Анализ и визуализация эволюции программного обеспечения

Романов В.Ю.

Аннотация. **В статье приводится обзор методов визуализации эволюции программных систем в составе инструмента обратного проектирования и восстановления архитектуры программной системы. Рассматриваются методы визуализации и анализа изменений элементов программы и отношений между ними в различных версиях программных систем и используемых ими библиотек. Описываются метафоры позволяющие при визуализации программного обеспечения использовать привычные для пользователя инструмента понятия их обычной жизни - метафоры растущего города, растущего дерева, "температуры" в частях программной системы описывающей интенсивность происходивших в системе изменений.**

Ключевые слова **— software visualization, reverse engineering, software evolution, visualization metaphor.**

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 зависимости визуализации между пакетами для анализа архитектуры программной системы.

В данной статье 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 [10] и Maven Central [11]. Эти библиотеки эволюционируют независимо друг от друга, а также от использующего их программного обеспечения. Для сопровождения разрабатываемой системы становится важным отслеживание такой эволюции.

Главное препятствие при анализе эволюции программной системы является резкое возрастание объема и сложности информации о различных версиях программной системы и используемых ею библиотек. В анализ должны использоваться тысячи файлов возможно десятков версий используемых библиотек.

Важным является также то, на каком уровне желательно отслеживать изменения в программной системе и используемых ею библиотек. Могут быть интересны изменения в новых версиях на уровне строк программного кода. Изменения в системе могут быть на более высоком уровне - на уровне конструкций языка программирования типов, классов, интерфейсов, пакетов. Важны изменения не только в перечисленных элементах программы, но и в отношениях между ними. Наиболее интересными и важными являются изменения в ходе эволюции в архитектуре программной системы.

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

При визуализации эволюции важно иметь возможность оценить не только изменения в программной системе, но и состояние системы между такими изменениями. Поэтому визуализацию эволюции системы нельзя рассматривать в отрыве от визуализации состояния системы.

В данной статье будет рассмотрена визуализация и анализ состояния и эволюции системы на уровне элементов программы и связей между ними. Визуализация системы на уровне элементов программы и программного кода, а также на уровне элементов программы и архитектуры систем заслуживает рассмотрения в отдельных статьях.

II. Визуализация эволюции программной системы

A. Визуализации эволюции пакетов системы

Наглядное представление о программной системе дает ее визуализация с использованием метафоры города [12]. Метафора города трактует типы программы - классы, интерфейсы и перечисления как здания в городе, а структурирующие программную систему пакеты как кварталы города. Пример такой визуализации приведен на рисунке 1.

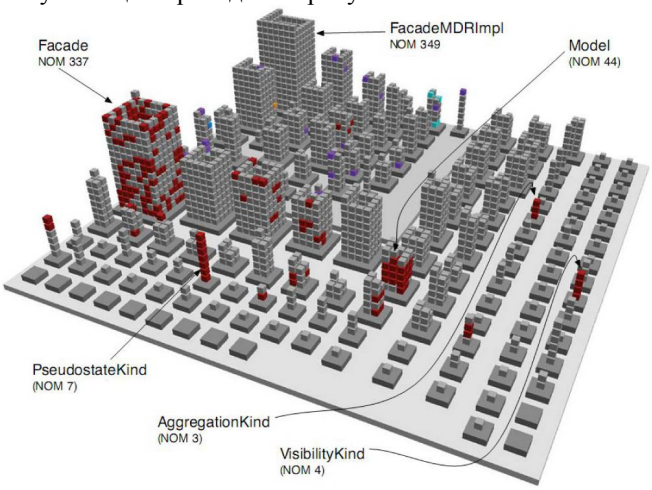

Рис.1. Метафора города для визуализации программной системы.

Кирпичами здания-класса являются методы. На рисунке 1 для некоторых классов показаны значения метрики NOM - количество методов в классе. Для изображения значений метрики может быть использован и цвет. Красным цветом на рисунке 1 показаны методы, обладающие некоторым предельно допустимым значением метрики LOC - количество строк для метода. Имена изображаемых элементов программы и их атрибуты (например, значения метрик класса) показываются во всплывающих подсказках. Использование метафоры города позволяет пользователю инструмента ориентироваться в изображении, используя уже знакомые ему в обычной жизни привычные понятия.

Следует отметить, что трехмерное изображение города наиболее наглядно для пользователя, однако его реализация существенно сложнее. Приемлемой может быть и более простая для реализации, но вместе с тем также наглядная визуализация одной из проекций города (карта города). Для получения представления об общей структуре системы-города может быть полезен вид на город сверху. Далее приводится пример использования метафоры города в системе. На рисунке 2 показана визуализация сравнения двух версий файлов библиотеки Guava: guava-10.0-rc1.jar и guava-19.0.jar

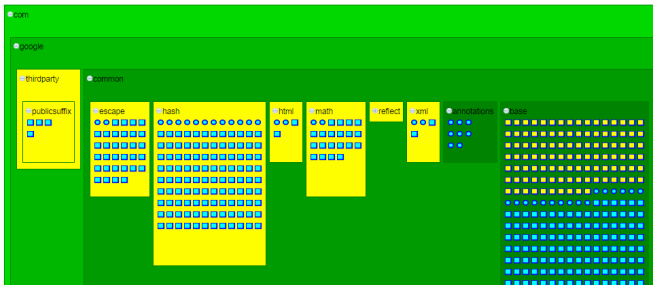

Рис.2. Сравнение двух версий библиотеки Guava.

Вместо трехмерного изображения города на рисунке 2 используется взгляд на город сверху (карта города). Зеленым цветом на рисунке показаны пакеты-кварталы, которые присутствуют в обеих сравниваемых версиях системы. Для более наглядной визуализации вложенности пакетов вложенные пакеты показываются более темным цветом. Желтым цветом показаны пакеты-кварталы и классы-здания, которые присутствуют только во второй (новой) версии системы Guava.

Метафора города может быть применена также для визуализации и анализа эволюции программной системы. Цвет зданий и кварталов города в этом случае используется для визуализации их возраста, а само изображение программной системы как города называется картой возраста системы. На рисунке 3 показана карта возраста системы ArgoUML. Самые новые здания и кварталы города-системы показаны светло желтым цветом, а самые старые темно синим.

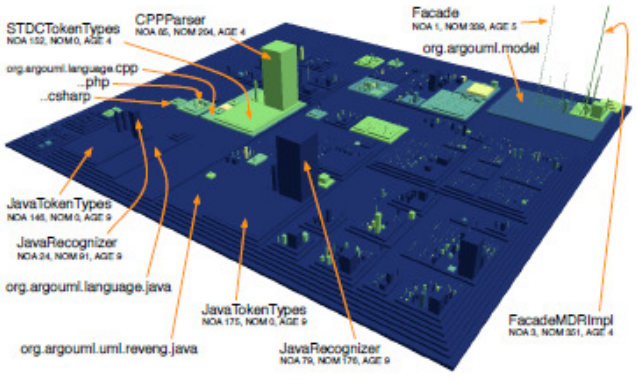

Рис.3. Карта возраста системы ArgoUML.

Визуализация растущего города-системы возможна и в режиме анимации [13]. Это позволяет получить общее представление об эволюции системы уже имеющей большое число версий.

B. Визуализации эволюции классов системы

Для визуализации эволюции программного элемента системы, например, класса, также может быть

использована метафора растущего здания города [12] [14]. В качестве "кирпичей" такого здания показываются методы класса. В изображение эволюции класса-здания вводится понятие временной шкалы, вдоль которой располагаются изображения класса разных версий. Так, на рисунке 4, показана эволюция класса C в версиях v1, v2 и v3:

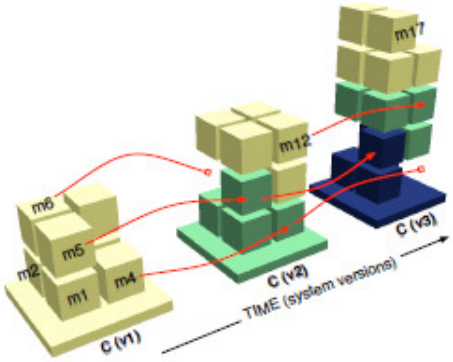

Рис.4. Изображение трех версий класса C.

Положение каждого метода в здании-классе фиксированное. В случае исчезновения метода в изображении здания появляется пустое место. Для изображения возраста метода используется цвет. При переходе на следующую версию класса окраска метода становится темнее, показывая его старение. Возраст метода - это целое число, показывающее сколько версий пережил этот метод до текущей версии. Таким образом, на изображении эволюции класса могут показаны нестабильные классы, содержащие большое количество пустот и новых методов. Также будут видны классы с резко возрастающим количеством методов. Роль таких классов в новой версии возрастает.

На рисунке 5 показана эволюция класса Graphics3D протяжении всей его жизни.

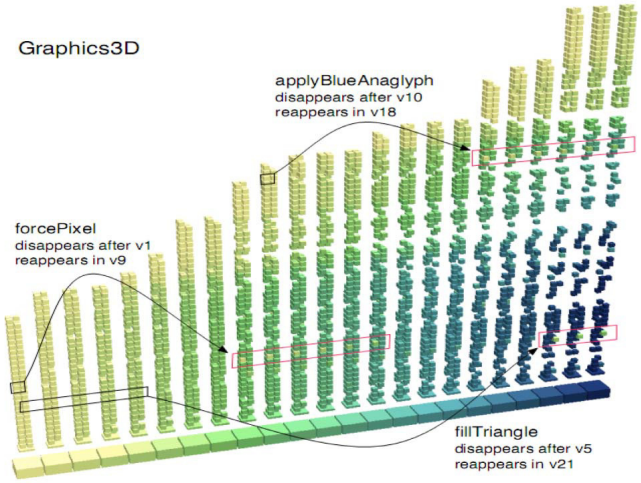

Рис.5. Эволюция класса Graphics3D.

Поскольку класс Graphics3D, начиная со своей первой версии, имел большое количество методов, то он относится к ядру системы. На изображении можно заметить исчезновение метода forcePixel после версии 1 и повторное его появление в версии 9. Поскольку вновь появившиеся методы имеют более светлую окраску по сравнению с более старыми методами с темной окраской, то такие вновь возникающие методы хорошо заметны. Также по изображению эволюции класса (появлению и исчезновению его методов) можно оценить стабильность класса различных этапах существования системы, что может повлиять на выбор используемой версии системы.

Важно отметить, что возможна и более простая в реализации двумерная визуализации эволюции класса - вид на здания-классы сбоку. На рисунке 6 показана матрица эволюции [15] [16]. Каждый прямоугольник представляет версию класса, и каждая строка хранит все версии этого класса. Строки упорядочены по именам классов. Размер прямоугольника может показывать различные измерения метрик класса. На рисунке 6 ширина прямоугольника показывает количество атрибутов класса, а высота прямоугольника показывает количество методов класса. Наличие таких метрик в изображении класса позволяет отследить в его истории фазы расширения, модификации и сокращения в результате появления новой функциональности класса или рефакторинга класса.

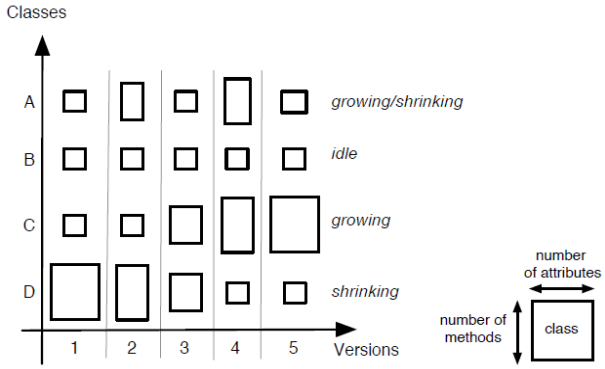

Рис.6. Принципы построения матрицы эволюции.

Визуализация системы в виде матрицы эволюции позволяет оценить размеры системы (высоту колонки в матрице). В матрице эволюции системы MooseFinder [16], показанной на рисунке 7, легко обнаруживаются вновь добавляемые классы, располагаемые в нижних строках матрицы, и удаленные классы, оставляющие пустые строки. Видны фазы роста и стагнации системы.

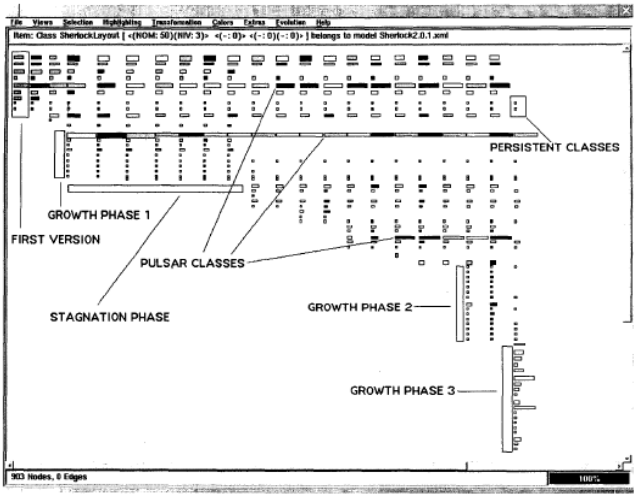

Рис.7. Матрица эволюции системы.

B. Визуализации эволюции отношений в системе

В работе [17] описана визуализация эволюции отношений между элементами программной системы. Так, например, в этой работе описывается визуализация отношений возникающих в результате наследования классов и вызовов методов класса другими методами (граф вызовов). На рисунке 8 показана эволюция графа наследования для системы SandMark [17].

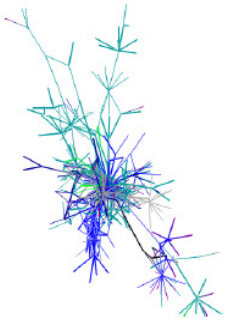

Рис.8. Граф наследования последовательных версий системы.

Узлы и отношения графа наследования в исходной версии расцвечиваются синим цветом. Вновь появляющиеся узлы графа или отношения расцвечивается следующим цветом в приводимой на рисунке 9 цветовой палитре:

Рис.9. Палитра для расцветки отношений.

Имеется возможность задавать отдельные палитры для каждого из авторов участвующих в разработке. Детальная информация об узлах и отношениях показывается с помощью всплывающих подсказок.

Следующий способ визуализации, показанный на рисунке 10, предоставляет очень компактное использование двумерного пространства [18]. Для узлов, которые не являются конечными в иерархии отношений, используются горизонтальные или вертикальные линии.

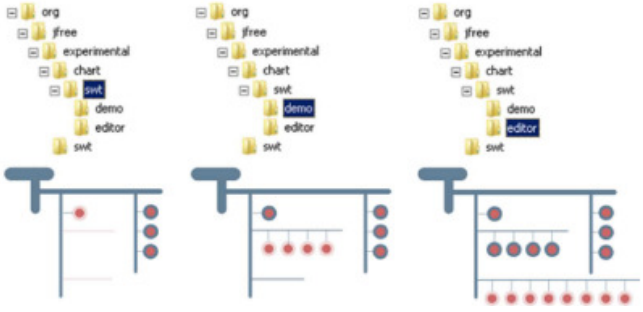

Рис.10. Компактное дерево для визуализации вложенности пакетов.

Например, горизонтальными и вертикальными линиями могут быть пакеты в иерархии вложенности, или класс-предок в иерархии классов. Кругами могут быть листья дерева, например, функции в иерархии вложенности элементов программы или классы без подклассов.

Другим достоинством такой визуализации отношений является то, что оно позволяет в интерактивном режиме сравнивать несколько иерархий отношений для различных версий системы. С помощью расцветки узлов и ветвей графа можно показывать различные версии системы, и последовательно спускаться внутрь иерархии отношений, если будет обнаружено различие в версиях.

В работах [19] [20] рассмотрена метафора города для визуализации эволюции отношений в программной системе. Ранее уже рассматривалась метафора города для визуализации структуры системы, в которой кварталы города представляли пакеты (пространства имен) системы, а здания представляли классы и интерфейсы системы. Для визуализации отношений в системе метафора эволюционирующего города использует улицы для представления отношений и здания для представления классов и интерфейсов. На рисунке 11 показаны три последовательные версии системы.

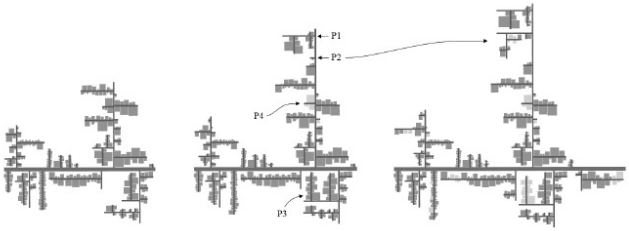

Рис.11. Эволюционирующие города для визуализации отношений.

Цветом могут быть показаны добавленные, удаленные и перемещенные элементы внутри иерархии. Помимо различий в версиях в таком городе высотой и шириной здания могут быть показаны различные свойства классов и интерфейсов. Например, для класса может быть показаны количество его модификаций и дата модификации.

В работах [21] [22] рассматривается использование полиметрических видов [23], [4], [24] для визуализации эволюции иерархии наследования. Полиметрический вид - граф с узлами и ребрами, в котором ширина, высота и цвет узла показывают значение метрики для элемента программы, представляемого этим узлом. Ширина и цвет ребра графа также могут показывать значения метрик вычисленных для отношений между элементами программы.
Для визуализации эволюции иерархии отношений наследования были предложен набор метрики позволяющих оценить интенсивность изменений в программной системе.

Возраст истории определяется как количество версий в истории. Далее будут рассмотрены возраст истории класса (ClassAge), возраст истории отношения наследования (InhAge), и возраст истории системы (SystemAge).

Считается что история была удалена, если последняя версия в истории не является частью последней версии системы. В зависимости от того какого рода история

анализируется можно ссылаться на удаленные истории класса, отношения наследования и системы.

Классы являются первичным элементом при проектировании и структурировании объектно-ориентированных систем. Функциональность системы определяется классами и их методами. Изменение класса можно рассматривать как добавление или удаление хотя бы одного метода. Тогда для измерения изменений в истории класса необходимо измерить различия в количестве методов в различных версиях класса.
Введем метрику ENOM (Evolution of Number of Methods). Определим ENOMi как разницу количестве методов между версиями i−1 и i класса C:

$$(i > 1)\ ENOM_i(C) = |\ NOM_i(C) - NOM_{i-1}(C)\ |$$

Определим $ENOM_{j..k}$ как сумму количества методов добавленных или удаленных в последовательных версий от версии j до версии k из n версий класса C:

$$(1 \leqslant j < k \leqslant n)$$
$$ENOM_{j..k}(C) = \sum_{i=j+1}^{k} ENOM_i(C)$$

Метрика ENOM будет показывать общее число изменений за время жизни класса C.

Обобщим метрику измерения истории изменений произвольного свойства P в истории измеряемого объекта H.

E1..n(P,H) - сумма абсолютных различий свойства P в последовательности версий от версии 1 (первой версии) до версии n (последней версии) в истории измеряемого объекта H [21].

$$(n > 1) \quad E_{1..n}(P,H) = \sum_{i=2}^{n} |P_i(H) - P_{i-1}(H)|$$

Выполним эти измерения для различных версий свойств класса: NOM - количество методов класса, NOS - количество предложений класса. В результате будут получены две метрики истории класса: эволюция количества методов класса и эволюция количества предложений.

$$ENOM_{1..n}(C) = E_{1..n}(NOM,C)$$

$$ENOS_{1..n}(C) = E_{1..n}(NOS,C)$$

Для более короткой записи будем считать, что $E(P,H) = E_{1..n}(P,H)$.

На рисунке 12 в матрице эволюции показаны описанные метрики:

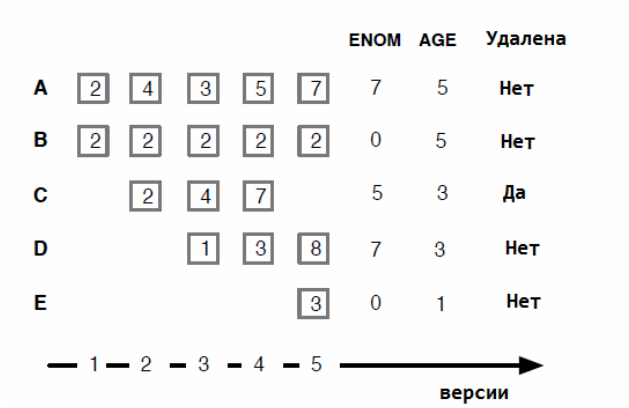

Рис.12. Матрица эволюции системы.

[x] это изображение в матрице эволюции представляет версию класса с **x** методами.

Далее для измерения истории иерархии классов используются групповые операторы среднее (Avg), максимум (Max) и общее значение (Tot). Используя эти операторы, можно записать средний возраст историй класса в иерархии классов Ch как Avg(ClassAge,Ch).

Для оценки возраста иерархии будут использоваться следующие типы историй иерархии наследования, вычисленные на основе среднего возраста истории их классов:

Новорожденная иерархия - это недавно введенная в систему иерархия:
Avg(ClassAge, Ch) < 0.1 * SystemAge

Молодая иерархия старше новорожденной иерархии но младше половины возраста системы:
Avg(ClassAge,Ch) > (0.1 * SystemAge) and
Avg(ClassAge,Ch) < (0.5 * SystemAge)

Старая иерархия старше молодой иерархии, но немного младше возраста системы:
Avg(ClassAge,Ch) > (0.5 * SystemAge) and
Avg(ClassAge,Ch) < (0.9 * SystemAge)

Постоянная иерархия почти совпадает с возрастом системы:
Avg(ClassAge, Ch) > 0.9 * SystemAge

Для определения места, где происходят изменения в иерархии, предлагается разделить их на две категории. Прочная иерархия - когда отношения наследования между классами стабильные и старые:
Tot(RemovedInh, Ch) < 0.3 * NOInh(Ch)
Хрупкая иерархия тогда, когда существует много исчезнувших отношений иерархии:
Tot(RemovedInh,Ch) > 0.3 * NOInh(Ch)

Для определения того насколько часто классы из одной иерархии модифицировались по сравнению с классами из другой иерархии предлагается свойство стабильности иерархии. Иерархия считается стабильной, если среднее число добавленных или удаленных методов и

предложений меньше чем среднее число добавленных или удаленных методов и предложений системы:

(Avg(ENOM,Ch) < Avg(ENOM, S)) and
(Avg(ENOS,Ch) < Avg(ENOS, S))

и нестабильной, если
(Avg(ENOM,Ch) > Avg(ENOM, S)) or
(Avg(ENOS,Ch) > Avg(ENOS, S))

Для оценки того, насколько изменения равномерно распределены среди классов иерархии, предлагается ввести понятие сбалансированности иерархии. Иерархия сбалансирована, если
Avg(ENOM,Ch) > 0.8 * Max(ENOM,Ch)
и несбалансированна, если
Avg(ENOM,Ch) < 0.8 * Max(ENOM,Ch)

Ниже на рисунке 13 приводится визуализация описанных метрик с помощью полиметрического вида сложности системы [23].

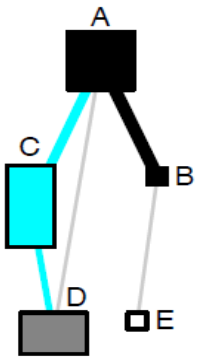

Рис.13. Изображение сложности эволюции иерархии наследования.

Узлы на диаграмме представляют историю класса. Ширина каждого узла диаграммы представляет метрику ENOM истории класса. Ширина каждого узла диаграммы представляет метрику ENOS истории класса. Цвет узла представляет возраст истории класса. Голубой цвет используется для удаленной истории. Ребра представляют истории наследования. Ширина ребра представляет возраст истории наследования. Цвет ребра представляет возраст истории наследования. Голубой цвет используется для удаленной истории.

Классы A и B присутствуют в системе с самого начала, и поэтому показаны черным цветом. Отношение наследования между этими классами показано черной тонкой линией, что применяется для старых отношений. Класс E небольшого размера и белого цвета, поскольку он введен в систему недавно.
Класс C удален из системы и поэтому голубого цвета. Класс D был введен после нескольких версий как подкласс класса C, но в последней версии стал подклассом класса A.
Класс B небольшого размера, поскольку в нем не обнаружено изменений. Классы A и D одинаковой ширины, но класс D имеет меньше добавленных или удаленных предложений, поэтому он имеет меньшую высоту. Класс C имеет высоту больше ширины, что означает большую затрату усилий на его реализацию.

На основе рисунка 13 можно сделать выводы об представленной иерархии наследования.
Неоднородность. История иерархии классов неоднородна, поскольку истории классов имеют существенно разный возраст. Такие иерархии показываются разными оттенками серого.
Нестабильный корень. На рисунке 13 иерархия с нестабильным корнем, поскольку узел-корень большего размера по сравнению с другими узлами.

Данный метод визуализации и анализа был применен к нескольким иерархиям наследования классов в системе JBoss Application Server [25].
На рисунке 14 показана иерархия наследования, начинающаяся с класса J2EEManagedObject.

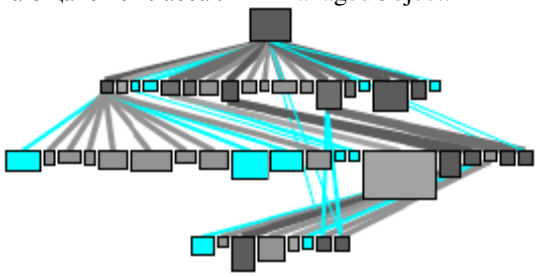

Рис.14. Иерархия наследования класса J2EEManagedObject

Эта иерархия неоднородна и не сбалансирована с точки зрения сделанных изменений. Иерархия наследования хрупкая, поскольку содержит множество удаленных отношений показанных голубым цветом, а остальные отношения показаны различными оттенками серого цвета.
Показанная на рисунке 15 иерархия наследования класса Stats появилась недавно и еще не содержит каких либо существенных изменений.

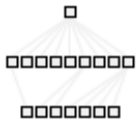

Рис.15. Иерархия наследования класса Stats.

На рисунке 16 показана иерархия наследования класса MetaData.

Рис.16. Иерархия наследования класса MetaData.

Узлы в иерархии либо голубого цвета, либо различных оттенков серого цвета. Это старая иерархия, которая нестабильна и не сбалансирована. Ребра в иерархии либо толстые и темные, либо голубые. Поскольку количество удаленных (голубых) ребер невелико, то

можно охарактеризовать отношения наследования как прочные.
Иерархия класса SimpleNode, показанная на рисунке 17 довольно старая с небольшим числом изменений сделанных в течение ее жизни. Это очень стабильная иерархия.

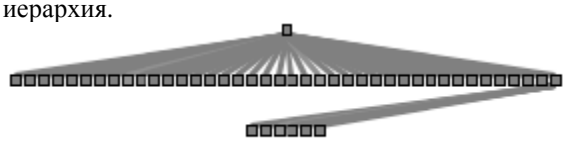

Рис.17. Иерархия наследования класса SimpleNode.

К отдельной категории способов визуализации программных систем относится визуализация с радиальной планировкой, предоставляющая возможность добиться более компактного расположения информации [26] [27] [28]. При одной из таких способов визуализации используется метафора годовых колец дерева [29]. На спиле этого дерева можно проследить историю роста этого дерева по так называемым годовым кольцам. Для программной системы такое годовое кольцо представляет отдельную версию программной системы.
В работе [30] изображение с радиальной планировкой информации используется для визуализации эволюции зависимости системы от используемых ею библиотек. На рисунке 18 показаны 12 версий системы FINDBUGS с 192 отношениями зависимости от 16 библиотек.

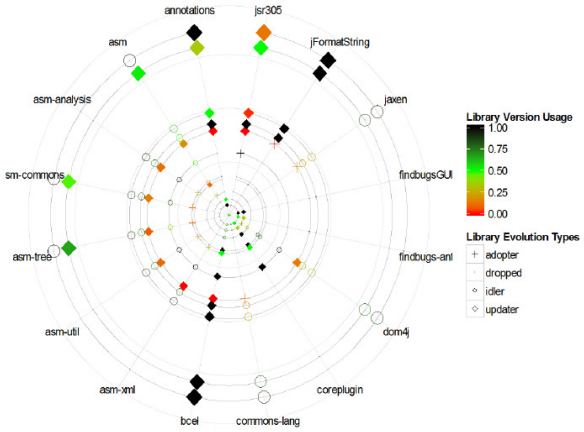

Рис.18. Визуализация зависимости системы от используемых библиотек.

На рисунке 19 каждая ось представляет библиотеку, от которой зависит система. Начиная от центра, каждое кольцо представляет версию рассматриваемой системы. Расстояние между кольцами показывает время между реализациями версий системы. На каждом кольце фигурами помечаются зависимости использования библиотек текущей реализацией системы. Тип отношения зависимости и степень зависимости показываются цветом и формой фигуры представляющей зависимость. Далее будут рассмотрены детально эти три визуальных свойства изображения.
Отношения зависимости рисуются с использованием радиальной системы координат, в которой каждая ось используется для представления библиотеки, от которой зависит система. Как отмечалось в работах [26] [31] [32] радиальная планировка позволяет сохранить структуру изображения по мере роста числа его элементов. Это важно, поскольку при появлении новых версий системы может существенно возрасти число используемых системой библиотек.

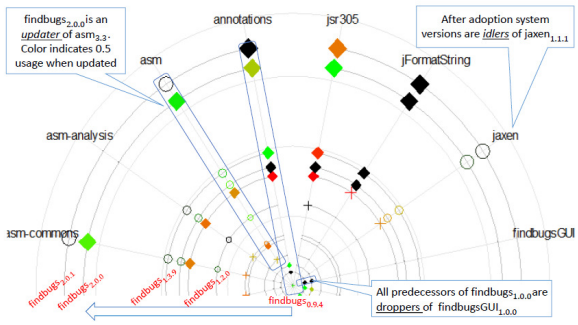

Рис.19. Секция визуализации зависимости системы от используемых библиотек.

Форма и цвет фигуры на каждой из осей координат представляет свойства используемой системой библиотеки.
Форма фигуры в точке пересечения кольца с осью координат описывает отношение зависимости системы текущей версии от библиотеки:

+ (адаптация) эта фигура означает, что система впервые адаптировала библиотеку.
◇ (обновление) эта фигура означает, что в текущей версии система начала использовать библиотеку новой версии.
○ (нет изменений) эта фигура означает, что сохраняется существующая зависимость.

Для библиотек, которые перестали использоваться, начиная с некоторой версии системы, фигуры более не рисуются. Что бы избежать взаимного пересечения и закрытия фигур для них рисуются только контуры с прозрачным содержимым. Исключение делается только для фигуры-обновления.
Использование на диаграмме расположенных на осях фигур позволяет оценить, насколько часто система обновляет свои библиотеки.

Цвет фигуры представляет степень используемости системой данной версии библиотеки. Красный цвет означает низкую используемость библиотеки. Зеленый цвет означает среднюю используемость библиотеки. Черный цвет означает высокую используемость библиотеки.
Есть основания полагать, что большинство разработчиков не склонны обновлять наименее используемые библиотеки. Описанная "карта температуры" с заданной яркостью цвета используемых библиотек может быть применена для визуальной оценки того, насколько система готова для адаптации новых версий библиотек.
Системы с более новыми версиями библиотек (менее используемыми) известны как системы-инноваторы. Можно выразить готовность к обновлению как отношение использования в момент адаптации к текущему использованию в репозитории проектов. Например, в репозитории Maven [11], можно узнать, что

в момент появления системы FindBugs версии 2.0.0, библиотека asm была обновлена с версии 3.1 на версию 3.3. В этот момент 36 аналогичных систем в репозитории Maven зависели от acm версии 3.3. В данный момент 78 систем используют acm версии 3.3. Получается соотношение 0.52 (зеленая интенсивность).
Можно также сделать вывод о готовности к обновлению слабо используемых версий библиотеки JSR305 (фигура красного цвета).
Введем переменную t которая представляет время, и переменную usage, которая представляет частоту использования библиотеки системами. Пусть переменная vt есть время, когда переменная была реализована. Тогда $usage_{vt}$ представляет использование в некоторой точке эволюции. Обозначим текущее время как ct. Тогда можно обозначить относительное использование библиотеки как отношение:

$$usage = usage_{vt} \ / \ usage_{ct}$$

Как можно видеть в примере на рисунке 18, непрерывная шкала цветов представляет различную интенсивность использования версий библиотек. Красный цвет означает систему, рано включившую библиотеку с высоким риском. Зеленый цвет означает, что система включила привлекательную библиотеку, которую уже использует большинство систем. И наконец, черный цвет означает, система включила уже стабильную библиотеку вместе с другими "неторопливыми" системами.

III. Заключение

В статье рассмотрены вопросы анализа и визуализации эволюции программных систем. Последовательно рассмотрены визуализация и анализ эволюции пакетов программных систем (пространств имен), классов программных систем и отношений между классами программных систем. Данные задачи весьма актуальны для восстановления эволюции программной системы при обратном проектировании системы с помощью CASE-инструмента. Данный инструмент основан на языке моделирования UML и реализован как расширение среды Eclipse, а также как облачное приложение фирмы Google. Статья является продолжением цикла публикаций по программной инженерии и применению языка моделирования UML, начатой в журнале INJOIT работами [1, 2, 4, 5, 6, 7, 8, 9]. Эта работа относится к числу одного из направлений исследований в Лаборатории ОИТ факультета ВМК МГУ [33] [34].

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] Source Forge https://sourceforge.net/

[11] Maven Central. http://mvnrepository.com/

[12] R. Wettel and M. Lanza, “Visual exploration of large-scale system evolution,” Proc. 15th IEEE Working Conference on Reverse Engineering (WCRE'08) 2008, pp. 219-228.

[13] G. Langelier, H.A. Sahraoui, and P. Poulin, “Exploring the Evolution of Software Quality with Animated Visualization,” Proc. IEEE Symp. Visual Languages and Human-Centric Computing, 2008.

[14] P. Caserta and O. Zendra, “Visualization of the static aspects of software: a survey,” IEEE transactions on visualization and computer graphics, vol. 17, no. 7, 2011, pp. 913-933.

[15] Tudor Gîrba and Stéphane Ducasse. Modeling History to Analyze Software Evolution. In Journal of Software Maintenance: Research and Practice (JSME) 18 p. 207—236, 2006

[16] M. Lanza, “The evolution matrix: Recovering software evolution using software visualization techniques,” Proc. 4th ACM International Workshop on Principles of Software Evolution (IWPSE'01), 2001, pp. 37-42.

[17] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A system for graph-based visualization of the evolution of software. In Proceedings of the 2003 ACM symposium on Software visualization, SoftVis ’03, pages 77–ff, New York, NY, USA, 2003. ACM.

[18] A. Gonzalez, Theron, R., Telea, A., Garcia, F. J., “Combined visualization of structural and metric information for software evolution analysis,” Proc. joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops, 2009, pp. 25-30.

[19] T. Khan, H. Barthel, A. Ebert, and P. Liggesmeyer. Visualization and Evolution of Software Architectures. In C. Garth, A. Middel, and H. Hagen, editors, Visualization of Large and Unstructured Data Sets: Applications in Geospatial Planning, Modeling and Engineering - Proceedings of IRTG 1131 Workshop 2011, volume 27 of OpenAccess Series in Informatics (OASIcs), pages 25–42, Dagstuhl, Germany, 2012.

[20] Frank Steinbrückner and Claus Lewerentz. Representing development history in software cities. In Proceedings of the 5th international symposium on Software visualization, SOFTVIS ’10, pages 193–202, New York, NY, USA, 2010. ACM.

[21] T. Girba, S. Ducasse, M. Lanza, Yesterday’s Weather: guiding early reverse engineering efforts by summarizing the evolution of changes, in: Proc. of International Conference on Software Maintenance, IEEE Computer Society, 2004, pp. 40–49.

[22] Gîrba T, Lanza M, and Ducasse S. Characterizing the Evolution of Class Hierarchies. 9th European Conference on Software Maintenance and Reengineering - CSMR 2005 (Manchester, UK), IEEE Computer Society Press; 2-11.

[23] M. Lanza and S. Ducasse, “Polymetric views-a lightweight visual approach to reverse engineering,” IEEE Trans. Softw. Eng., vol. 29, no. 9, pp. 782–795, Sep. 2003.

[24] R Francese, M Risi, G Scanniello, G Tortora. Proposing and assessing a software visualization approach based on polymetric views. Journal of Visual Languages & Computing 34, 2016, pp. 11-24

[25] JBoss Application Server. http://www.jboss.org

[26] K. Andrews and H. Heidegger, “Information Slices: Visualising and Exploring Large Hierarchies Using Cascading, Semi-Circular Discs,” Proc. IEEE Symp. Information Visualization, pp. 9-11, Oct. 1998.

[27] J. Stasko and E. Zhang, “Focus + Context Display and Navigation Techniques for Enhancing Radial, Space-Filling Hierarchy Visualizations,” Proc. IEEE Symp. Information Visualization, pp. 57-65, 2000.
[28] M. Chuah, “Dynamic Aggregation with Circular Visual Designs,” Proc. IEEE Symp. Information Visualization, pp. 35-43, 1998.
[29] Therón, R.: Hierarchical-Temporal Data Visualization Using a Tree-Ring Metaphor. In: Lecture Notes in Computer Science. Smart Graphics 2006, vol. 4663, SpringerVerlag, Germany, 70-81.
[30] R.G. Kula, et al., “Visualizing the Evolution of Systems and their Library Dependencies,” Proc. Second IEEE Working Conference on Software Visualization (VISSOFT'014) 2014, pp. 127-136.
[31] M. Krzywinski, I. Birol, S. J. Jones, and M. A. Marra, “Hive plots: A rational approach to visualizing networks,” Briefings in Bioinformatics, 2011.
[32] Hive plots. Rational network visualization. http://www.hiveplot.net/
[33] Намиот Д., Сухомлин В. О проектах лаборатории ОИТ //International Journal of Open Information Technologies. – 2013. – Т. 1. – №. 5. – С. 18-21.
[34] 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.

Software Evolution Analysis and Visualization

Romanov V.Y.

Abstract - **This article provides an overview of visualization techniques for the evolution of software systems in the reverse engineering tools and tools for restoring the architecture of a software system. This paper discusses methods for the visualization and analysis of changes in program elements and relations between them in various versions of software systems and libraries used by them. The article describes metaphors allow visualization software use familiar to the user of the tool the notion of their ordinary life - the metaphor of the growing city, the growing tree, "temperatures" in parts of a software system describes the intensity of what was happening in the system changes.**

Keywords — **software visualization, reverse engineering, software evolution, visualization metaphor.**