Визуализация для измерения и рефакторинга программного обеспечения.

Романов В.Ю.

Аннотация — **В статье приводится обзор методов оценки качества проекта программного обеспечения, методов обнаружения дефектов проекта программного обеспечения с помощью согласованного набора объектно-ориентированных метрик – стратегий обнаружения дефектов. Предложена классификация таких стратегий, построена модель дефектов и их взаимосвязей. Определены методы визуализации значений метрик с помощью набора полиметрических видов – как расширения традиционных диаграмм, используемых для визуализации моделей программного обеспечения.**

Ключевые слова — **object oriented metrics, polymeric view, software project defects, defect detection strategy.**

I. Введение

Традиционно, для представления элементов и связей элементов проекта системы использовались графические нотации, стандартизация которых привела к появлению графической нотации диаграмм унифицированного языка моделирования UML[1].

Для оценки качества проектов были предложены наборы объектно-ориентированных метрик [2, 3, 4], которые позволяют оценить исходные тексты и код программ написанных на различных объектно-ориентированных языках программирования. Результаты измерения качества проектов с помощью объектно-ориентированных метрик представлялись в виде таблиц содержащих числовые значения для десятков предопределенных метрик (колонок таблицы) и сотен измеренных элементов проекта (строк таблицы). Решение о том, каким образом следует интерпретировать представленные в табличной форме значения метрик, как их использовать для выполнения рефакторинга системы с целью улучшения характеристик проекта, оставлялось на усмотрение пользователю инструмента измерения. В ряде работ было предложено совместить визуализацию структуры модели программного обеспечения и визуализацию результатов измерения качества этого программного обеспечения [5, 6, 7]. Для апробации предлагаемых подходов была сделана экспериментальная реализация инструмента на языке Smalltalk[9].

В данной статье приводится обзор совместной визуализации структуры элементов модели и их связей, и количественных характеристик элементов модели представляемых значениями объектно-ориентированных метрик. В стандартной графической нотации языка UML, ни размеры элементов диаграммы, ни цвет этих элементов не несут семантической нагрузки. Как использовать эти свойства элемента диаграммы может решать пользователь CASE-инструмента. Эти три измерения можно использовать для визуализации количественных характеристик элементов модели, измеренных с помощью объектно-ориентированных метрик.

Важным является вопрос о целях измерения проекта. Измерение всех характеристик зачастую занимает чрезмерно много времени, а результаты измерений в большей части не используются. Более продуктивным является целенаправленное измерение для целей рефакторинга программной системы [9, 10, 11, 12].

Для автоматизированного обнаружения дефектов они должны быть описаны формально. Зачастую для обнаружения дефекта проекта системы бывает недостаточно лишь одной объектно-ориентированной метрики. Для обнаружения необходимо, чтобы значение нескольких метрик находились в определенных интервалах значений, заданных для каждой из метрик. Таким образом, формируются (программно или в диалоговом режиме) стратегии обнаружения дефектов – запросы, представленные как составные логические условия, основанные на комбинации метрик, с помощью которых в исходном коде обнаруживаются элементы проекта с описанными свойствами. В одной из обозреваемых работ [5] была сделана попытка классифицировать некоторые из распространенных программных дефектов и описать их взаимосвязи. Дефекты были классифицированы следующим образом. Индивидуальные дефекты – те дефекты, которые влияют на отдельные элементы проекта. Например, дефекты класса или метода. Для оценки влияния этих дефектов на качество проекта эти элементы проекта могут быть рассмотрены в изоляции. Дефекты взаимодействия – это дефекты, затрагивающие несколько элементов проекта в способе их взаимодействия при выполнении ими некоторой функциональности. Дефекты классификации – это дефекты, возникающие у класса в иерархии наследования между его предками и потомками.

Для визуализации результатов поиска дефектов были предложены два полиметрических вида. Вид «черновик класса» - это полиметрический вид с уровневой визуализацией, показывающей потоки управления класса и доступ к атрибутам класса. Вид «сложность системы» показывает иерархию наследования указание сложности каждого элемента вида – количество атрибутов, методов класса и его объем в строках или байтах. Эти два полиметрических вида используются для визуализации обнаруженных дефектов.

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

Данный обзор представляет собой введение в терминологию и понятия, используемые при измерении программных систем с целью их последующего рефакторинга. Эти терминология и понятия используются в разработке инструмента обратного проектирования инструмента обратного проектирования и рефакторинга программного обеспечения написанного на языке Java [13, 14, 15, 16]. В последующих работах предполагается рассмотреть, в частности, возможности расширения метамодели языка UML [1] с помощью стандартного механизма расширения этого языка – профилей [17, 18, 19]. Профили предполагается использовать для интегрированного представления в метамодели языка UML измеряемых элементов модели, дефектов этих элементов и объектно-ориентированных метрик использованных для обнаружения дефектов. Для языка моделирования UML разработано унифицированное представление UML диаграмм [20, 21]. Это представление дополняется также и возможностью представления полиметрических видов. В качестве контекста для сбора метрик используется не только представление программной системы в виде UML модели, но и преставление системы в виде абстрактного синтаксического дерева среды Eclipse [22]. Такое преставление позволяет осуществлять сложные виды рефакторинга, основанные на измерения, с помощью элементарных видов рефакторинга предоставляемых API Eclipse [23].

II. Сбор характеристик проекта

A. *Полиметрические виды*

Узлы графа используются для представления элементов модели программной системы или их абстракций. Ребра используются для представления отношений между этими элементами. Такая графическая нотация традиционно используется для представления моделей. Для визуализации метрической информации об элементах модели используются расширения графической нотации, показанные на рисунке 1.

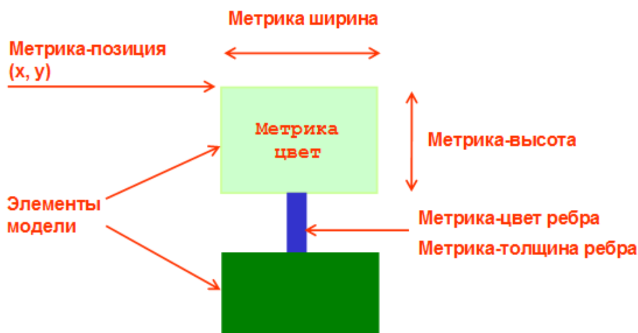

Рис.1. Принцип формирования полиметрического вида.

Размер узла графа (ширина и высота) представляет измерения двух различных метрик, сделанных для представляемого этим узлом элемента модели.

Имя элемента диаграммы, размеры которого используются для представления значений метрик, показываются с помощью всплывающей подсказки. При необходимости диаграмма может быть развернута в состояние с показом имен элементов диаграммы, а потом свернута в исходное состояние.

Цвет узла графа представляет измерение третьей метрики для такого элемента модели. Значения цвета могут задаваться в шкале серого цвета для представления всех значений метрики, либо определенными цветами для выделения элементов, значение метрики которых превысили некоторое пороговое значение.

На приводимом ниже рисунке 2 в примере высота узла представляет собой метрику Number Of Methods (NOM) показывающую количество методов в классе, ширина узла представляет метрику Number Of Attributes (NOA) показывающую количество атрибутов в классе, а оттенки серого цвета узла представляет значения метрики Lines Of Code (LOC).

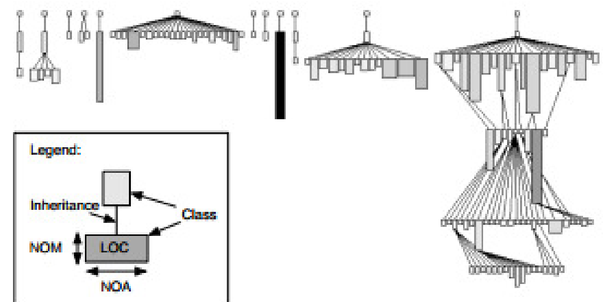

Рис. 2. Использование высоты, ширины и цвета узла в полиметрическом виде «Сложность системы».

Позиция узла – для некоторых полиметрических видов координаты узла представляют собой значения двух метрик. Для такого полиметрического вида требуется наличие системы координат.

На рисунке 3 показан пример использования полиметрического вида, использующего координаты для задания значение метрик.

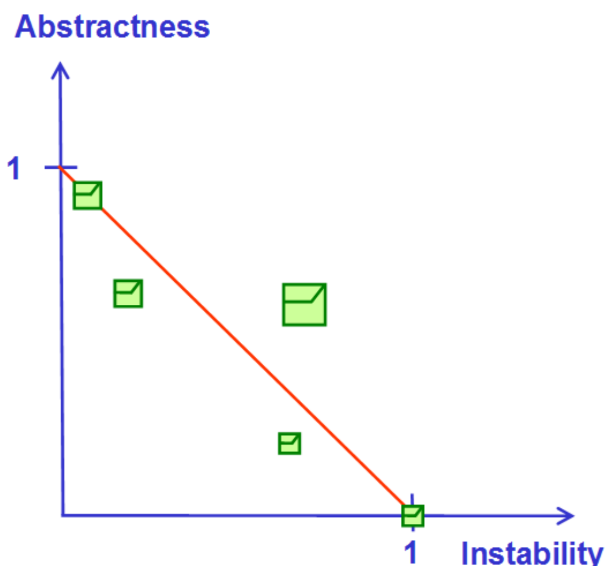

Рис.3. Полиметрический вид со значениями метрик – представленных координатами.

На диаграмме зеленым цветом показаны узлы графа – пакеты языка Java. Размер узла показывает количество

классов в пакете. Значения метрик Абстрактность (Abstractness) и Нестабильность (Instability) [10, 13] задаются координатами узлов. Красной линией показана «главная последовательность». Близость к этой линии обеспечивает оптимальную пропорцию метрик абстрактности и нестабильности пакета.

Толщина и цвет ребра графа могут дополнительно представлять в полиметрическом виде значения еще двух метрик. Например, представлять интенсивность вызова методов одного класса методами другого класса.

B. Полиметрический вид «Пирамида обзора»

Этот вид представляет набор связанных метрик из категорий: метрики наследования (inheritance), размера/сложности (Size&Complexity) и связанности (Coupling). Этот вид используется для получения «первого впечатления» о системе. Особенность этого вида в том, что на нем метрики одной категории располагаются рядом, в одной области, как показано на рисунке 4.

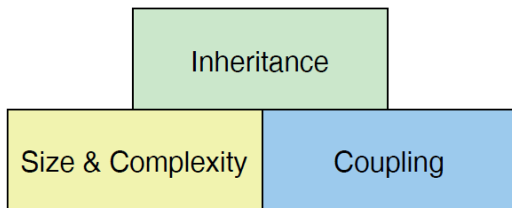

Рис.4. Вид – пирамида обзора.

Значения метрик из одной категории принимают возрастающие значения и показываются шириной узла. Таким образом, получается изображение ступенчатой пирамиды как показано на рисунке 5.

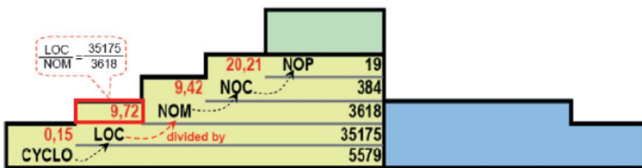

Рис.5. Метрики размера и сложности в пирамиде обзора.

На рисунке 5 показаны значения метрик размера и сложности: Number Of Packages (NOP) – количество пакетов, Number Of Classes (NOC) – количество классов, Number Of Methods (NOM) – количество методов, Line Of Code (LOC) – количество строк кода в системе, Cyclomatic Complexity (CYCLO) [2] – метрика определяющая насколько код программы является разветвленным (Условная сложность). Например, значение метрики 0.2 означает, что новая ветвь добавляется через каждые 5 строк. Для ступеней этой пирамиды вычисляются средние пропорции, показанные на рисунке 5 красным цветом. Например, пропорция LOC / NOM = 9.27 показывает, насколько хорошо код распределен среди методов.

На рисунке 6 показаны метрики сцепления (Coupling) в системе посредством вызова методов. С помощью этих метрик определяется насколько сцепление интенсивно и распределено. Метрика CALLS – представляет общее число вызовов различных методов в системе. Если какой либо метод вызывается в другом методе многократно, то он учитывается только один раз. Вторая метрика вычисляется как сумма метрик FANOUT [2] (количество вызванных методом классов) для всех методов системы. То есть классов, из которых операции вызывают методы.

С использованием указанных значений метрик вычисляются две пропорции: интенсивность сцепления Coupling Intensity = CALLS / NOM, и дисперсия сцепления Coupling Dispersion = (FANOUT / Operation Call). Высокое значение интенсивности сцепления означает чрезмерное общение метода со своими «коллегами». Высокое значение дисперсии сцепления означает чрезмерное общение с «коллегами» из других классов. Например, значение 0.2 для дисперсии означает, что каждый второй вызов направлен в другой класс.

Рис.6. Метрики сцепления (coupling) в пирамиде обзора.

Наверху пирамиды обзора расположены метрики представляющие использование в системе иерархии классов и полиморфизма. Эти метрики определяют насколько измеряемая система является объектно-ориентированной.

Первая метрика - Average Number of Derived Classes (ANDC) вычисляется как общее число прямых подклассов класса, деленное на количество классов в измеряемой системе. Интерфейсы в подсчете не учитываются. Используемые при наследовании в качестве предков библиотечные классы также не учитываются. Если у класса нет потомков, то этот класс добавляет 0 к значению метрики ANDC.

Вторая метрика - Average Hierarchy Height (AHH) есть общая высота наследования. Она вычисляется как сумма метрик Height of the Inheritance Tree (HIT) - высоты дерева наследования вычисленной для всех классов являющихся корнем в дереве наследования. Для классов, которые не имеют предков, значение метрики HIT равно 0.

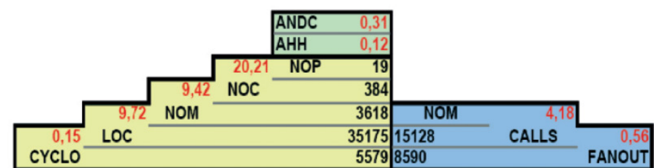

Рис.7. Метрики наследования в пирамиде обзора.

Таким образом, метрика ANDC представляет ширину деревьев наследования системы, а метрика AHH представляет глубину этих деревьев. Как можно видеть, пирамида обзора предоставляет показанные черным цветом значения прямых метрик (зависят от размера системы) и показанные красным пропорции (не зависимые от размера системы).

Для интерпретации полученных пропорций можно использовать статистические данные, которые получены в результате измерения систем написанных на языках Java (45 систем) и C++ (37 систем) [6]:

Metric	Java Low	Java Average	Java High	C++ Low	C++ Average	C++ High
CYCLO/Line of code	0.16	0.20	0.24	0.20	0.25	0.30
LOC/Operation	7	10	13	5	10	16
NOM/Class	4	7	10	4	9	15
NOC /Package	6	17	26	3	19	35
CALLS/Operation	2.01	2.62	3.2	1.17	1.58	2
FANOUT /Call	0.56	0.62	0.68	0.20	0.34	0.48
ANDC	0.25	0.41	0.57	0.19	0.28	0.37
AHH	0.09	0.21	0.32	0.05	0.13	0.21

Рис.8. Статистические значения пропорций.

По сравнению со статистическими данными, измеряемая система (написанная на языке Java) имеет низкую внутреннюю сложность, средний размер методов, и большие пакеты и классы. Система сильно сцеплена посредством вызова методов, однако эти вызовы локализованы. Большинство вызываемых методов расположено в небольшом количестве классов. В измеряемой системе часто используются иерархии классов имеющие очень небольшую глубину.
Расположив средние значения пропорций на голубом фоне, низкие на зеленом и высокие на красном, получим следующую пирамиду обзора:

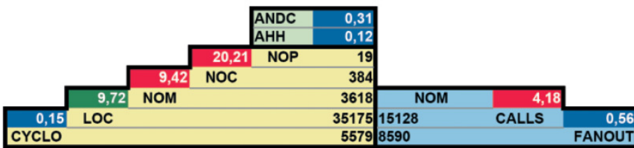

Рис.9. Использование фонового цвета для интерпретации значений пропорций.

III. Оценка проекта

A. Стратегии обнаружения

Стратегии обнаружения дефектов – это запросы, представленные как составные логические условия, основанные на комбинации метрик, с помощью которых в исходном коде обнаруживаются элементы проекта с описанными свойствами.

Приведем пример формирования такого запроса для поиска в проекте классов взявших на себя чрезмерную ответственность God Classes «божественный класс». В запросе на поиск дефекта будут использоваться следующие метрики:

Weighted Method Count (WMC) «количество взвешенных методов». Эта метрика вычисляется как сумма метрик сложности (CYCLO) каждого метода класса.

Tight Class Cohesion (TCC) «тесная связанность класса». Вычисляется как относительное количество методов класса связанных друг с другом через доступ к атрибутам класса.

Access To Foreign Data (ATFD) «доступ к внешним данным». Количество внешних классов, из которых предоставляется доступ к атрибутам данного класса, непосредственно или через методы.

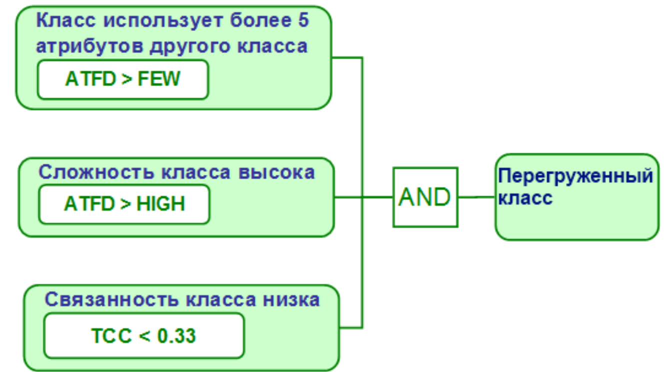

Рис.9. Формирование в диалоговом режиме стратегии обнаружения дефекта.

Формирование запроса с подбором необходимых метрик и логических выражений может быть выполнено динамически в диалоговом режиме.

B. Полиметрический вид «Черновик класса»

Полиметрический вид черновика класса облегчает понимание структуры класса без необходимости при этом читать весь код класса. Черновик класса структурирован на уровни, группирующие атрибуты и методы класса. Узлы графа на черновике представляют методы и атрибуты класса. Цвет узлов представляет семантику узла. Например, абстрактность метода, переопределение им метода базового класса, возврат значений атрибутов. Размер узла представляет значение метрики исходного кода класса. На рисунке 10 приведено уровневое представление черновика класса.

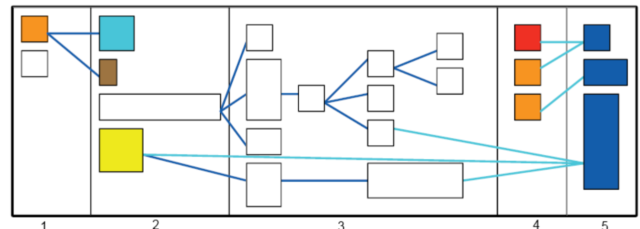

Рис.10. Черновик класса.

Узлы, расположенные слева выполняют вызов узлов-методов, расположенных справа, либо получают доступ к узлам-атрибутам расположенным справа.

1-ый уровень инициализации. Он представляет методы класса: конструкторы и методы с приставкой init.

2-ой уровень внешний интерфейс. Здесь расположены методы, удовлетворяющие одному из условий: вызываются методами из уровня инициализации; имеют не скрытую видимость; не вызываются другими методами класса, а только внешними методами или методами подклассов. Метод, вызываемый и извне, и внутри класса, помещается на уровень реализации. Сюда не включаются методы доступа (getters и setters), которые выделены в отдельный уровень доступа.

3-ий уровень внутренней реализации, на котором расположены методы ядра класса, видимость которых вне класса не предполагается. Сюда помещаются методы, которые вызываются хотя бы один раз другим методом этого класса.
4-ый уровень доступа. Здесь расположены методы, единственное назначение которых получить или записать значение атрибута.
5-ый уровень атрибутов. Атрибуты на этом уровне соединены отношениями с методами расположенными на других уровнях.

Отношение вызова одного метода другим методом показывается синим цветом. Отношение доступа метода к атрибуту показывается голубым цветом.

Как уже упоминалось, размеры узлов в черновике класса представляют значения метрик для этого класса. На рисунке 11 показаны метрики для узла-метода:

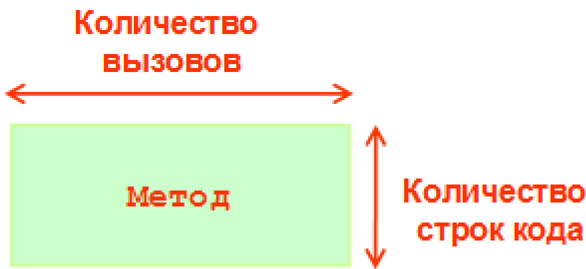

Рис.11. Представление метрик метода в черновике класса

Ширина узла-метода представляет собой количество статических вызовов идущих извне этого узла.
Для представления узла-атрибута используются метрики доступа к атрибуту из внешних и внутренних методов, как это показано на рисунке 12.

Рис.12. Представление метрик атрибута в черновике класса

Заметим, что имена методов и атрибутов на черновике класса можно, при желании, показать, а затем скрыть.
Для представления свойств методов используется следующее их представлением цветом:

Рис.13. Представление свойств методов с помощью цвета

Заметим, что в зависимости от используемого языка программирования цветом могут быть заданы и другие свойства класса и его элементов.

IV. Дефекты проекта

На рисунке 14 изображены некоторые из возможных дефектов проекта[6].

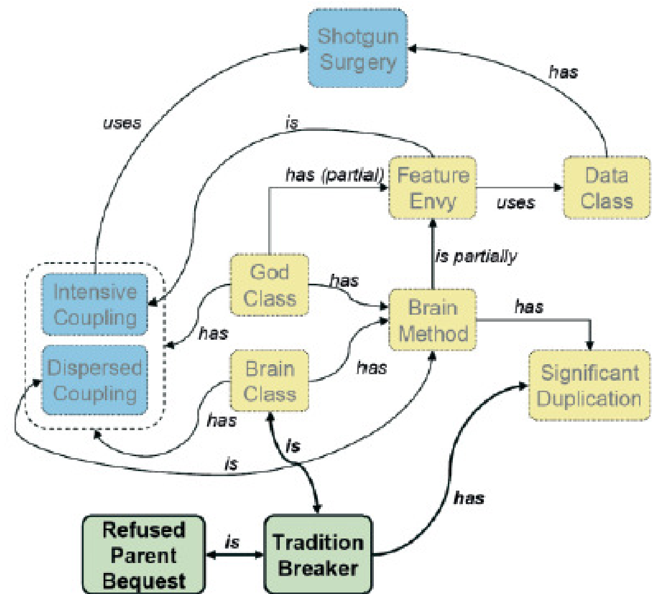

Рис.14. Дефекты проекта

Желтым цветом выделены индивидуальные дефекты классов, синим цветом – дефекты взаимодействия объектов, зеленым – дефекты классификации объектов.
На рисунке 14 указанные классы связаны отношениями. Далее приводится соответствие надписей на отношениях рисунка отношениям языка UML[1]: is – отношение наследования, has – отношение ассоциации (со свойством агрегации), uses – отношение зависимости со стереотипом <<uses>>. Сделаем обзор описанных категорий дефектов и отношений между ними.

A. *Индивидуальные дефекты*

Индивидуальные дефекты – это дефекты конкретного класса или метода класса. Негативное влияние таких дефектов на качество элементов проекта может быть обнаружено рассмотрением этих элементов в изоляции. Дефект элемента проекта может быть обусловлен тремя его аспектами – размером, интерфейсом и реализацией.

Data Class «класс данных» – дефект класса, который просто хранит данные, не имея сложной функциональности. Вместе с тем существует множество других классов, которые ссылаются на класс данных. Возможно, функциональность класса данных рассредоточена и дублируется в классах использующих класс данных. Дефектный «класс данных» может включать в себя элемент с дефектов взаимодействия Shortgun Surgery «хирург с дробовиком». Изменения в дефектном методе дефектного «класса данных» приведет к необходимости изменений во множестве методов и классов использующих дефектный метод. В свою очередь «класс данных» может быть использован в дефекте Feature Envy, который рассматривается ниже.

Feature Envy «завистливый метод» - методы с таким дефектным классом более интересуются атрибутами других классов, чем атрибутами собственного класса. Возможно, для завистливых методов был ошибочно определен владеющий ими класс, и «завистливые методы» должны быть перемещены в классы, которыми они интересуются.

Significant Duplication «существенное дублирование». Фрагменты кода, имеющие существенную длину и расположенные достаточно близко (в пределах одного метода или методов одного класса).

Brain method «метод-умник». Часто метод, который был создан как «нормальный», начинает брать на себя все большую функциональность, становясь все более тяжелым для понимания, отладки и сопровождения. Возможно, дефектный метод может иметь также дефект «существенное дублирование». Дефект метода обнаруживается по длине кода, избыточному ветвлению и большому числу используемых переменных.

Brain class «класс-умник». Класс, включающий в себя все большее число «методов-умников». Имеет сходство с дефектом God Class, поскольку также ссылается на сложный класс. Отличается от этого дефекта двумя характеристиками.
Особенность дефекта God Class не только в его сложности, но и в том, что он снижает инкапсуляцию, пытаясь получить прямой доступ ко многим атрибутам в других классов. Дефект «класс-умник» обнаруживает классы, не обнаруживаемые дефектом God Class, поскольку учитывает сложные классы либо не работающие интенсивно с атрибутами других классов, либо имеющие меньшее сцепление с другими классами. Это не God Class, имеющий по меньшей мере один «метод-умник».

God class «божественный класс». Этот класс централизует интеллектуальность системы, выполняя самостоятельно большую часть работы, делегируя минимальные детали тривиальным классам, и используя данные других классов.

В качестве примера формирования стратегии обнаружения и визуализации индивидуального дефекта рассмотрим дефект «Класс данных». Такой класс просто хранит данные, не имея сложной функциональности. Вместе с тем существует множество других классов, которые ссылаются на класс данных. Возможно, функциональность класса данных рассредоточена и дублируется в классах использующих класс данных. Таким образом, нарушается принцип инкапсуляции: данные и функциональность для работы с ними должна быть в одном классе.

Сформируем для класса данных стратегию обнаружения, как показано на рисунке 15.

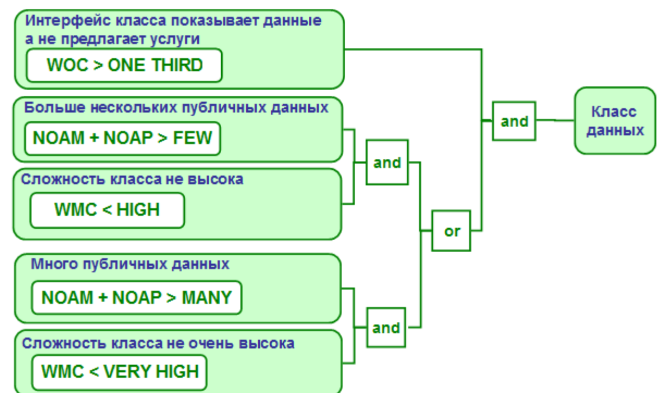

Рис.15. Стратегия обнаружения дефекта «класс данных»

Для выявления классов выставляющих данные, а не предоставляющих сервис, использовалась метрика Weight Of Class (WOC) вес класса. Эта метрика вычислялась как количество "функциональных" (не getter и setter) публичных методов, деленное на общее количество публичных членов класса [2]. Таким образом, мы убеждаемся, что интерфейс класса «занят» данными и методами доступа к данным. Теперь убедимся, что абсолютное число таких нарушителей инкапсуляции велико. Выделим далее два случая.

Классические «классы данных» не очень большие, почти без функциональности, и предоставляют только данные и методы доступа к ним. В таком случае значение метрики Weighted Method Count (WMC), суммы статической сложности всех методов класса, должно быть невысоко (WMC < HIGH). Кроме того, необходимо убедиться, что количество представителей публичных данных Number Of Public Attributes (NOPA) и Number Of Accessor Method (NOAM) больше чем несколько. Таким образом, мы исключим из дефектных такие «классические» классы данных как, например,

class Point { int x, y; }

или

class Size { int width, height; }

В противном случае рассмотрим большие классы, которые выглядят «нормальными» (определяющими некоторую функциональность), за исключением того факта, что их публичный интерфейс, помимо предоставленных сервисов, содержит значительное число данных и методов доступа данных. В этом

случае, что бы признать этот класс дефектным, он должен иметь много публичных данных (NOAM + NOAP > MANY) и не быть очень сложным (WMC < VERY HIGH).

На рисунке 16 показан обнаруженный дефектный класс Property содержащий пять атрибутов.

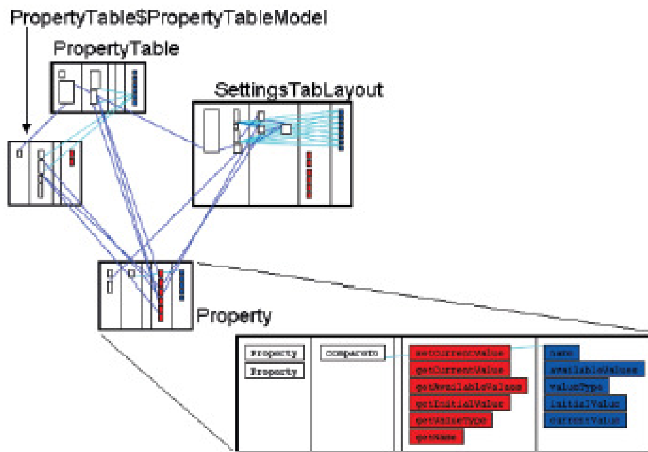

Рис.16. Дефектный класс Property

Поскольку класс Property имеет только методы доступа к атрибутам, он является чистым хранителем данных. Атрибуты данного класса используются тремя классами: *PropertyTable*, *PropertyTable&PropertyTableModel* и *SettingTableLayout*. Следующим шагом может стать выделение в этих трех классах фрагментов кода в отдельные методы, а затем перемещение этих методов как сервисов в класс Property. Возможно, что код в этих методах дублируется, и поэтому возможна унификация и сокращение числа этих методов.

B. Дефекты взаимодействия

Дефекты взаимодействия затрагивают способ взаимодействия нескольких элементов проекта. Взаимодействия должны осуществляться только посредством вызова методов. На такие взаимодействия должны быть наложены ограничения.

Ограничение интенсивности. Методы должны взаимодействовать с ограниченным числом сервисов предоставляемых другими классами Взаимодействие должно быть однонаправленным.

Ограничение пространства (extent). Методы должны взаимодействовать (вызывать и быть вызванными) с методами из ограниченного числа других классов.

Ограничение разброса (dispersion). Вызывающие и вызываемые методы должны иметь ограниченный разброс внутри системы. Они должны взаимодействовать с методами расположенными (в порядке предпочтения): 1-в той же абстракции, 2-в той же иерархии наследования, 3- в том же пакете.

Для оценки качества проектирования взаимодействия элементов проекта определены следующие дефекты:

Intensive Coupling «интенсивное сцепление» - дефект метода использующего небольшое число классов и огромное число методов в этих классах.

Dispersed Coupling «распределенное сцепление» - дефект метода использующего методы из очень большого числа классов.

Shotgun Surgery «хирург с дробовиком» - дефект метода, который используется в большом количестве классов. Изменения в этом методе приведут к необходимости изменения в большом количестве методов из различных классов.

В качестве примера стратегии обнаружения дефекта рассмотрим формирование дисперсии сцепления на рисунке 17.

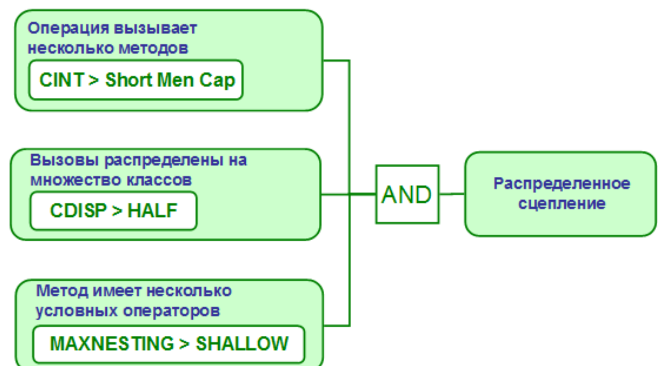

Рис.17. Стратегия обнаружения дефекта «распределенное сцепление».

При определении стратегии обнаружения метода использовались следующие метрики:

Coupling intensity (CINT) - количество различных методов, вызываемых данным методом.

Coupling Dispersion (CDISP) - количество классов, из которых вызываются методы из измеряемого метода, деленное на значение метрики **CINT**.

Maximum Nesting Level (MAXNESTING) – максимальный уровень вложенности управляющих структур в измеряемом методе.

Результат применения такой стратегии обнаружения показан рисунке 18 как полиметрический вид «сложность системы». Ширина и высота узлов – количество атрибутов и методов классов, цвет – количество строк исходного текста. Красным цветом показаны не измеряемые классы из Java библиотек. Показан также дефектный класс ActionOpenProject с большим количеством вызываемых (вызовы на рисунке показываются синим цветом) методов из других классов.

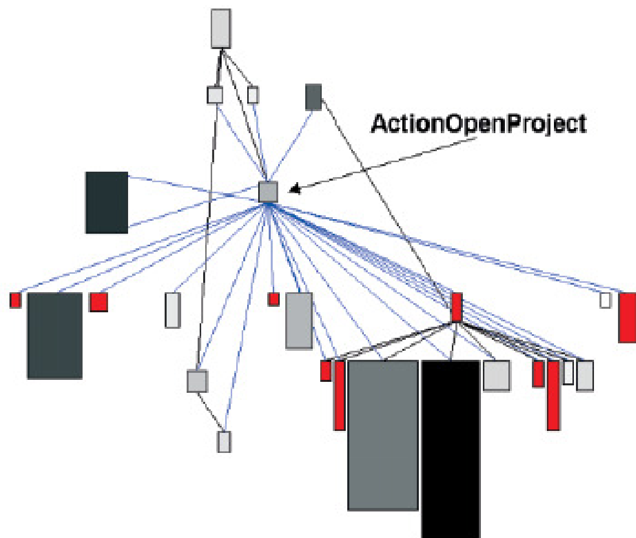

Рис.18. Обнаруженный класс ActionOpenProject с дефектом «распределенное сцепление»

C. Дефекты классификации

Дефекты классификации – это дефекты взаимодействия класса со своими предками или потомками. При организации иерархий наследования следует избегать слишком широких или слишком узких иерархий. Для этого необходимо постепенное расширение интерфейсов в иерархии наследования интерфейсов. Необходимо плавное уменьшение абстрактности в иерархии наследования классов. Абстрактные классы должны быть в верхней части иерархии. Необходима плавная специализация в иерархии классов. При этом не должно быть отказа от поведения, унаследованного от предков, а лишь дальнейшая специализация этого поведения. Взаимодействия в иерархии наследования должны быть направлены только в сторону предков и должны использоваться только для уточнения поведения потомков.

Для оценки качества проектирования классификации элементов проекта определены следующие дефекты:

Refused Parent Bequest «отказ от наследства предков». Не дефектный потомок класса использует наследство, если осуществляет доступ к защищенным атрибутам предка, вызывает защищенные методы предка, перекрывает или специализирует методы предка. Для измерения этих характеристик потомка используются метрики:

Baseclass Usage Ratio (BUR) коэффициент использования базового класса – насколько класс использует защищенные атрибуты и методы предка.

Base-class Overriding Ratio (BOvR) – коэффициент перекрытия и специализации методов базового класса.

Number of Protected Members (NPrM) вычиляет количество доступного для использования наследства.

Вычислим по формуле следующую характеристику:

```
Класс-потомок-игнорирует-наследство =
// Предок предоставляет более чем
// несколько защищенных атрибутов и
// методов
(  (NProtM  >  FEW)  AND
// Потомок использует небольшую часть
// наследства.
   (BUR < A THIRD) ) OR
// Потомок редко перекрывает методы
    (BOvR  <  A THIRD);

Класс-потомок-не-маленький-и-простой =
// Функциональная сложность потомка
// выше средней
( (AMW > AVERAGE) OR
// Функциональная сложность класса
// не ниже средней
  (WMC > AVERAGE) ) AND
// Размер класса выше среднего
(NOM > AVERAGE);

Деффект-отказ-от-наследства =
Класс-потомок-игнорирует-наследство
AND
Класс-потомок-не-маленький-и-простой;
```

Рис.19. Обнаруженные классы с дефектом «отказ от наследства»

Tradition Breaker «нарушитель традиций». Таким дефектом обладают классы-потомки, у которых произошло существенное увеличение интерфейса по сравнению с классом-предком. Класс-потомок имеет существенно больший размер по сравнению с предком. Класс-предок при этом не является небольшим или простым классом.

V. Заключение

Обзор, приведенный в данной статье, служит введением в терминологию и понятия, используемые при измерении программных систем с целью их последующего рефакторинга. Эти терминология и понятия используются в разработке инструмента обратного проектирования инструмента обратного проектирования и рефакторинга программного обеспечения написанного на языке Java. Статья является продолжением цикла публикаций по программной инженерии и применению UML, начатой в журнале INJOIT работами [13, 14], а также отраженной в более ранних публикациях [15, 16]. Эта работа относится к числу одного из направлений исследований в Лаборатории ОИТ факультета ВМК МГУ [24].

Библиография

[1] Object Management Group, UML 2.4 Superstructure Specification, OMG document. http://www.omg.org/spec/UML/2.4.1/

[2] Mark Lorenz and Jeff Kidd. Object-Oriented Software Metrics: A Practical Guide. Prentice-Hall, 1994.

[3] B. Boehm and W. Brown. Value-based software metrics. IEE Seminar Digests, 2004(909):4–6, 2004.
[4] Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz. Finding Refactorings via Change Metrics. OOPSLA ‘00, 10/00 Minneapolis, MN, USA 2000 ACM ISBN 1-58113-200-x/00/0010
[5] Michele Lanza, Radu Marinescu, S. Ducasse. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer 2006, ISBN 978-3540244295
[6] Michele Lanza and Stéphane Ducasse, Polymetric Views—A Lightweight Visual Approach to Reverse Engineering, Transactions on Software Engineering (TSE), 29, 782–795, IEEE Computer Society, 2003
[7] Michele Lanza and Stéphane Ducasse, Understanding Software Evolution Using a Combination of Software Visualization and Software Metrics, Proceedings of Langages et Modèles à Objets (LMO'02), 135–149, Lavoisier, 2002
[8] Moose - platform for software and data analysis http://www.moosetechnology.org/
[9] Martin Fowler. Refactoring. Improving the Design of Existing Code.
[10] Robert C. Martin, Designing Object Oriented Applications using UML, 2d.ed. Prentice Hall, 1999
[11] Martin, Robert C. Agile Software Development, Principles, Patterns, and Practices. Prentice Hall, 2002, ISBN 978-0135974445.
[12] Kirk Knoernschild Java Application Architecture: Modularity Patterns with Examples Using OSGi, Addison-Wesley Professional, 2012, ISBN 978-0-321-24713-1
[13] 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.
[14] 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.
[15] 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, с. 343-348
[16] Романов В.Ю. Реализация метамодели языка UML на основе хранилища данных фирмы Google. Сб. трудов VII Международной научно-практической конференции "Современные информационные технологии и ИТ-образование". М.:, 2012. с.605-610.
[17] UML Profiles And Related Specifications: http://www.uml.org/#UMLProfiles
[18] Introduction to UML2 Profiles. http://wiki.eclipse.org/ MDT/UML2/ Introduction_to_UML2_Profiles
[19] Customizing UML: Which Technique is Right for You? http://www.eclipse.org/modeling/mdt/uml2/docs/articles/ Customizing_UML2_Which_Technique_is_Right_For_You/article.html
[20] Object Management Group, UML Diagram Interchange, OMG document, http://www.omg.org/spec/UMLDI/1.0/PDF
[21] Object Management Group, Diagram Definition, Version 1.0, http://www.omg.org/spec/DD/1.0/
[22] Abstract Syntax Tree, http://www.eclipse.org/articles/article.php ?file=Article-JavaCodeManipulation_AST/index.html
[23] The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs, http://www.eclipse.org/articles/Article-LTK/ltk.html
[24] Намиот Д., Сухомлин В. О проектах лаборатории ОИТ //International Journal of Open Information Technologies. – 2013. – Т. 1. – №. 5. – С. 18-21.

Visualization of software measurement and refactoring

Romanov V.Y.

Abstract — **the article provides review of methods for projects quality measuring, software defects detections strategies, software measures visualization with the polymetric views. The object oriented metrics for quality assurance of design and refactoring methods to correct the design defects are described.**

Keywords — **object oriented metrics, polymeric view, software project defects, defect detection strategy.**