-вверх из элементов нижнего уровня. Затем для построенной модели восстанавливается абстрактный уровень группированием связанных элементов [7, 8, 9, 10]. Основной недостаток такого подхода — необходимость знания предметной области, что не всегда возможно при построении архитектуры программной системы разработанной сторонними разработчиками. Такой подход требует большого объема рутинной ручной работы и занимает много времени.

Следующий класс подходов использует для автоматизации проводимой работы методы кластеризации для того, чтобы упорядочить связанное множество элементов модели в подсистемы [11, 12, 13, 14]. Такой подход может существенно ускорить работу, но зачастую приводит к нескольким возможным вариантам декомпозиции программной системы на подсистемы, которые не учитывают семантику анализируемой системы. Такой подход требует значительного времени на анализ и верификацию результатов автоматической работы.

Ряд других подходов [15, 16, 17] является сочетанием первых двух подходов, выполняя извлечение архитектуры программной системы в интерактивном режиме и осуществляя постоянную визуализацию значимых для построения архитектуры элементов системы. Для этого применяются инструменты, позволяющие осуществлять проход и анализ программной системы сверху-вниз, используя иерархическую декомпозицию программной системы. Для программных систем, реализованных на языке Java, такая декомпозиция определяется иерархией вложенности пакетов и классов языка Java.

В данной статье предлагается классификация пакетов языка Java позволяющая упростить интерактивное исследование структуры и выделение архитектурно важных элементов системы, снабжая визуализацию пакетов языка Java информацией упрощающей навигацию по иерархии. Такая классификация основана

на структурных свойствах пакетов и способах их взаимодействия друг с другом.

I. ВИЗУАЛЬНОЕ ВОССТАНОВЛЕНИЕ АРХИТЕКТУРЫ

Существует множество определений архитектуры программной системы. Одним из распространенных определения архитектуры [19] является следующее: архитектура программной системы или вычислительной системы есть структура или структуры этой системы, видимые извне системы, видимые извне свойства этих элементов и видимые извне отношения между ними.

Существует ряд инструментов поддерживающих выделение и восстановление архитектуры системы, в которых существенную роль играют интерактивность и визуализация [16, 17, 18, 20]. Некоторые шаги в этом процессе (извлечение элемента модели программной системы и генерация видов элемента модели) обычно автоматизированы, без какого либо вмешательства человека. Пользователю лишь необходимо сгруппировать элементы модели системы, или решить по какому пути в иерархии наследования ему желательно переместиться для просмотра и анализа.
Хотя для декомпозиции программной системы могут использоваться различные способы: учет структуры директорий, кластеров, пространств имен, в данной статье декомпозицией системы с помощью иерархии пакетов языка Java.
Начиная с самого верхнего уровня абстракции, дается возможность генерировать новые виды, применяя операции раскрытия и сжатия. При раскрытии узла вид обновляется, и узел замещается узлами-детьми. При сжатии узла виды соответствующие узлам-детям заменяются видом для узла родителя.
Новые виды (точки зрения на программную систему) генерируются при каждом перемещении по иерархической декомпозиции пакетов с помощью операций раскрытия и сжатия. При этом для текущего узла-пакета показываются только те виды, которые относятся к визуализации архитектуры программной системы.

Пример1.

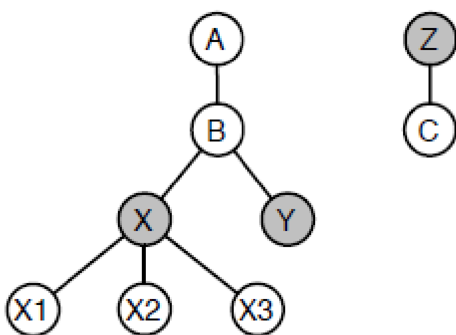

Рис.1. Пример пакетов программной системы

Представленные на рисунке 1 серым цветом пакеты входят в описание архитектуры, поскольку имеют семантику подсистемы. Например, они включают в себя классы важные для описания архитектуры системы. Пакет A архитектурно важным не является, поскольку они представляют лишь часть доменного имени сайта разработчика системы. Например, это доменное имя org. Вид программной системы, в котором пакет X раскрыт при перемещении во вложенные в него пакеты X1, X2, X3, также не будет представлять архитектурную информацию. Например, если пакеты X1, X2, X3 не имеют семантику подсистем, а лишь помогают реализовать функциональность для пакета X, и при этом не имеют никаких зависимостей от пакетов расположенных вне пакета X.

Вид, представляющий не расширенный пакет B, не предоставляет архитектурную информацию, поскольку он только лишь контейнер для пакетов X и Y и не содержит собственной функциональности. Примером для пакета-контейнера B может служить часть доменного имени apache.
Этот пример показывает, что ответственность в принятии решения о том включать ли пакет в описание архитектуры или нет, лежит на пользователе инструмента выполняющего задачу обратного проектирования. При этом он должен будет придерживаться следующих правил:

1. Если пакет более высокого уровня абстракции, чем нужно, то необходимо раскрыть этот пакет.
2. Если пакет правильного уровня абстракции, то не раскрывать этот пакет.
3. Если пакет более низкого уровня абстракции, чем нужно, то необходимо сжать этот пакет.

Как результат ответа на такие вопросы, процесс классификации пакетов автоматизируется.

Далее предлагается классификация пакетов на основе их отношений с другими пакетами в программной системе, а также на основе их внутренней структуры.
В результате такой классификации появляется множество шаблонов (patterns) пакетов, которые имеют связанные с ними операции просмотра иерархической структуры. Эти операции применяются тогда, когда пакет со структурой, удовлетворяющей предлагаемому шаблону, станет текущим при просмотре иерархической структуры.

II. 3. ПАКЕТЫ И ИХ ЗАВИСИМОСТИ

Пакеты в языке Java – основной механизм декомпозиции системы и разделения ее на модули. Пакеты в языке Java определены неявно, поскольку имя пакета задается в описании множества классов. Таким образом, семантика пакета в Java определена недостаточно четко. По этой причине, пакеты далее в статье будут рассматриваться с двух точек зрения.

1. Ограниченный пакет. Это коллекция классов принадлежащих пакету. Классы, определенные во вложенных пакетах не рассматриваются.

2. Расширенный пакет. Это коллекция классов принадлежащих как самому пакету, а также и всем его вложенных пакетам.
3.

Зачастую бывает важно рассматривать пакеты с обеих точек зрения.

Определим зависимость между пакетами как множество всех зависимостей между классами определенными в этих пакетах. Это отношение направленное. Таким образом, можно рассматривать входящие и выходящие зависимости пакета.

В языке Java явно описываются зависимости между пакетами с помощью конструкции *import*. Однако это относится только к описанию зависимостей на уровне ограниченных пакетов. Для понимания отношений между зависимости между пакетами, необходимо рассматривать пакет как расширенный пакет. Для этого необходимо добавить к зависимостям ограниченного пакета зависимости вложенных в него пакетов.

Проиллюстрируем вышесказанное с помощью пакетов и классов, показанных на рисунке 2.

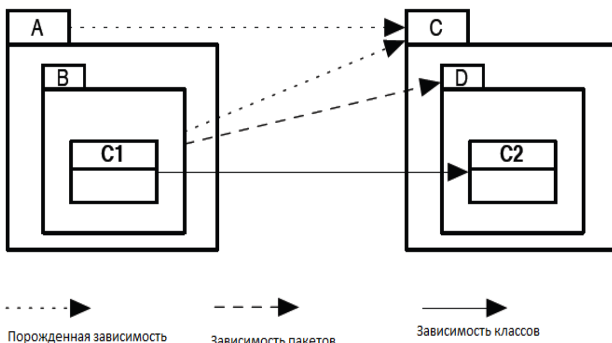

Рис 2. Категории зависимости пакетов и классов.

Между пакетами *A* и *C* существуют зависимости, если эти два пакета рассматривать как ограниченные пакеты. Если пакеты *A* и *C* рассматривать как расширенные пакеты, то зависимость между ними существует. Пакет *A* имеет выходящую зависимость с пакетом C. Пакет *C* имеет входящую зависимость с пакетом *A*.

При просмотре структуры системы существует множество пакетов, которые в данный момент времени видимы одновременно. Эти пакеты далее называются рабочим множеством пакетов.
Разделим ограниченные пакеты на четыре категории:

1. Независимый пакет - нет отношений у этого ограниченного пакета и пакетов рабочего множества.
2. Пакет-потребитель - существует отношение зависимости из этого ограниченного пакета в пакеты рабочего множества.
3. Пакет-поставщик - существует отношение зависимости в этот ограниченный пакет из пакетов рабочего множества.
4. Гибридный пакет - существует двунаправленная зависимость этого ограниченного пакета и пакетов рабочего множества.

Для этих категорий ограниченных пакетов будем использовать следующие графические обозначения.

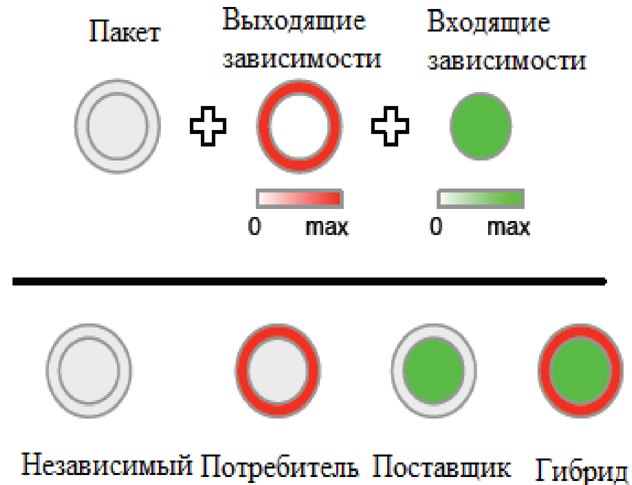

Рис. 3. Категории ограниченных пакетов.

Теперь, используя категории ограниченных пакетов и введенные для них визуальные обозначения, оценим взаимодействие между расширенным пакетов и его рабочим множеством, по категориям вложенных пакетов, как это показано на рисунке 4.

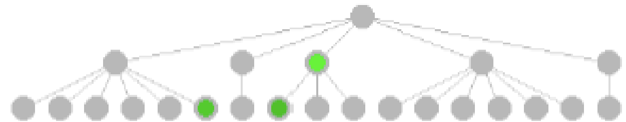

Рис.4. Структура пакета с визуальным изображением категорий вложенных пакетов.

Как видно из рисунка 4, что хотя пакет имеет разветвленную структуру вложенных пакетов, только три пакета предоставляют функциональность для пакетов рабочего множества.

III. Шаблоны в структуре пакетов

Главная проблема при визуальном просмотре системы - решение о том, какие видимые в данный момент пакеты должна быть раскрыты, а какие нет. Поскольку такой анализ после каждого шага просмотра занимает слишком много времени, то для ускорения анализа применяются знания о часто встречающихся шаблонах структур в иерархии пакетов.
Шаблоны описываются с использованием следующей структуры: короткое описание, предложение, правила обнаружения, обоснование и анализ. Правила обнаружения — это множество тестов используемых для проверки соответствует ли пакет шаблону или нет. Если соответствует, то далее следует предложение что делать дальше и обоснование, почему это следует делать. В завершении описания следует дополнительная информация о шаблоне.

A. Шаблон Айсберг

Это пакет, от которого зависят другие пакеты рабочего множества, но зависимость только от ограниченной версии этого пакета. Это значит, что с точки зрения других пакетов, вложенные пакеты рассматриваемого пакета скрыты. Все могут видеть только верхнюю часть айсберга. Пакеты со структурой шаблона Айсберг приведены на рисунке 5.

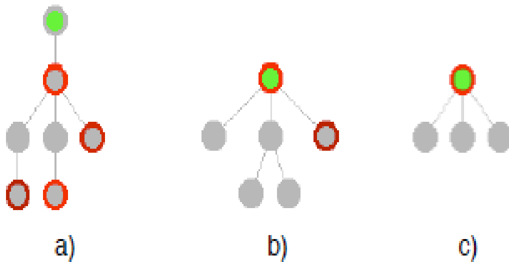

Рис.5. Конфигурация шаблона Айсберг.

Предложение. Данные пакеты не рекомендуется раскрывать.

Обоснование. Хотя вложенные пакеты такого пакета могут использовать функциональность других пакетов в рабочем множестве (это показано красным контуром на рисунке 5), раскрытый пакет действует как единый поставщик функциональности, и его раскрытие не даст ничего для понимания их функциональности.

Правила обнаружения. Пакет Айсберг — это пакет, для которого выполняются следующие правила:
1. Этот пакет (в ограниченном смысле) либо поставщик, либо гибрид.
2. Ни один из вложенных пакетов (в ограниченном смысле) не является пакетом-поставщиком или гибридным пакетом.
Другим признаком, который может использоваться для обнаружения пакетов с таким шаблоном — малое значение у следующего соотношения: доступная_извне_функциональность/ определенная_функциональность.

Анализ. Существует специальный случай шаблона пакета Айсберг, называемый «Идеальный Айсберг», у которого все вложенные пакеты принадлежат категории Независимый. Такой пакет, вероятно, хорошо разделяет программную систему на компоненты и единицы повторного использования. Или же может быть реализацией шаблона проектирования Фасад. Это пакет *C*, показанный на рисунке 5.

B. Шаблон Автономный

Автономный пакет — в котором вложен по меньшей мере один пакет — поставщик, и нет вложенных пакетов потребителей или гибридных пакетов. Короче говоря, автономный пакет не зависит от других пакетов в рабочем множестве. Шаблон пакета провал показан на рисунке 6.

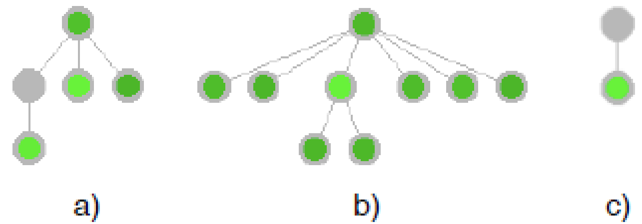

Рис.6. Конфигурация шаблона Автономный.

Предложение. Такие пакеты не рекомендуется раскрывать.

Обоснование. Из определения следует, что пакет не зависит от каких либо пакетов рабочего множества. Это независимый поставщик функциональности.

Правила обнаружения. Пакет Автономный должен удовлетворять двум правилам:
1. По меньшей мере, один вложенный пакет рассматриваемого пакета, или сам пакет, рассматриваемые в ограниченном смысле, должны быть категории Поставщик.
2. Ни один вложенный пакет рассматриваемого пакета, или сам пакет, рассматриваемые в ограниченном смысле, должны быть категории потребитель или гибрид.

Анализ. Шаблон Автономный имеет более строгие правила для определения модульных компонент, чем шаблон Айсберг. Второе правило запрещает иметь какие-либо зависимости от каких-либо пакетов рабочего множества. Если пакет является и Айсбергом, и Автономным, то предложение не раскрывать пакет действует уже для двух шаблонов.
Однако если пакет удовлетворяет и правилам шаблона Автономный, и правилам шаблона Провал (этот шаблон будет рассмотрен далее), то приоритет следует отдавать предложению для шаблона Провал. Этот случай представлен пакетом *C* на рисунке 6.
Существование автономных пакетов в программной системе есть признак хорошего проектирования модулей.

C. Шаблон Архипелаг

Это пакет, который содержит, по меньшей мере, три независимых друг от друга пакета, непосредственно вложенных в пакет-архипелаг. Эти пакеты рассматриваются в расширенном смысле.
Шаблон пакета Архипелаг показан на рисунке 7.

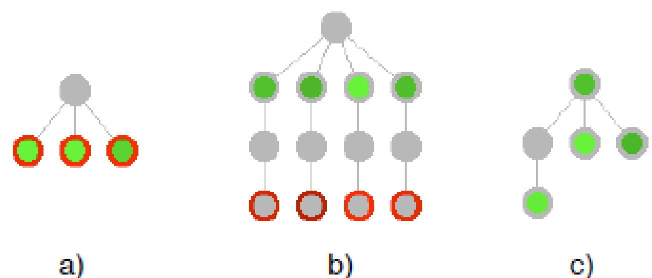

Рис.7. Конфигурация шаблона Архипелаг.

Предложение. Такие пакеты не рекомендуется раскрывать.

Обоснование. Поскольку не существуют вызовы между вложенными пакетами, то нет необходимости во взаимодействии для достижения необходимой функциональности. Такая ситуация может появиться в трех случаях:

1. Пакет содержит альтернативные реализации одной и той же функциональности.
2. Пакет представляет коллекцию сущностей одного рода (плагины, сущности из какой либо одной предметной области).
3. Вложенные пакеты объединены в одни пакет из-за отсутствия других альтернатив расположения пакетов. Например, это может быть пакет утилит.

Только для последнего случая возможно получение дополнительной информации об архитектуре при раскрытии пакета. Можно, однако, предположить, что вложенные пакеты не важны для понимания архитектуры, если они объединены в общий пакет утилит.

Правила обнаружения. Пакет с шаблоном Архипелаг — это пакет, для которого выполняются следующие правила:
1. Существует, по меньшей мере, три непосредственно вложенных пакета.
2. Непосредственно вложенные пакеты в расширенном смысле должны быть независимы.

Анализ. Возможна ситуация, когда между парой вложенных пакетов существует слабая зависимость. Одним из решений может быть оценка степени зависимости пакетов с помощью объектно-ориентированных метрик.

D. Шаблон Провал

Пакет с шаблоном Провал — это пакет, который имеет один вложенный пакет, который в ограниченном смысле является Независимым пакетом. Такой пакет должен быть расширен.
Шаблон пакета Провал показан на рисунке 8.

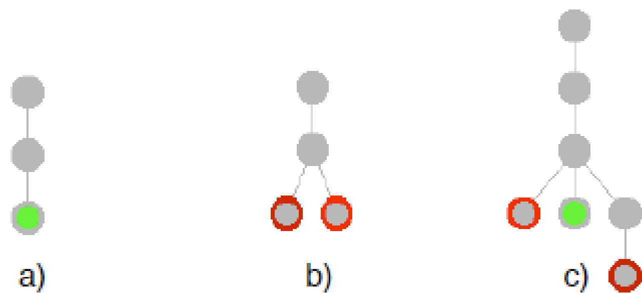

Рис.8. Конфигурация шаблона Автономный.

Предложение. Данные пакеты должны быть раскрыты.

Обоснование. При раскрытии пакета дополнительная информация — только имя вложенного пакета. Однако при раскрытии пакетов на следующем уровне сложности информация может представлять больший интерес.

Правила обнаружения. Пакет с шаблоном Провал - это пакет для которого выполняются следующие правила:
1. Пакет должен иметь только один непосредственно вложенный пакет.
2. Этот пакет, в ограниченном смысле, должен быть Независимым пакетом.

Анализ. Обычно пакеты верхнего уровня системы Java удовлетворяют шаблону Провал. Обычно пакеты верхнего уровня представляют доменное имя сайта автора программной системы.
Если пакет удовлетворяет также и другим шаблонам (например, шаблону Автономный), то предложение для шаблона Провал должно иметь приоритет.

IV. Перекрытие шаблонов

Как уже отмечалось во время определения шаблонов, шаблоны не являются взаимоисключающими. Некоторые пакеты могут удовлетворять правилам нескольких шаблонов одновременно. На таблице 1 показаны такие ситуации «перекрытия» шаблонов.

	Айсберг	Провал	Автономный	Архипелаг
Айсберг	X	Невозможно		+
Провал	Невозможно	X	-	Невозможно
Автономный		-	X	+
Архипелаг	+	Невозможно	+	X

Таблица 1. Перекрытие шаблонов.

Знаками + показаны ситуации, когда рекомендации для шаблонов совпадают. Знаками - показаны ситуации, когда рекомендации для шаблонов противоречат друг другу.
Например, в случае если пакет классифицирован и как удовлетворяющий шаблону Автономный, и как удовлетворяющий шаблону провал Провал, то предложения для этих шаблонов противоречат друг другу. В этом случае приоритет имеет предложение раскрыть данный пакет.

V. Заключение

Представленный в статье подход является лишь первым шагом в направлении автоматической декомпозиции системы на основе структуры ее пакетов. Рассмотренный подход не дает полной автоматизации в двух случаях: когда не применимы предлагаемые эвристики и когда эвристики предлагают применять различные действия к тому же пакету. В этом случае решение о дальнейшей декомпозиции системы должен принимать пользователь инструмента.

Статья является продолжением цикла публикаций по программной инженерии и применению UML, начатой в журнале INJOIT работами [1, 2, 3, 4, 5, 6], а также отраженной в более ранних публикациях [21]. Эта работа относится к числу одного из направлений исследований в Лаборатории ОИТ факультета ВМК МГУ [22].

Библиография

[1] Романов В.Ю. Инструмент обратного проектирования и рефакторинга программного обеспечения написанного на языке Java //International Journal of Open Information Technologies. – 2013. – Т. 1. – №. 8. – С. 1-6.
[2] 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.
[3] Романов В.Ю. Моделирование и верификация архитектуры программного обеспечения разработанного на языке Java. Сб. трудов VIII Международной конференции «Современные информационные технологии и ИТ-образование», Москва, 2013, с. 343-348
[4] Романов В. Ю. Визуализация для измерения и рефакторинга программного обеспечения //International Journal of Open Information Technologies. – 2013. – Т. 1. – №. 9. – С. 1-10.
[5] Романов В.Ю. Визуализация программных метрик при описании архитектуры программного обеспечения //International Journal of Open Information Technologies. – 2014. – Т. 2. – №. 2. – С. 21-28.
[6] Романов В.Ю. Анализ объектно-ориентированных метрик для проектирования архитектуры программного обеспечения//International Journal of Open Information Technologies. – 2014. – Т. 2. – №. 3. – С. 11-17.
[7] M. Pinzger and H. Gall. Pattern-supported architecture recovery. In *Proc. of the 10th International Workshop on Program Comprehension*, pages 53–61, Paris, France, June 2002. IEEE Computer Society Press.
[8] C. Riva. Reverse architecting: an industrial experience report. In *Proceedings WCRE 2000*, pages 42–50. IEEE Computer Society, 2000.
[9] C. Riva. *View-based Software Architecture Reconstruction*. PhD thesis, Technical University of Vienna, 2004.
[10] T. Zimmermann, S. Diehl, and A. Zeller. How History Justifies System Architecture. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 73–83. IEEE Computer Society Press, September 2003.
[11] R. Koschke. An incremental semi-automatic method for component recovery. In *Working Conference on Reverse Engineering*, pages 256–, 1999.
[12] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universitat Stuttgart, 2000.
[13] S. Mancoridis and B. S. Mitchell. Using Automatic Clustering to produce High-Level System Organizations of Source Code. In *Proceedings of IWPC '98 (International Workshop on Program Comprehension)*. IEEE Computer Society Press, 1998.
[14] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, Oxford, England, 1999. IEEE Computer Society Press.
[15] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. M¨uller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. IBM Systems Journal, 36(4):564–593, November 1997.
[16] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 18–28. ACM Press, 1995.
[17] M.-A. D. Storey, K.Wong, F. D. Fracchia, and H. A. Mueller. On integrating visualization techniques for effective software exploration. In INFOVIS '97: Proceedings of the 1997 IEEE Symposium on Information Visualization (InfoVis '97), page 38, Washington, DC, USA, 1997. IEEE Computer Society.
[18] H. A. M¨uller and K. Klashinsky. Rigi – a system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86, Singapore, 1988. IEEE Computer Society Press.
[19] L. Bass, P. Clements, and R. Kazman. Software Architecture in Practice, 2/E. Addison Wesley Professional, 2003.
[20] M. Lungu, A. Kuhn, T. Girba, and M. Lanza. Interactive exploration of semantic clusters. In *Proceedings of VISSOFT 2005 (3rd IEEE International Workshop on Visualizing Software For Understanding and Analysis)*, pages 95–100. IEEE CS Press, 2005.
[21] 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.
[22] Намиот Д., Сухомлин В. О проектах лаборатории ОИТ //International Journal of Open Information Technologies. – 2013. – Т. 1. – №. 5. – С. 18-21.

Romanov V.Y.

Package patterns using for software architecture exploring

***Abstract* — the first step in reverse engineering of software is architecture recovering. Some tools solve the problem by top down exploring of software components hierarchy. But some tools provide possibility to solve the problem interactively and visual, immediately providing architecturally important views by package patterns using.**

Keywords — software visualization, package visualization, architecture recovery, package patterns, reverse engineering.