Визуализация внутренней структуры и зависимостей пакетов в программной системе

Романов В.Ю.

Аннотация. **В статье рассматриваются методы визуализации архитектуры программной системы в составе инструмента обратного проектирования и восстановления архитектуры программной системы. Рассматриваются методы визуализации и анализа зависимости пакетов программной системы, написанной на языке Java. Для этого используется матричное представление графа, описывающего связи между классами внутри анализируемого пакета, а также связи между классами разных пакетов. Описанные в статье возможности инструмента предоставляют инфраструктуру для последующего обнаружения и исправления ошибок проектирования программных систем, а также для рефакторинга программной системы.**

Ключевые слова **— software visualization, reengineering, package matrix, package, dependency, architecture recovery, reverse engineering.**

I. Введение

Задача обратного проектирования (reverse engineering) программной системы является весьма важной при разработке программной системы с использованием библиотек с исходными кодами. Построение и визуализация UML-модели для вновь разрабатываемой программной системы, так и для используемых ею библиотек существенно упрощает понимание их структуры и функциональности, выбор необходимой версии и разработчика 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 лишь для наиболее существенной ее части - архитектуры. Для построенной UML-модели 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] рассмотрен анализ архитектуры программной системы на основе шаблонов пакетов языка Java. В работах [4,5] рассматривались визуализация архитектуры и объектно-ориентированных метрик с помощью уже хорошо известных двумерных изображений. Возможности использования для визуализации программных систем трехмерных изображений рассмотрены в статье [8]. Визуализация архитектуры программной системы с помощью матриц структурной зависимости рассмотрена в статье [9].

В данной статье рассматривается визуализация с помощью матриц входящих и исходящих зависимостей пакетов, позволяющая проанализировать существующие зависимости как между классами внутри пакета, так и между классами разных пакетов. Получение такой информации позволяет понять причину возникновения зависимостей между пакетами, определяющими архитектуру системы, а также выполнить при необходимости рефакторинг системы.

II. Проблемы в понимании структуры и зависимостей пакетов

На структурирование сложных программных систем по пакетам могут повлиять самые разные факторы. Пакеты могут определять модули кода системы, которые будут использоваться для распространения системы. Пакеты могут отражать собственность на программный код, полученный от внешних разработчиков. Пакеты могут отражать организационную структуру команды, разрабатывавшей систему, архитектуру системы или разбиение системы на уровни. Вместе с тем, правильное структурирование системы должно минимизировать зависимости между пакетами.

Ошибки проектирования структуры пакетов системы зачастую сказываются также и при работе системы. Рекурсивная зависимость пакета от других пакетов требует загрузки кода этих пакетов в память на устройствах с ограниченными ресурсами. Для решения этой проблемы бывает необходимо реструктурирование пакетов. При этом необходимо выявление в пакете классов, имеющих максимальное количество входящих и исходящих связей с классами других пакетов. Возможное перемещение класса во внешний пакет минимизирует зависимость между пакетами.

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

Не менее важен также анализ связей между классами расположенными внутри пакетов. Минимизация зависимостей между классами больших пакетов позволит реструктурировать пакет, разбив его на несколько пакетов меньшего размера. Общее количество зависимостей между пакетами системы при этом может уменьшиться. Для решения задач реструктурирования пакетов необходима специальная визуализации пакетов, позволяющая детально проанализировать зависимости между пакетами и классами пакетов. В предшествующей работе [9] были рассмотрены анализ и визуализация зависимостей пакетов с помощью матриц структуры зависимостей (design structure matrix) в инструменте обратного проектирования. Данный метод анализа пакетов системы позволяет упростить структурирование системы на уровни, упрощая извлечение архитектуры системы. Для анализа и визуализации зависимостей между парами пакетов системы использовалась детальная визуализация ячеек матрицы зависимостей [9], показывающая связи между парой пакетов, представляемых этой ячейкой. Вместе с тем, зачастую информации в ячейках матрицы структурной зависимости бывает недостаточно. Для удаления циклических зависимостей между пакетами, уменьшения количества связей между пакетами системы могут быть полезны матрицы, показывающие причины зависимости пакета со всеми другими пакетами системы.

Для понимания взаимосвязей пакетов существенную помощь может оказать визуализация метрик пакетов, как это было сделано в работах [10]. Визуализации вложенности пакетов и влияния такой вложенности на архитектуру программной системы рассмотрены в работе [11]. Визуализация и анализ сцепления пакетов, а также совместное использование пакетов классами рассмотрены в работах [12, 13]. Анализ архитектуры программной системы с помощью матриц структурной зависимости рассмотрен в работах [9, 14].

III. Визуализация для понимания роли пакета в системе

A. Выбор способов визуализации пакета

Хотя визуализация всех связей пакета, возможно, должна будет показать очень большое количество связанной с пакетом информации, тем не менее, она должна упростить анализ пакета. Для визуализации графов наибольшее распространение получили визуализация в виде узлов и ребер между узлами, а также визуализация в виде матриц. Как было отмечено в работе [15], представление в виде узлов и ребер является проще читаемым и интуитивно понимаемым при небольшом числе узлов и ребер у графа. Однако матричное представление не имеет проблем, связанных с пересечением ребер графа и наложением узлов графа при большом числе связей между узами. Поэтому матричное представление более подходит для визуализации сложных графов.

B. Основные принципы визуализации пакета

Для детальной визуализации пакета независимо от сложности графа зависимостей пакета предлагается использовать матричное представление графа. Пакет представляется прямоугольником, стороны которого образуют контактные области называемые *поверхностями*. Каждая строка/столбец представляет внутренний класс рассматриваемого пакета или класс внешнего пакета, с которым взаимодействует внутренний класс рассматриваемого пакета. У поверхности есть *заголовок*, представляющий отношения между внутренними классами рассматриваемого пакета, и тело, представляющее взаимодействие внутренних классов рассматриваемого пакета с внешними классами. Для представления входящих и исходящих зависимостей пакета используются отдельные виды пакета.

На рисунке 1 показана матрица для чтения выходящих зависимости пакета. Эта матрица показывает, от каких классов рассматриваемого пакета и классов других пакетов зависят классы рассматриваемого пакета. Строки матрицы используются для визуализации как внутренних классов, так и классов от которых зависят классы рассматриваемого пакета. Колонки матрицы используются для визуализации классов рассматриваемого пакета.

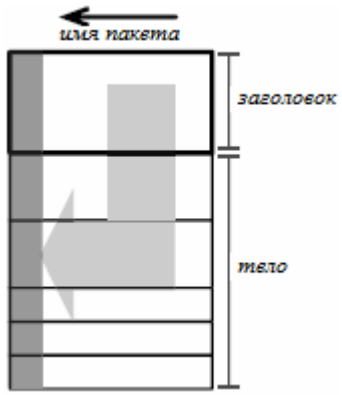

Рис.1. Чтение выходящих зависимостей пакета.

На рисунке 2 показаны входящие зависимости пакета, показывающие какие классы из других пакетов, и из рассматриваемого пакета, зависят от классов рассматриваемого пакета. Колонки матрицы используются для визуализации как внутренних классов, так и классов которые зависят от классов рассматриваемого пакета. Строки матрицы используются для визуализации классов рассматриваемого пакета.

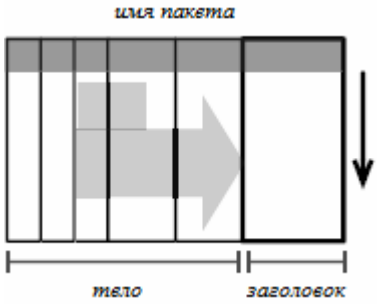

Рис.2. Чтение входящих зависимостей пакета

Как можно заметить, заголовок матрицы пакета представляет собой матрицу структурной зависимости классов пакета, схожей с матрицей структурной зависимости пакетов рассмотренной в работе [9].

Рассмотрим матрицу зависимости пакета более детально. На рисунке 3 приводится пример визуализации пакетов и их зависимостей с помощью узлов и ребер. Пакет P1, показанный на этом рисунке, будет затем представлен с помощью матриц входящих и исходящих зависимостей этого пакета.

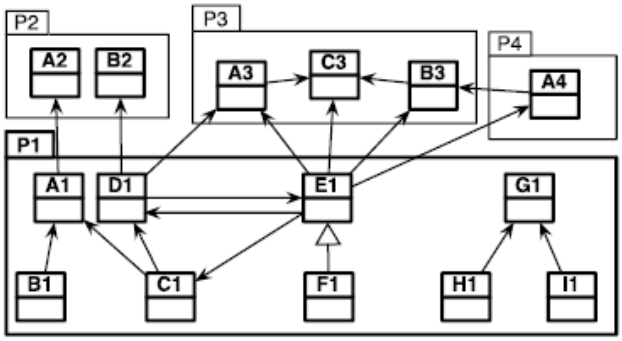

Рис.3. Визуализация зависимостей пакетов в виде графа.

Так, на рисунке 4, показана матрица выходящих зависимостей для пакета P1.

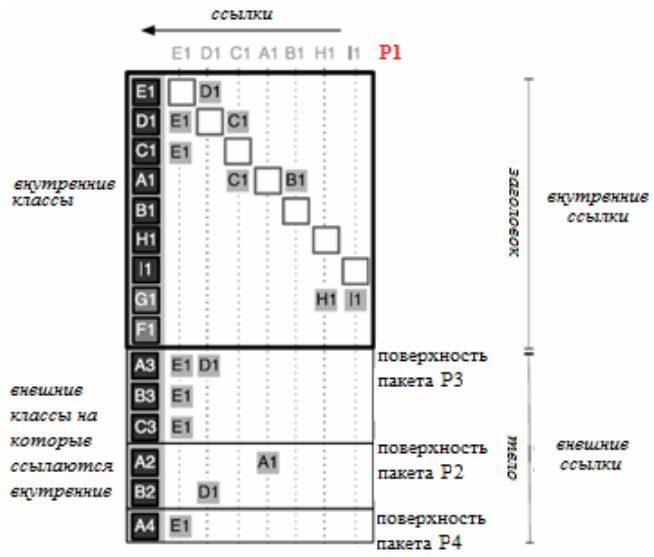

Рис.4. Визуализация выходящих зависимостей пакета P1

Строки матрицы на рисунке 4 представляют классы пакета *P1*, а также классы, от которых зависят классы пакета *P1*. Эти же классы показаны в колонках матрицы. Колонки матрицы показывают классы, ссылающиеся на другие классы рассматриваемого пакета. В случае матрицы визуализации выходящих зависимостей, показанной на рисунке 4, это только классы пакета *P1*. Заполненные ячейки представляют ссылки класса, показанного в колонке, на класс, показанный в строке. Группа строк матрицы, относящиеся к одному пакету, образуют поверхность пакета. Первая поверхность, относящаяся к рассматриваемому пакету, есть заголовок пакета.

Класс E1 ссылается на внутренние классы C1 и D1. На классы B1, H1, I1 и F1 никакие классы пакета P1 не ссылаются, поскольку нет заполненных ячеек в соответствующих этим классам строках. На классы из поверхности пакета P3 есть ссылки из рассматриваемого пакета P1. Это классы A3, B3 и C3. Поверхность пакета P3 расположена в матрице выше поверхностей пакетов P2 и P4, поскольку включает больше классов, чем поверхности пакетов P2 и P4.

Для упорядочивания колонок, поверхностей и строк в поверхностях используется единое правило. Ближе к заголовку (выше) располагаются поверхности пакетов, на которые имеется больше всего ссылок. Внутри поверхности ближе к заголовку (выше) располагаются те классы, на которые больше всего ссылок из классов рассматриваемого пакета.

Яркость цвета фона для имени класса, заданная для ссылающегося класса, показывает, сколько ссылок идет из ссылающегося класса в ячейке колонки, в пакет представленный поверхностью ячейки. Темная ячейка имеет больше ссылок. И горизонтальная позиция класса, и его яркость представляют количество ссылок. Однако позиция показывает количество ссылок для всей матрицы, а яркость для конкретной поверхности матрицы.

Для разделения представленных в матрице классов на категории может использоваться цвет класса. В заголовке матрицы цвет может использоваться для разделения классов рассматриваемого пакета на классы, имеющие ссылки и не имеющие ссылки. Не имеющие ссылок классы раскрашиваются в более светлые цвета, имеющие ссылки в более темные цвета. Так, на рисунке 4 классы *G1* и *F1* более светлые, чем классы *E1* и *I1*. В теле матрицы возможно выделение цветом пакеты и классы, не входящие в анализируемое приложение. Например, таким образом, могут быть раскрашены классы, расположенные в пакетах из библиотек, полученных от внешних разработчиков.

Рассмотрим теперь матрицу пакета, показывающую входящие зависимости пакета. Для этого используется аналогичная матрица с небольшими отличиями: поверхности матрицы входящих зависимостей располагаются горизонтально. Таким образом, будет проще различить матрицы входящих и выходящих зависимостей, если они расположены на экране рядом.

На рисунке 5 показана матрица входящих зависимостей для пакета *P3*. На рисунке 5 пакеты-клиенты пакета *P3* располагаются слева от внутренних классов пакета *P3*. Поверхности с большим числом ссылающихся классов расположены ближе к заголовку (правее). Внутри поверхности классы с большим числом ссылок также располагаются правее.

В этой матрице представлены поверхности для пакетов *P1* и *P4*, поскольку на рисунке 3 класс *E1* из пакета *P1* использует классы *A3, C3* и *B3* пакета *P3*, а класс *A4* из пакета *P4* использует класс *B3* из пакета *P3*

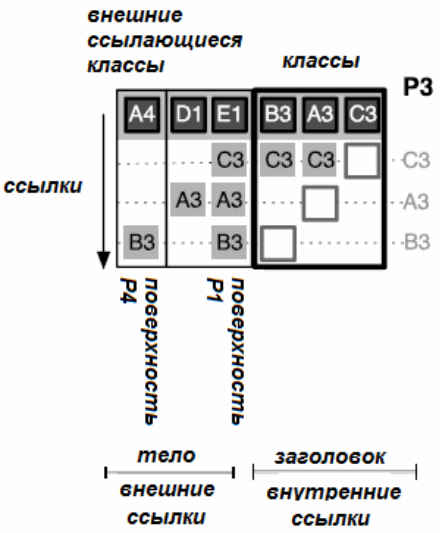

Рис.5. Визуализация входящих зависимостей для пакета P3

IV. Анализ структуры пакета с помощью матрицы пакета

Проиллюстрируем теперь использование матрицы пакета для изучения структуры и зависимостей пакета.

A. Взаимодействие пользователя CASE-инструмента с матрицей пакета

Для анализа структуры пакета с помощью матрицы пакета могут быть выполнены выбор или маркировка классов или пакетов (поверхностей представляющих пакет). При выборе класса узлы класса и связанные с ним ссылки окрашиваются в красный цвет. То же самое происходит, когда класс помечается заданным цветом по желанию пользователя инструмента. Выбор/маркировка поверхности означает, что аналогично выбираются/маркируются все отношения, в которые вступает пакет представленный данной поверхностью. На рисунке 6 показана матрица выходящих зависимостей пакета *Protocols* с выбранным в матрице классом *HTTPSocket*.

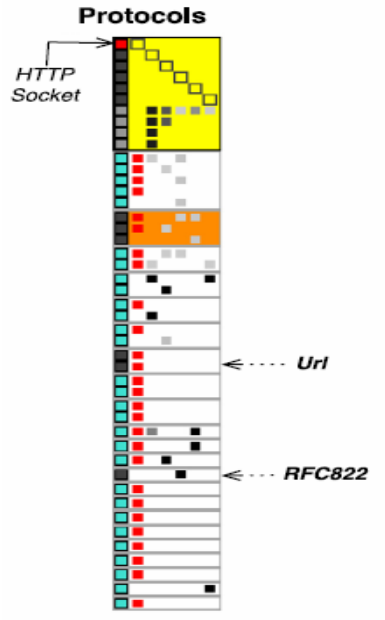

Рис.6. Визуализация выбранного класса *HTTPSocket* в матрице пакета *Protocols*

Красным цветом на рисунке 6 показаны как сам класс *HTTPSocket* в первой строке первой колонки, так и классы на которые он ссылается (вторая колонка матрицы выходящих зависимостей).

Пример маркировки классов в матрице выходящих зависимостей пакета *NetworkKernel* приведен на рисунке 7. Синим цветом на рисунке 7 помечен класс *SocksSocket*, зеленым цветом - класс *InternetConfiguration*, а малиновым цветом - класс *Password*. Также на рисунках 6 и 7 были маркированы поверхности пакетов. Оранжевым цветом была промаркирована поверхность пакета *NetworkKernel*, а желтым цветом промаркирована поверхность пакета *Protocols*. Голубым цветом помечены классы, которые не принадлежат анализируемому приложению (классы из внешних библиотек).

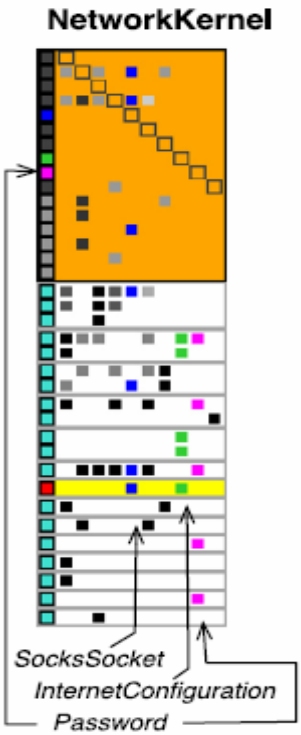

Рис.7. Пример маркировки классов в матрице выходящих зависимостей пакета NetworkKernel

Классы и поверхности пакетов представлены в матрицах зависимостей в компактном виде. Более подробная о классе или пакете появляется как всплывающая подсказка, как показано на рисунке 8.

Пользователь инструмента может выполнить фильтрацию информации отображаемой матрицей пакета. Возможно отображение ссылок относящихся только к анализируемому приложению, либо к описанной группе пакетов. После исключения всех классов используемых библиотек матрица входящих зависимостей пакета *Protocols* принимает компактный вид, как показано на рисунке 9.

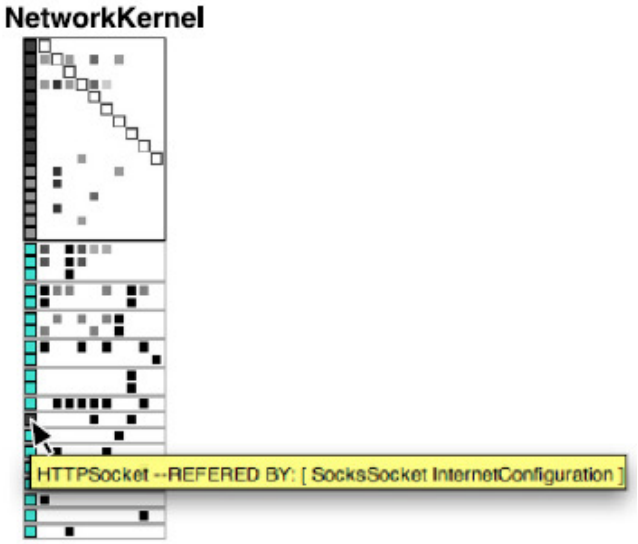

Рис.8. Всплывающая подсказка для класса *HTTPSocket* в матрице выходящих зависимостей пакета *NetworkKernel*

Пользователь может также с помощью фильтров убрать классы, не имеющие отношения зависимости с классами внешних пакетов. Или скрыть заголовок матрицы пакета, сосредоточившись на анализе лишь зависимостей между классами разных пакетов

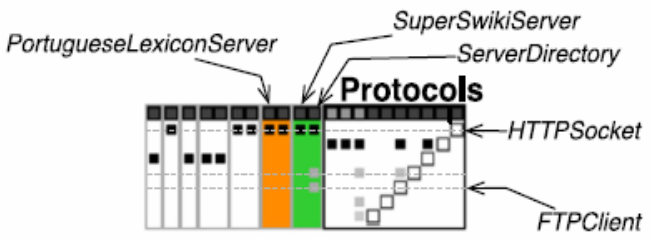

Рис.9. Матрица входящих зависимостей пакета Protocols после фильтрации классов.

B. Анализ пакета с помощью матрицы выходящих зависимостей пакета

Рассмотрим, как матрица выходящих зависимостей может быть использована для анализа пакета. Такой беглый взгляд помощью матрицы на "черновик" пакета позволяет оценить с особенности реализации пакета.

Анализ больших пакетов.

Рассмотрим пакеты, имеющие большой размер матриц выходящих зависимостей. Причины этого могут быть различны. На рисунке 10 представлены три пакета с большой матрицей пакета.

Пакет *HTMLParserEntities* имеет большое число собственных классов, поскольку имеет большой заголовок. С другой стороны, пакеты *RemoteDirectory* и *Protocols* имеют большую матрицу, поскольку содержат большое число ссылок на классы в других пакетах (большое тело) при относительно небольшом заголовке матрицы. Большое число поверхностей матрицы характеризует тесно связанные (coupled) пакеты. Таким образом, два последних пакета имеют сильную связанность со своим внешним окружением.

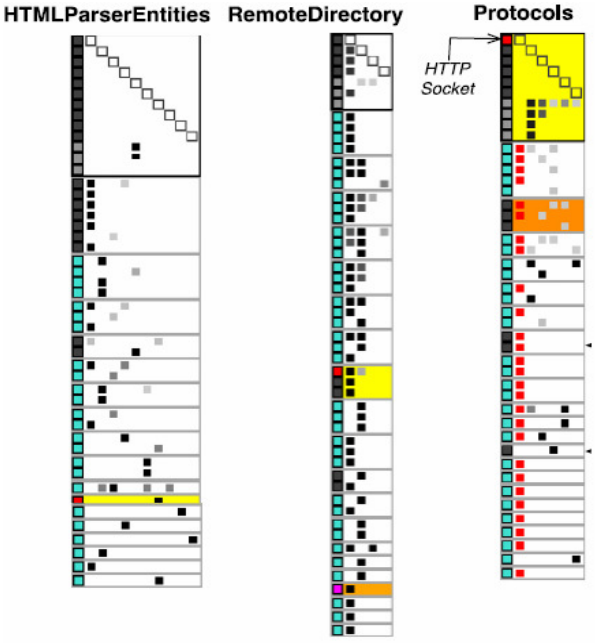

Рис.10. Пакеты с большой матрицей пакетов.

Небольшие пакеты со сложной реализацией.

Класс *TelNetWordNet*, показанный на рисунке 11, имеет всего четыре собственных класса.

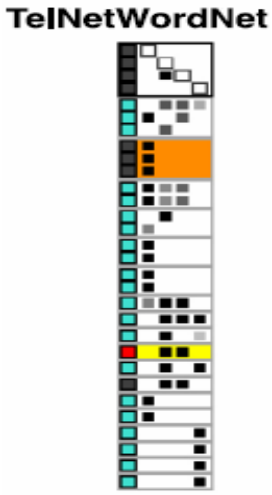

Рис.11. TelNetWordNet - небольшой пакет со сложной реализацией.

Вместе тем, пакет имеет большое тело матрицы и большое число поверхностей матрицы. Исходя из этого, можно сделать выводы, что загрузка в память этого небольшого пакета потребует также загрузки кода большого числа других пакетов. Это возможно приведет к проблемам на устройствах с небольшой памятью.

Класс *RemoteDirectory* также имеет небольшое число классов. Однако его реализация намного сложнее реализации класса *TelNetWordNet*, поскольку зависимости его классов распределены среди большего числа внешних классов и поверхностей.

Разреженные пакеты.

Пакет *HtmlParserEntities* на рисунке 10 и пакет *TelNetWordNet* на рисунке 11 имеют разреженные

заголовки. Это означает, что сцепление между классами внутри этих пакетов невелико. Возможно, что это кандидаты на декомпозицию этих пакетов (распределение классов пакета по другим пакетам). Вместе с тем, можно заметить, что пакет *HtmlParserEntities* имеет не только разреженный заголовок, но и разреженное тело. По этой причине его декомпозиция более вероятна.

Пакеты с внутренним сцеплением (cohesion).
Пакет URL, показанный на рисунке 12, имеет в заголовке матрицы выходящих зависимостей большое число заполненных узлов.

Однако из рисунка 12 также видно, что пакет URL имеет в теле множество ссылок на внешние пакеты. Здесь более существенно сцепление с классами внешних пакетов, чем с классами внутри пакетов.

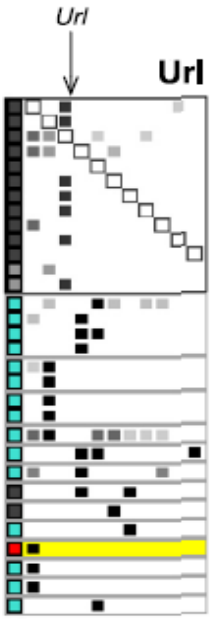

Рис.12. URL - пакет с сильным внутренним сцеплением.

Неправильный выбор положения класса.
С помощью матрицы выходящих зависимостей можно легко обнаружить классы, для которых неудачно выбран содержащий их пакет. Так показанный малиновым прямоугольником на рисунке 13 класс *Password* не имеет ни входящих, не исходящих зависимостей внутри заголовка матрицы *NetworkKernel*.

Таким образом, выявленный класс становится кандидатом на перемещение его в пакеты, с классами которых он такие зависимости имеет. Перемещение класса в использующий пакет увеличит сцепленость (*cohession*) обоих пакетов

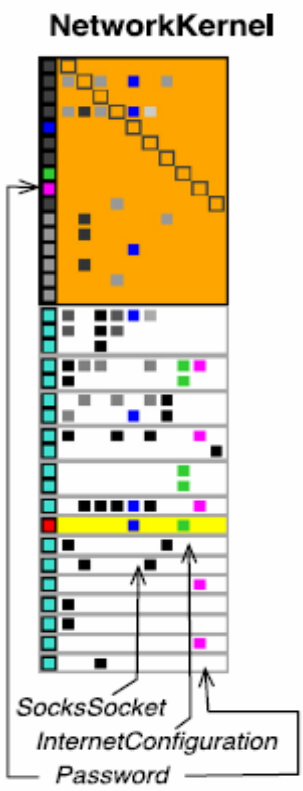

Рис.13. Неправильный выбор положения класса *Password* в пакете *NetworkKernel*

C. Анализ пакета с помощью матрицы входящих зависимостей

Матрица входящих зависимостей показывает, как пакет используется другими пакетами приложения. При анализе пакетов с помощью таких матриц можно выявить, например, шаблоны пакетов.

Пакеты-листья и изолированные пакеты.
На рисунке 14 показан пакет-лист *MailReaderFilters*, на который ссылается только один пакет *MailReader*.

Рис.14. Пакеты-листья и полностью изолированные пакеты.

Также с помощью матрицы входящих зависимостей пакета можно легко выявить в системе и такие полностью изолированные пакеты, как показанный на рисунке 14 пакет *SqueakPage*.

Наиболее интенсивно используемые классы пакета.
Для иллюстрации тесно связанных (coupled) пакетов рассмотрим матрицу входящих зависимостей пакета *Kernel*, показанную на рисунке 15. Классами, на которые имеется наибольшее количество ссылок, являются классы, которые имеют в теле пакеты с большими поверхностями, как *Socket* и *NetNameResolver* расположенные в двух верхних строках матрицы. Однако строка для класса *NetNameResolver* темнее, чем строка класса Socket. Это означает, что класс *NetNameResolver* имеет больше внутренних входящих

зависимостей, чем класс *Socket*. А класс *Socket* имеет больше входящих внешних зависимостей, поскольку яркость представляется количество ссылок на классы пакета в теле пакета.

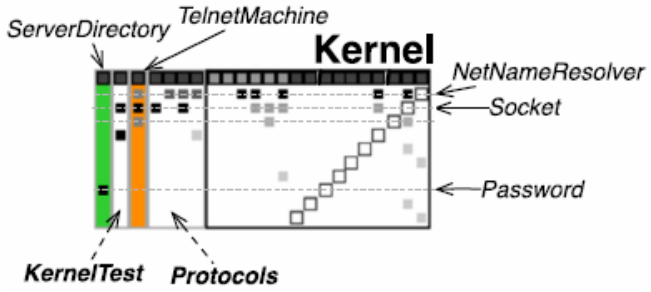

Рис.15. Интенсивно используемые классы пакета..

Тесно связанные пакеты.

Для оценки влияния изменения в одном пакете на другой пакет часто бывает необходимо выявить в системе тесно связанные пакеты. Признаком тесной связанности пакетов является большая поверхность пакета в теле рассматриваемого пакета, близко расположенного к заголовку пакета. Пример тесно связанных пакетов показан на рисунке 14 с матрицей входящих зависимостей для пакета *MailReaderFilter*. Тесно связанным с ним пакетом, в соответствии с приведенными критериями связанности, является пакет *MailReader*. Другой пример тесной связанности пакетов показан на рисунке 15. Пакет *Protocol*s тесно связан с пакетом *Kernel*. Изменения в пакете *Kernel* существенно повлияют на классы в пакет Protocols.

Пакеты ядра.

При анализе программной системы важно выявить пакеты определяющие ядро системы. Это пакеты, от которых зависит большинство других пакетов системы. На рисунке 16 показаны два пакета *Url* и *Protocols*, являющиеся таким ядром.

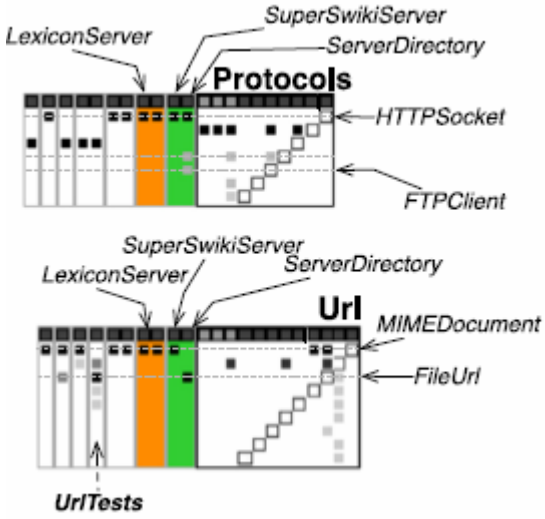

Рис.16. Пакеты Url и Protocols из ядра системы.

Выявить пакеты ядра системы можно по наибольшему количеству поверхностей в теле матриц входящих зависимостей для этих пакетов.

V. Заключение

В статье рассмотрены вопросы анализа и визуализации зависимостей между классами в пакетах программных систем написанных на языке Java. Данная задача весьма актуальна для восстановления архитектуры программной системы при решении задачи обратного проектирования. В статье рассмотрены методы визуализации таких зависимостей с помощью матричного представление графа, описывающего связи между классами внутри анализируемого пакета, а также связи между классами разных пакетов. Показано использование такого метода визуализации зависимостей между пакетами в инструменте обратного проектирования и восстановления архитектуры. Данный инструмент основан на языке моделирования UML и реализован как расширение среды Eclipse. Статья является продолжением цикла публикаций по программной инженерии и применению языка моделирования UML, начатой в журнале INJOIT работами [1-9]. Эта работа относится к числу одного из направлений исследований в Лаборатории ОИТ факультета ВМК МГУ [16, 17].

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. Использование шаблонов пакетов для анализа архитектуры программной системы//International Journal of Open Information Technologies. – 2014. – Т. 2. – №. 4. – С. 18-24.

[8] 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.

[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] S.Ducasse, M.Lanza, L.Ponisio, Butterflies: A visual approach to characterize packages, in: Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05), IEEE Computer Society, 2005, pp.70–77.

[11] M.Lungu, M.Lanza, T.Gîrba, Package patterns for visual architecture recovery, in: Proceedings of CSMR2006 (10th European Conferenceon Software Maintenance and Reengineering), IEEE Computer Society Press, Los Alamitos, CA, 2006, pp.185–196.

[12] H.Abdeen, I.Alloui, S.Ducasse, D.Pollet, M.Suen, Package reference fingerprint: a rich and compact visualization to understand package relationships, in: Europe an Conference on Software Maintenance and Reengineering (CSMR), IEEE Computer Society Press, 2008, pp.213–222.

[13] N.Sangal, E.Jordan, V.Sinha, D.Jackson, Using dependency models to manage complex software architecture, in: Proceedings of OOPSLA'05, 2005, pp.167–176.
[14] H.Abdeen, S.Ducasse, D.Pollet, I.Alloui, Package fingerprints: A visual summary of package interface usage, Inf. Softw. Technol. 52(12) (2010) 1312–1330.
[15] M.Ghoniem, J.-D.Fekete, P.Castagliola, A comparison of the read ability of graphs using node-link and matrix-based representations, in: Proceedings of the IEEE Symposiumon Information Visualization, INFOVIS'04, IEEEComputerSociety, Washington, DC, USA, 2004, pp.17–24.
[16] Намиот Д., Сухомлин В. О проектах лаборатории ОИТ //International Journal of Open Information Technologies. – 2013. – Т. 1. – №. 5. – С. 18-21.
[17] Гурьев Д. Е., Намиот Д. Е., Шнепс М. А. О телекоммуникационных сервисах //International Journal of Open Information Technologies. – 2014. – Т. 2. – №. 4. – С. 13-17.

Visualization of software package internal structure and dependencies

Vladimir Romanov

Abstract — **This article discusses the visualization of a software system architecture as a part of the Reverse Engineering and Recovery tool. We discuss the methods of visualization and analysis for software package based system written in Java. As a main method, we use a matrix representation of the graph describing the relationship between classes within the analyzed package as well as the connections between different classes of analyzed packets. We describe the possibilities of the tool which provides a framework for the subsequent error detection and correction software design. The same tool could be used for refactoring software system.**

Keywords — **software visualization, reengineering, package matrix, package, dependency, architecture recovery, reverse engineering.**