Визуализация архитектуры программной системы и ее эволюции

Романов В.Ю.

Аннотация. **В статье рассматриваются методы обратного проектирования позволяющие восстановить архитектуру программной системы из ее кода, а также визуализировать и анализировать эту архитектуру. Предлагается метод позволяющий сравнивать различные архитектуры программной системы, основанные на различных способах декомпозиции программной системы и зависимостях между элементами этой системы. В частности, позволяет сравнивать архитектуры для различных версий системы. В статье показывается, как кластеризация программного обеспечения может быть применена на практике.**

Ключевые слова **— software architecture, software decomposition, software elements dependencies, software clustering.**

I. Введение

Данная статья продолжает цикл статей посвященных визуализации и анализу программного обеспечения, выполняемых в CASE-инструментах при решении задачи обратного проектирования. Задача обратного проектирования (reverse engineering) программной системы является весьма важной и частой при разработке программной системы с использованием библиотек с исходными кодами. При решении этой задачи с помощью CASE-инструмента выполняется построение и визуализация UML-модели для вновь разрабатываемой программной системы, а также для используемых системой библиотек. Компактное визуальное представление программы существенно упрощает понимание структуры и функциональности библиотек, выбор необходимой версии библиотеки, команды разрабатывающей и сопровождающей одной из ветвей в развитии этой библиотеки. Способы построения и визуализация модели программного системы были рассмотрены в предыдущих работах автора [1, 2, 3]. Объем информации, получаемой при решении этих задач, может быть слишком велик для их восприятия человеком, а получение и визуализация всех связей может требовать чрезмерно большого времени. Поэтому визуализация программной системы необходима в первую очередь для наиболее существенной ее части - архитектуры. Для построенной UML-модели программы необходимо вычисление и визуализация значений объектно-ориентированных метрик, позволяющих оценить качество проектирования системы. 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 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. В работе автора [7] рассмотрены методы анализа и визуализации программной системы на основе ее трехмерного представления. В работе [8], для визуализации и анализа архитектуры системы, рассматриваются шаблоны пакетов, из которых состоит программная система. В работе [9] - методы визуализации зависимости между пакетами для анализа архитектуры программной системы. В работе [10] рассматриваются методы визуализации и анализа эволюции программной системы, и используемых ею библиотек. Зависимость вновь разрабатываемого программного обеспечения, от библиотек созданных сторонними разработчиками, становится общепринятой практикой, как при разработке программного обеспечения с открытым исходным кодом, так и индустриального программного обеспечения. При этом используются библиотеки с большим объемом исходного кода из больших репозиториев библиотек, например, Source Forge [11] и Maven Central [12]. Эти библиотеки эволюционируют независимо друг от друга, а также от использующего их программного обеспечения. Для сопровождения разрабатываемой системы становится важным отслеживание такой эволюции. В этой работе была рассмотрена визуализация и анализ состояния, и эволюции системы на уровне элементов программы (классов, интерфейсов, перечислений), и связей между ними.

В данной статье рассматриваются методы извлечения архитектуры из программного кода, ее визуализации, анализа и сравнения с другими архитектурами. Решение этих задач позволяет существенно упростить анализ архитектуры эволюционирующих программных систем и используемых ими библиотек.

Наличие описания архитектуры разрабатываемой системы, сделанное квалифицированным разработчиком системы - архитектором, существенно влияет на успешность разработки программной системы. Вместе с тем, такое описание архитектуры обладает существенным недостатком - оно существует отдельно от программной системы. Зачастую свободное программное обеспечение, распространяемое в исходных текстах, такого описания не имеет, или такое описание содержит лишь частично, содержит информацию необходимую для понимания и

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

дальнейшего сопровождения системы. По мере эволюции системы такое описание может устаревать и уже не содержать корректного описания структуры элементов системы, и их взаимосвязей. Необходим инструмент, позволяющий автоматизировать восстановление архитектуры (architecture recovery tool) по исходному коду системы. Такой инструмент обратного проектирования системы (reverse engineering) позволит отслеживать ошибки во время проектирования системы, выявлять несоответствие системы ее описанию, оценить происходящие в системе изменения в ходе ее эволюции.

II. Задача визуализации архитектуры системы

В сложных системах количество анализируемых с помощью матрицы зависимостей элементов программы (библиотечных исполняемых файлов, пакетов языка Java) может достигать нескольких десятков. В таком случае отслеживание зависимостей с помощью приведенной на Рис 4. простейшей матрицы бывает затруднено. Упростить понимание зависимостей может раскраска элементов матрицы. Две ячейки матрицы для взаимозависимых элементов окрашиваются в красный цвет, зависимый элемент окрашивается в зеленый цвет, а элемент, от которого зависят, в синий. Матрица, представленная на рисунке 4, в раскрашенном виде показана на рисунке 5.

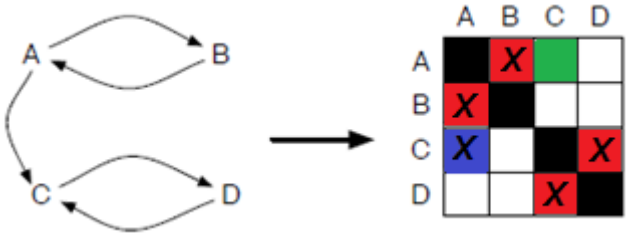

Рис.5. Пример раскрашенной матрицы структуры зависимостей.

На рисунке 6 показан фрагмент матрицы зависимости пакетов содержащихся в библиотеке Java Development Tools (JDT), содержащая, с учетов вложенности пакетов, 83 пакета классов.

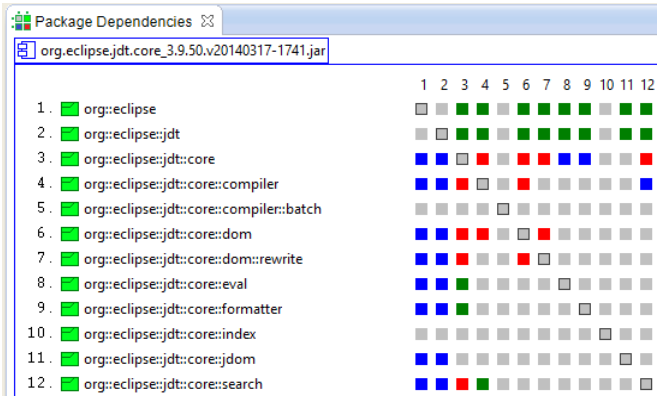

Рис.. Пример раскрашенной матрицы структуры зависимостей для библиотеки Java Development Tools.

Для упрощения визуального анализа объемных матриц возможно их структурирование. Простейший способ - из рассмотрения могут быть убраны вложенные пакеты, и далее рассматриваться лишь зависимости между пакетами верхнего уровня. Для некоторых таких пакетов верхнего уровня циклическая зависимость с другими пакетами верхнего уровня будет отсутствовать, а удаление циклов внутри пакетов верхнего уровня может представлять отдельную, и возможно, простую задачу.

Для оптимизации восприятия зависимостей элементы, которые зависят от других элементов, смещаются вверх/влево, а элементы, от которых зависят, смещаются вниз/вправо. Таким образом, пакеты ядра программы оказываются справа и внизу матрицы, а пакеты верхнего уровня оказываются справа вверху.

Для оптимизации матрицы могут быть использованы алгоритмы разделения (partition) [13] и алгоритмы кластеризации матриц [14, 15]. В случае разделения матрицы строки и колонки матрицы перемещаются (множество отображаемых пакетов сортируется) таким образом, что ячейки зависимых пакетов перемещаются в правый верхний угол. Получается правая треугольная матрица, как показано в примере на рисунке 8.

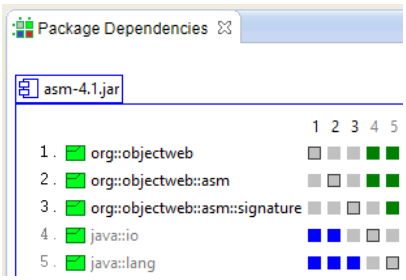

Рис.8. Пример треугольной матрицы зависимости после применения операции разделения.

Простейшие циклы, когда два пакета зависят друг от друга непосредственно, показываются в матрице красным цветом. В больших системах возможны и более сложные циклы с зависимостями - посредниками. Так, например, на рисунке 9 показана зависимость пакетов A -> B->C->A.

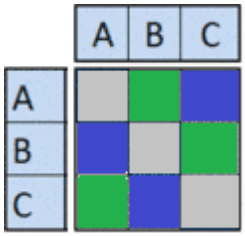

Рис.9. Пример сложного цикла зависимостей.

В рассматриваемом случае зависимость B->C является зависимостью - посредником.

Для поиска сложных циклов используется алгоритм, описанный в работе [16].

Важно отметить, что приведенные в качестве примеров матрицы структурной зависимости показывали лишь наличие зависимостей между парами пакетов. Какова причина существующей между пакетами зависимости на

матрице не показывалось. Также не показывалось, насколько сильна такая зависимость, сколько классификаторов (классов, интерфейсов и перечислений) со стороны каждого пакета в этой зависимости участвуют. Вместе с тем, такая информация имеет большое значение при реструктурировании пакетов с целью уменьшения зависимости пакетов, удаления таких ошибок проектирования, как циклы зависимостей, выявления уровневой организации сложной программной системы. Для детальной визуализации причин и степени зависимости пакетов необходим отдельный вид, показывающий содержимое ячейки-зависимости. Вместе с тем с самой матрице структурной зависимости необходимо показать, какие ячейки заслуживают детального рассмотрения.

Частично перечисленные задачи визуализации могут быть решены простановкой числовых значений в ячейки матрицы. На рисунке 10 в ячейках матрицы показано количество участвующих в отношении зависимости классификаторов.

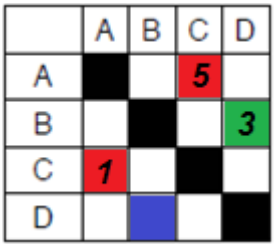

Рис.10. Оценка степени зависимости пакетов с помощью числовых значений.

Между пакетами A и C на рисунке 10 существует взаимозависимость, что показано в матрице ячейками красного цвета. В пакете A пять классов зависят от классов в пакете C. В пакете C один класс зависит от классов в пакете A. Возможно перемещение этого класса из пакета C в пакет A приведет к исчезновению взаимозависимости пакетов, что позволит структурировать программную систему на уровни.

III. Визуализация и анализ архитектуры

Отсортированная матрица структурной зависимости позволяет выявить в существующей системе архитектурные шаблоны [17]. Например, на рисунке 11 правый верхний угол матрицы пуст. Это означает, что пакеты системы распределены по уровням.

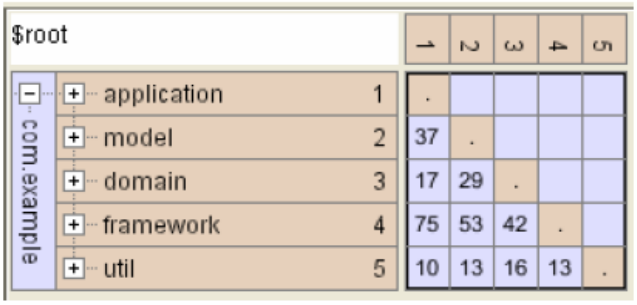

Рис.11. Система с пакетами, распределенным по уровням.

На рисунке 12 показана система, в которой каждый уровень зависит только от уровня расположенного непосредственно ниже него.

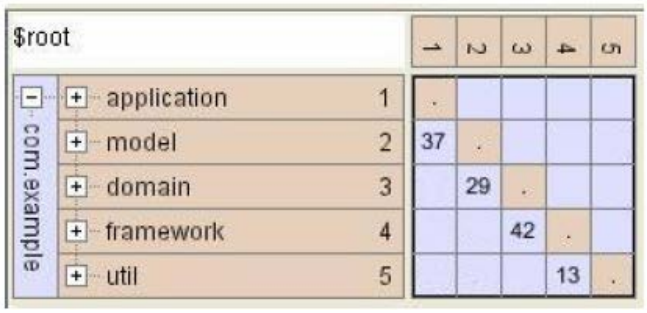

Рис.12. Система с полным распределением по уровням.

На рисунке 13 показан шаблон проектирования Посредник. В матрице показан класс Project, который зависит от большого числа других пакетов. В тоже время от этого класса также зависит большое число других пакетов.

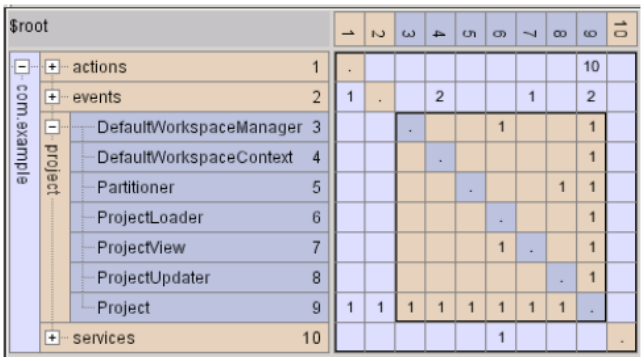

Рис.13. Система с шаблоном проектирования Посредник.

Наличие классов-посредников делает программные системы хрупкими, поскольку они увеличивают вероятность того, что эффект изменений будет передан непропорционально большой части системы.
Так изменение в пакете Services повлияет почти на всю систему. Класс ProjectLoader зависит от Services, класс Project зависит от ProjectLoader, и каждая подсистема зависит от Project.

Достоинства модульных систем в значительной степени приходят от скрытых подсистем. Скрытые подсистемы могут быть легко заменены. Их проще сопровождать, поскольку они имеют ограниченный и четко определенный интерфейс с системой и, следовательно, не зависят от большинства модификаций системы.

На рисунке 14 показаны две подсистемы Comp-1 и COMP-2, которые считаются скрытыми внутри домена подсистемы, так как ни одна другая подсистема не зависит от них.

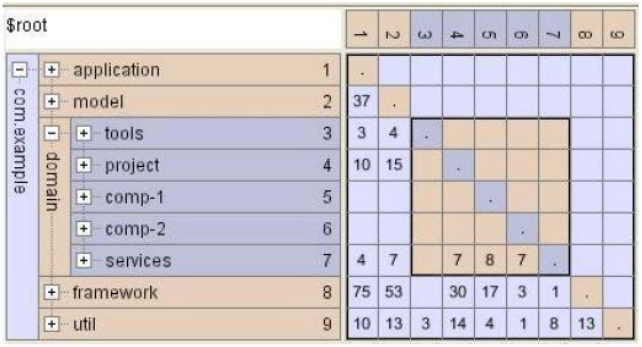

Рис.14. Две скрытые подсистемы comp-1 и comp-2.

Основное преимущество матричного представления по сравнению с графовым представлением в том, что преобладание зависимостей в нижней треугольной части легко увидеть уровневую модель даже тогда, когда система спроектирован с дефектами. На рисунке 15 показана система, которая не полностью уровневая из-за зависимостей в колонке 5.

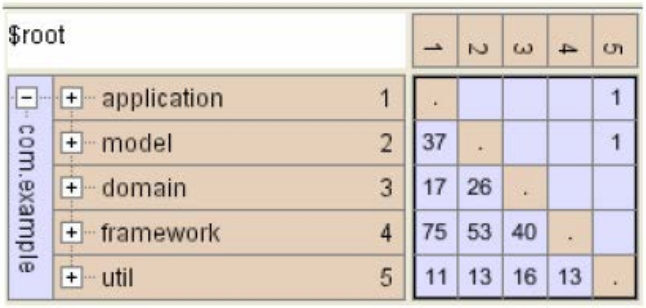

Рис.15. Уровневая система с дефектами.

Модуль Util зависит от пакетов application и model, но числовые значения в ячейке матрицы позволяют предположить, что эта зависимость не столь сильна, как у обратной зависимости пактов application или model от пакета Util.

IV. Сравнение архитектур программной системы

Существует множество инструментов [18] предназначенных для визуализации архитектуры программной системы. Однако, в этих инструментах в каждый момент времени визуализируется только одна архитектура системы. Далее рассматриваются методы визуализации системы, позволяющие анализировать и сравнивать две возможные архитектуры системы, выявляя сходства и различия таких архитектур. Это могут быть как архитектуры двух различных версий системы, так и две архитектуры системы, декомпозиция элементов которых была выполнена на основе различных критериев.

Инструмент обратного проектирования предназначен для решения следующих трех задач.

Первая задача - проанализировать и сравнить зависимости между программными элементами. Зависимости между элементами могут возникать по многим причинам. Один класс может являться наследником другого класса. Метод одного класса вызывает метод другого класса, что также приводит к появлению зависимости между классами. Два класса могут одновременно изменяться в ходе эволюции системы и поэтому также могут быть зависимыми. Выбор зависимостей между программными элементами, которые могут учитываться при построении архитектуры программной системы, заслуживает отдельного рассмотрения [19, 20, 21, 22, 23].

В данной статье описывается применение для визуализации зависимостей между элементами программы матрицы смежности. В отличии от визуализации зависимостей с помощью узлов и ребер графа, при использовании матрицы для сложных систем с большим числом зависимостей, не возникает проблем пересечения ребер затрудняющих анализ зависимостей. Пример изображения такой матрицы приведен на рисунке 1. Программные элементы представляются как строки и колонки матрицы смежности. На пересечении строки A и колонки B расположены ячейки представляющие зависимость программных элементов A и B. Порядок программных элементов в строках и колонках может быть различным. Целью декомпозиции программной системы может быть стремление добиться для программной системы высокой связанности (cohesion) и низкого сцепления (coupling) [24]. Программные элементы (например, классы, интерфейсы) программной системы при декомпозиции разделяются на кластеры (например, пакеты), а сам алгоритм декомпозиции называется кластеризацией. Программные элементы внутри полученного кластера должны иметь множество зависимостей (высокую связанность). В тоже время границы кластеров должно пересекать небольшое число зависимостей (низкое сцепление). Таким образом, декомпозиция программной системы тесно связана со структурой зависимости программных элементов. Соединив вместе на одной диаграмме и декомпозицию системы, и структуру зависимостей системы, можно будет ответить на вопросы о причинах принадлежности программного элемента к кластеру, о величине связанности кластера и сцепления между кластерами. Если декомпозиция для программных элементов создана программно, то можно оценить, как зависимости повлияли на такую кластеризацию. При оценке декомпозиции уже существующей системы, можно определить наличие каких зависимостей между элементами системы ухудшает характеристики сцепления и связанности.

Вторая задача - связывание программной декомпозиции со структурой зависимости. Часто бывает необходимо сравнить две различные декомпозиции программной системы. Например, сравнить декомпозиции двух версий той же системы. При этом интересно бывает найти как совпадающие, так и различные кластеры двух архитектур. Важно также задать для декомпозиций уровень, на котором происходит сравнение архитектур. При удалении из рассмотрения излишней детализации (кластеров нижнего уровня), возможно получение интересующего пользователя инструмента результатов. Инструмент должен позволять пользователю выбирать такой уровень детализации.

Третья задача - визуализация результатов сравнения архитектур, которая позволил наглядно оценить степень найденных сходств и различий.

Метод визуализации результатов сравнения для решения указанных трех задач должен позволять пользователю инструмента обратного проектирования:
• одновременно отображать графы зависимости и декомпозицию программной системы;
• показывать сходства и различия графов и декомпозиций;
• поддерживать поиск уровней детализации для различных программных декомпозиций.

На рисунке 16 показан пример предлагаемого способа визуализации результатов сравнения двух программных архитектур. Рассмотрим последовательно элементы этого визуального представления.

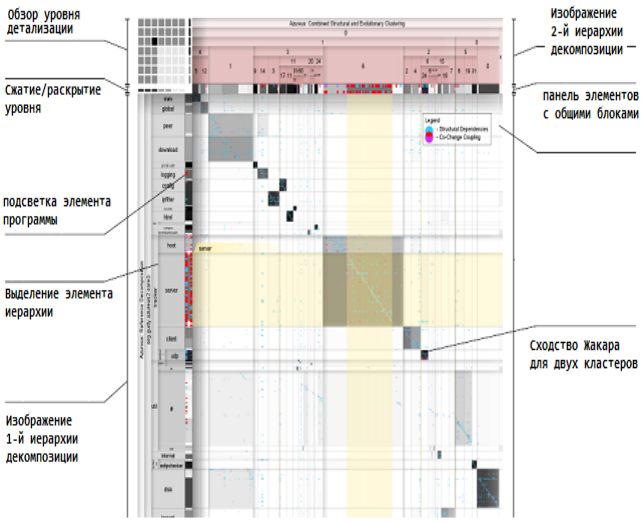

Рис.16. Матрица сравнения архитектур программных систем.

Визуализация зависимостей между элементами программной системы выполняется с помощью ячеек матрицы. Предлагаемый метод позволяет визуализировать на единой диаграмме два графа для единого множества элементов программы. Зависимости в разных графах должны быть нарисованы различными цветами для каждого графа. Третий цвет представляет на диаграмме зависимости между элементами, возникающие в обоих графах. Так, на рисунке 2, синим цветом показаны ячейки матрицы, представляющие зависимости граф структурной зависимости элементов. Фиолетовым цветом показывается зависимости для элементов программы, возникающие при их одновременном изменении в ходе эволюции программной системы.
Визуализация декомпозиции программной системы показывается как иерархия вложенных пакетов. Эта иерархия представлена в левой части матрицы. При наличии места на экране, для пакета текстом показывается имя кластера. Заданием уменьшающейся яркости для вложенных кластеров в этой иерархии выделяются кластеры (пакеты). Так элементы иерархии с большим уровнем вложенности пакетов показываются темнее. Цвет кластера используется также для задания фона, для ячеек матрицы, показывающих зависимости элементов этого кластера. Сверху от матрицы показывается вторая декомпозиция, представляющая граф второй архитектуры. Строки и колонки матрицы могут иметь различное упорядочивание.

IV. Сравнение архитектур программной системы

Задача сравнения двух декомпозиций программной системы состоит из поиска сходств и различий в структуре кластеров системы. Без поддержки CASE-инструмента это задача требует больших усилий и времени квалифицированного специалиста. Для автоматизации решения этой задачи инструментом предлагается метрика, определяющая степень сходства кластеров из различных декомпозиций. Кластер содержит множество элементов программной системы. Сравнение двух кластеров эквивалентно сравнению двух множеств A и B. Для получения метрики сходства sim(A,B) необходимо определить как отношение количества элементов, одновременно принадлежащих обоим кластерам, к размеру обоих кластеров. Это величина, известная как коэффициент Жаккара[25], вычисляется как размер пересечения двух множеств деленная на размер объединения этих множеств.

$$\mathrm{sim}(A,B) := \frac{|A \cap B|}{|A \cup B|}.$$

Коэффициент Жаккара используется для задания фона в представлении матрицы. Кластер образует среди ячеек мета структуру, похожую на подматрицу, содержащую строки и колонки для элементов входящих или не входящих в кластер. Каждое сравнение двух кластеров может быть представлено как ячейка этой матрицы. Яркость фона этой матрицы показывает величину сходства - значение коэффициента Жаккара. Темные фоны представляют значения для более схожих элементов. Такой подход позволяет сравнивать две декомпозиции на том уровне иерархии, который выбран для каждой декомпозиции отдельно. Этот уровень пользователь инструмента может выбрать взаимодействуя с диаграммой.

A. Задание на диаграмме уровня детализации

При сравнении двух декомпозиций необходимо выбрать в иерархии соответствующий уровень детализации. Особенно это важно, если уровень вложенности кластеров велик, а кластеры (пакеты) хорошо структурированы. Для поиска соответствующего уровня детализации у пользователя инструмента имеется возможность сжать кластеры. Необходимый уровень достигнут, если выполняются следующие условия:
• Условие соответствия. Декомпозиции совпадают насколько это возможно по метрике схожести.
• Сохранение структуры. Структура обоих декомпозиций сохраняется. Не должно быть слишком много сжатых кластеров.

Сжимая кластеры всегда можно добиться выполнения условия соответствия. Однако это может привести к нарушению условия сохранения структуры. Поэтому необходимо добиваться баланса этих условий. На рисунке 17 приведен пример достижения необходимого уровня детализации. Слева показан умалчиваемый уровень детализации, на котором показано слишком много различий и сходств двух декомпозиций. Справа

показано более читаемое изображение с правильно выбранным уровнем детализации.

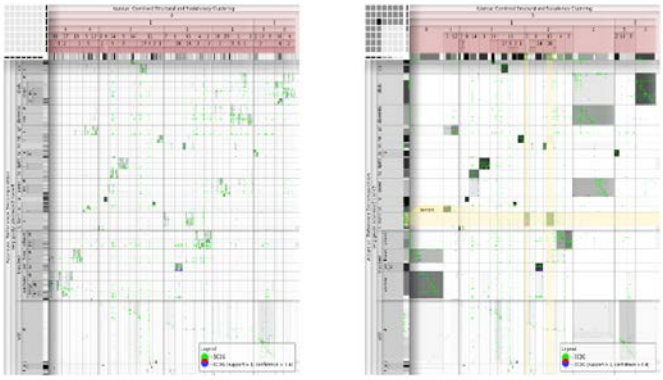

Рис.17. Выбор уровня детализации.

V. Сравнение архитектур программной системы

Для анализа и сравнения архитектур у пользователя инструмента имеется возможность взаимодействия с диаграммой. При этом используется следующая метафора взаимодействия: сначала обзор, затем масштабирование, потом фильтрация и, наконец, детализация.

Матрица по умолчанию показывает все множество данных без необходимости листания изображения. В любой момент времени умалчиваемый вид предоставляет обзор матрицы в левом верхнем углу диаграммы. Однако, метрика сходства Жаккара, значения которой показываются цветом фона, позволяют сравнивать кластеры только на самом нижнем уровне иерархии. Для сравнения кластеров высокого уровня возможно сжатие/раскрытие кластеров нижнего уровня. Сжатый кластер не меняется свой размер, но вложенные в него кластеры временно исчезают. Сжатый кластер непосредственно содержит все программные элементы – листья, содержащиеся во вложенных в него кластерах. Значение метрики сходства Жаккара показывает яркость серого фона для матрицы сжатого кластера. В левом верхнем углу диаграммы, как показано на рисунке 18, присутствуют небольшие тонкие маркеры на стороне каждой иерархии. Эти маркеры позволяют сжать/раскрыть все уровни иерархии. Эти маркеры также показывают, что данный уровень в иерархии полностью сжат (светло-серый), частично раскрыт (серый) или полностью раскрыт (темно-серый).

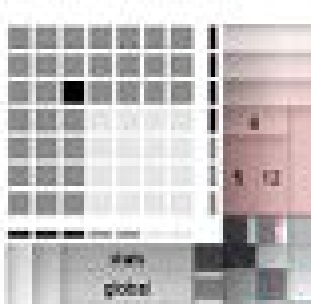

Рис.18. Выбор уровня детализации.

Для масштабирования изображения диаграммы используется колесо мыши. Увеличение и уменьшение масштаба изображения выполняется для кластера находящегося под курсором мыши. Такое масштабирование возможно для строк матрицы, для колонок матрицы, а также для строк и колонок одновременно.

Три цвета для ячеек матрицы представляют ребра графа. Для больших матриц ячейки могут быть представлены несколькими пикселями. Имеется возможность задавать фильтры, показывая ячейки только для заданного цвета и выключая изображения ячеек других цветов.

При перемещении мыши через расцвеченную ячейку матрицы, представляющую зависимость между элементами программы, во всплывающей подсказке показываются имена зависимых элементов. При этом также желтым фоном подсвечиваются строки и столбцы в изображении декомпозиций системы, как показано на рисунке 17.

Рассмотрим далее применение инструмента для решения трех указанных задач: сравнения зависимостей, связи декомпозиции и зависимости, и наконец, сравнение декомпозиций.

VI. Сравнение зависимостей программных элементов.

Графы зависимости являются основой для процесса кластеризации. Для формирования графа структурной зависимости классов (SCDG) будут использоваться отношения наследования, включения и использования, существующие между классами и интерфейсами программной системы. Также будет использоваться граф эволюционного сцепления классов (ECDG), который показывает скрытые зависимости в программной системе. Эволюционное сцепление связывает два класса системы, если эти два класса изменяются одновременно в истории проекта разработки системы. На первой фазе инструмент используется для решения задачи сравнения графов. В качестве декомпозиции системы будет использоваться умалчиваемая декомпозиция - структура пакетов в проекте JFtp. Как видно на рисунке 19, между анализируемыми графами существует большое различие, поскольку количество ячеек матрицы красного цвета не велико.

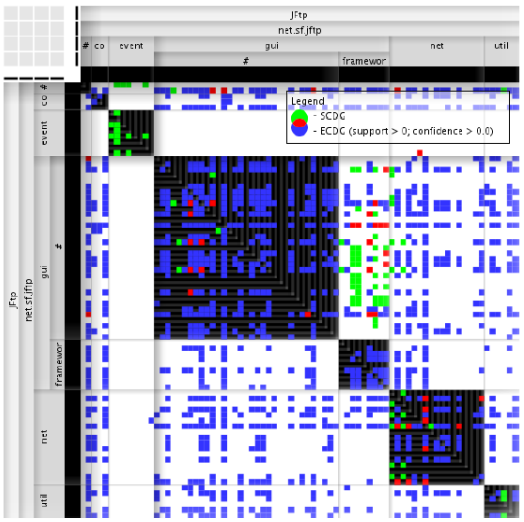

Рис.19. Сравнение графов SCDG и ECDG проекта JFtp с умалчиваемой декомпозицией - структурой пакетов.

Граф SCDG (зеленые и красные зависимости). Этот граф разреженный, но с зависимостями, попадающими в большинство кластеров-пакетов. Есть множество кластеров и с входящими, и с выходящими зависимостями.

Граф ECDG (синие и красные зависимости). В целом этот граф плотный, но для некоторых кластеров граф очень разреженный.

Также можно заметить, что пересечение зависимостей (красные ячейки) обоих графов мало и в основном относится к классам из того же пакета. Это объясняет, почему эти зависимости имеют большее значение в процессе кластеризации.

Вторая стадия анализа рассматривает декомпозиции предложенные используемым подходом к кластеризации. По вертикальной оси расположена декомпозиция на основе структурных зависимостей кода (SCDG). По горизонтальной оси расположена декомпозиция основанная на зависимости совместно эволюционирующих классов (ECDG). Визуализация таких зависимостей для проекта JFtp показана на рисунке 20.

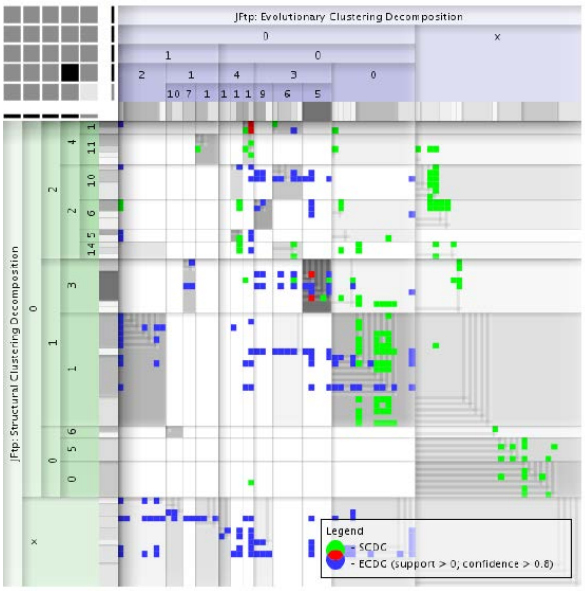

Рис.20. Сравнение графов SCDG и ECDG проекта JFtp с умалчиваемой декомпозицией - структурой пакетов.

В декомпозиции основанной на эволюции системы (показана наверху диаграммы) присутствует кластер X занимающий треть этой иерархии и не разделенный на какие либо части. Существует также кластер X в декомпозиции основанной на структуре системы (показан на диаграмме слева). В этом кластере расположены все элементы, для которых нет информации об их зависимостях. Например, потому что нет информации об эволюции для элементов из библиотек используемых анализируемой системой.

Анализируя более детально иерархии декомпозиций и графы зависимостей, можно увидеть на диаграмме, что глубокие иерархии в декомпозиции сопровождаются ясно идентифицируемыми визуальными кластерами и в графах зависимости. Алгоритм кластеризации порождает неглубокие иерархии тогда, когда структурирование по пакетам классов неочевидна.

В общем можно сделать вывод, соответствие между структурной и эволюционной декомпозициями программной системы невелико. Это может означать, что оба источника данных описывают различные размерности зависимостей.

Возможность визуального обнаружения хорошо кластеризованых пакетов может быть использована для идентификации тех пакетов, которые полностью совпадают, или полностью не совпадают в разных декомпозициях. В большинстве случаев, почти полностью совпадающие кластеры двух композиций обладают высокой структурной связанностью. Существует много ссылок между классами такого кластеров. Наличие хорошей эволюционной связанности у классов кластеров означает, что связанные структурно классы часто менялись вместе.

II. Заключение

В статье рассмотрены методы обратного проектирования позволяющие восстановить архитектуру программной системы из ее кода, а также визуализировать и анализировать эту архитектуру. Предложен метод позволяющий сравнивать различные архитектуры программной системы, основанные на различных способах декомпозиции программной системы и зависимостях между элементами этой системы. В частности, позволяющий сравнивать архитектуры для различных версий системы. В статье показано как кластеризация программного обеспечения может быть применена на практике. Рассмотренные методы применяются в инструменте обратного проектирования основанного на языке моделирования UML и реализованного как расширение среды Eclipse. Статья является продолжением цикла публикаций по программной инженерии и применению языка моделирования UML, начатой в журнале INJOIT работами [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].

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] Романов В. Ю. Визуализация и анализ больших программных систем с помощью их трехмерного представления //International

Journal of Open Information Technologies. – 2014. – Т. 2. – №. 5. – С. 1-9.
[8] Романов В.Ю. Использование шаблонов пакетов для анализа архитектуры программной системы//International Journal of Open Information Technologies. – 2014. – Т. 2. – №. 4. – С. 18-24.
[9] Романов В.Ю. Анализ и визуализация зависимостей между пакетами программных систем //International Journal of Open Information Technologies. – 2015. – Т. 3. – №. 1. – С. 23-29.
[10] Романов В.Ю. Анализ и визуализация эволюции программного обеспечения //International Journal of Open Information Technologies. – 2016. – Т. 4. – №. 9. – С. 64-73.
[11] Source Forge https://sourceforge.net/
[12] Maven Central. http://mvnrepository.com/
[13] J.N. Warfield. Binary Matrices in System Modeling. IEEE Transactions on Systems, Man, and Cybernetics, vol. 3, no. 5, pages 441–449, 1973.
[14] T.R. Browning. Applying the design structure matrix to system decomposition and integration problems: a review and new directions. IEEE Transactions on Engineering Management, vol. 48, no. 3, pages 292–306, 2001.
[15] Design Structure Matrix, http://www.dsmweb.org
[16] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. SIAM J.Comput., vol. 1, no. 2, pages 146–160, 1972.
[17] Sangal, N., E. Jordan, V. Sinha and D. Jackson, Using Dependency Models to Manage Complex Software Architecture, forthcoming in Proceedings of the 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, San Diego, 2005.
[18] Y. Ghanam and S. Carpendale. A survey paper on software architecture visualization. Technical report, 2008.
[19] M. Bauer and M. Trifu. Architecture-aware adaptive clustering of OO systems. In CSMR '04: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering, Washington, DC, USA, 2004. IEEE Computer Society.
[20] V. Tzerpos and R. C. Holt. MoJo: A distance metric for software clusterings. In WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering, pages 187-193, Washington, DC, USA, 1999. IEEE Computer Society.
[21] V. Tzerpos and R. C. Holt. ACDC: An algorithm for comprehension-driven clustering. In WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering, pages 258-267. IEEE Computer Society, 2000.
[22] Z.Wen and V. Tzerpos. Evaluating similarity measures for software decompositions. In ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance, pages 368-377,Washington, DC, USA, 2004. IEEE Computer Society.
[23] A. Wierda, E. Dortmans, and L. L. Somers. Using version information in architectural clustering - a case study. In CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering, pages 214-228, Washington, DC, USA, 2006. IEEE Computer Society.
[24] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. IBM Systems Journal, 13(2):115-139, 1974.
[25] Елисеева И. И., Рукавишников В. О. Группировка, корреляция, распознавание образов: (статистические методы классификации и измерения связей). — М.: Статистика, 1977. — 143 с.

Visualization of software system architecture and its evolution

Romanov V.Y.

***Abstract* — Reverse engineering CASE-tools can produce different descriptions of software architectures. The article analyzes and defines the task of exploring and comparing software architectures presented as design structure matrix. The visualization method to compare architectures based on the decomposition of the software system and on the dependencies of program elements. In particular, to compare the architectures for the different versions of the system. The paper shows how the clustering software can be applied in practice.**

Keywords — software architecture, software decomposition, software elements dependencies, software clustering.