СУПРОВІД І ЕВОЛЮЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

УДК 004.415.2.045 (076.5)

А.С. Нечай

ПОСТРОЕНИЕ МОДЕЛЕЙ ДЕФЕКТОВ ПРОЕКТИРОВАНИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Национальный авиационный университет
e-mail: alexander.nechay@livenau.net.

У статті пропонується модель дефекту проектування об'єктно-орієнтованого програмного забезпечення, що будується на основі онтології знань об'єктно-орієнтованого проектування та агрегуючих функцій. Пропонується набір кроків для побудови моделей. Побудовано модель дефекту проектування «God Class» та продемонстровано її застосування для виявлення дефектів у проекті ArgoUML.

В статье предлагается модель дефекта проектирования объектно-ориентированного программного обеспечения, которая строится на основе онтологии знаний объектно-ориентированного проектирования и агрегирующих функций. Предлагается набор шагов для построения моделей. Построена модель дефекта проектирования «God Class» и продемонстрировано ее применения для выявления дефектов в проекте ArgoUML.

The paper proposes an object-oriented software design flaw model, which is based on the ontology of object-oriented design knowledge and aggregating functions. A set of steps for constructing models is proposed. The «God Class» design flaw's model is constructed and its application for flaws detection in the ArgoUML project demonstrated.

Ключевые слова: дефекты проектирования, объектно-ориентированное проектирование, правила проектирования, качество программного обеспечения

Введение

В ходе сопровождения, для удовлетворения все время меняющихся требований, в программное обеспечение вносятся изменения, часто в жестких условиях ограниченных ресурсов. Со временем программное обеспечение становится больше и сложнее, а его конструкция отклоняется от первоначальной проектной модели. Как следствие, система оказывает большее сопротивление новым изменениям, внесение которых в одну часть системы требует значительной предварительной реструктуризации или вызывает побочные эффекты в другой части. Такое явление известно как распад программного обеспечения [1,2], а его основной причиной является накопление проектных решений, нарушающих правила проектирования.

Несоответствие структурных характеристик элемента или фрагмента конструкции программы правилу проектирования называют дефектом проектирования [3].

Таким образом, актуальна задача обнаружения и мониторинга дефектов проектирования. Для решения данной задачи необходимо представление дефекта проектирования как отдельной, имеющей свойства сущности, на основании которой была бы возможность рассуждать о дефекте.

Анализ последних исследований и публикаций

Согласно с [3], различают измеряемые и не измеряемые дефекты проектирования. Поскольку задача обнаружения не измеряемых дефектов достаточно эффективно решается путем статического анализа кода [4,5], далее в статье речь пойдет об измеряемых дефектах проектирования. За последнее десятилетие различные методы были разработаны для обнаружения дефектов проектирования объектно-ориентированного программного обеспечения, в основе которых лежит использование метрик.

В работе [6] определен набор стратегий обнаружения (количественно выраженных

© А.С.Нечай, 2010

правил проектирования) на основе метрик для обнаружения порядка десяти дефектов проектирования уровня метода, класса и пакета. Каждый найденный дефект с помощью стратегии подтверждается инженером путем ручной инспекции. Не смотря на то, что метод полагается на превышение метриками установленных порогов, способ установки значений порогов метрик не рассматривается.

В работе [7] предложен метод обнаружения дефектов проектирования, сущность которого состоит в специфицировании дефектов проектирования с помощью предметно-ориентированного языка программирования. На основе такой спецификации автоматически генерируются алгоритмы обнаружения соответствующих дефектов проектирования. Результатом работы алгоритмов является список обнаруженных дефектов. Каждый найденный дефект так же подтверждается инженером путем ручной инспекции. Метод позволяет пользователю самостоятельно специфицировать дефекты проектирования и выставлять значения порогов для метрик, однако, рекомендации как это делать так же отсутствуют.

Основным недостатком упомянутых методов является проблема, присущая всем методам, которые применяют метрики – выбор порога. Порог разбивает область значений метрики на две подобласти. В зависимости от подобласти, в которую попадает значение метрики, делается заключение о состоянии измеряемой сущности. Например, если измеряется способность к повторному использованию элемента конструкции, возможные результаты измерения находятся в интервале [0,1] и порог определен значением 0.7, то элементы, которые имеют значение измеримого параметра выше этого порога, считаются хорошо приспособленными к повторному использованию. Однако возникают вопросы: почему выбран порог именно 0.7? Почему не 0.5? Действительно ли элемент конструкции программы, имеющий значение способности к повторному использованию 0.68, не приспособлен к повторному использованию по сравнению с элементом, имеющим значение этого параметра равным 0.7? Будет ли иметь смысл такой порог, если анализируемая генеральная совокупность имеет максимальное значение способности к повторному использованию ровным 0.5?

В работе [8] был предложен метод, учитывающий неопределенность процесса обнаружения дефектов проектирования путем применения метода Байса. Сущность метода состоит в построении вероятностной модели дефекта проектирования, путем преобразования существующих правил проектирования, следуя систематическому процессу. Основным недостатком данного метода является необходимость в калибровке модели с помощью данных о реально обнаруженных дефектах в других проектах. Эти данные можно получить исключительно путем ручного поиска дефектов в коде инженерами. Результаты таких поисков сильно зависят от опыта и знаний инженера о программном обеспечении, а так же от его размера.

Таким образом, применение метрик для обнаружения дефектов проектирования представляется перспективным подходом, однако существующие методы в рамках подхода имеют следующие ограничения:

– не предлагают моделей дефектов, которые позволяли бы отслеживать развитие дефектов проектирования;

– ориентируются на превышение значений порогов, задание которых является нерешенной задачей и приводит к неточности обнаружения дефектов проектирования [9];

– отсутствие приоритетности, что вынуждает проводить ручную инспекцию всего массива обнаруженных дефектов, который может быть существенных размеров [10].

Постановка задачи

Целью данной статьи является построение и оценка моделей дефектов проектирования, которые бы смогли заполнить обозначенные в предыдущем разделе пробелы. Для достижения цели ставятся и решаются следующие задачи:

– определить модель дефекта проектирования и ее компоненты;

– сформулировать последовательность и состав шагов, выполнение которых необходимо для построения модели дефекта проектирования;

– построить модель примера дефекта проектирования и оценить ее работоспособность при обнаружении дефектов в существующем программном обеспечении;

Модель дефекта проектирования

Моделью дефекта проектирования назовем описание элемента программного обеспечения,

© А.С.Нечай, 2010

пораженного дефектом. Построение модели выполняется путем построения функций на основе онтологии знаний объектно-ориентированного проектирования и агрегирующих функций. Функции модели позволяют определить степень развития дефекта проектирования [3], а также среднюю интенсивность простых признаков дефекта, поскольку оба эти свойства дефекта проектирования необходимы для проведения его оценки.

Пусть $E = \{e_1, e_2, e_3, ..., e_k\}$ – множество элементов конструкции ПО, а $D = \{d_1, d_2, d_3, ..., d_p\}$ – множество дефектов проектирования, которые могут возникать у элемента конструкции типа $e \in E$. Тогда модель дефекта проектирования состоит из таких функций:

– $\varphi_d : E \to [0,1000]$ – функция для определения степени развития дефекта $d \in D$;

– $\mu_d : E \to [0,1000]$ – функция для определения средней интенсивности простых признаков дефекта $d \in D$.

Для построения модели необходимо выполнить следующие шаги:

– сформулировать правило проектирования элемента конструкции программного обеспечения;

– выполнить анализ правила для определения признаков его нарушения;

– определить метрику для вычисления интенсивности каждого простого признака;

– установить пороговое значение для каждой из метрик;

– построить функции модели дефекта проектирования путем комбинирования функций для определения интенсивностей его признаков.

Формулирование правила проектирования

Определим взаимно однозначное соответствие $violate : R \to D$, где $R = \{r_1, r_2, r_3, ..., r_p\}$. Инженер, основываясь на литературе, посвященной правилам проектирования или на собственном опыте, определяет множество R. Если $d = violate(r)$, то нарушение правила $r \in R$ вызывает дефект проектирования $d \in D$. Примером может быть правило, взятое из работы [11] – «класс не должен быть слишком большим и сложным».

Анализ правила для определения признаков его нарушения

Задачей этого шага является следующее: путем пошаговой детализации правила $r \in R$ выделить для каждого $d = violate(r)$ множество признаков $S_d = \{s_{d,1}, s_{d,2}, s_{d,3}, ..., s_{d,m}\}$ того, что элемент $e \in E$ поражен дефектом $d \in D$. Пусть S является множеством всех признаков, которые могут иметь элементы $e \in E$, тогда $S = \bigcup_{d \in D} S_d$. Определим интенсивность признака как количественную характеристику его проявления.

Признак может быть составной или составленный из других признаков. Простым будем считать составной признак, интенсивность которого возможно вычислить с помощью одной прямой метрики. Условие нарушения правила является составленным признаком (интенсивность этого признака является степенью развития дефекта) с которого начинается процесс пошаговой детализации.

На каждом шаге процесса, признаки, которые признаются составленными, делятся на более простые так, что составленный признак определяется либо одновременным проявлением составных признаков или проявлением хотя бы одного составного признака. Простые признаки дальнейшему делению не подлежат. Пошаговая детализация заканчивается, когда все составленные признаки поделены на составные признаки. Например, признак s_1= «очень сложный и слабо сцепленный e» является составленным, поскольку не может бать оценен одной метрикой. Поделим его на два признака, которые должны проявляться одновременно: s_2 = «очень сложный e» и s_3= «слабо сцепленный e», которые являются простыми, поскольку могут бать оценены метриками Weighted Method Count (WMC) [12] и Tight Class Cohesion TCC [13] соответственно.

Результатом пошаговой детализации является граф признаков дефекта проектирования $d \in D$ $SG = (S_d, is_component_of)$, где $is_component_of \subseteq S_d \times S_d$. Для каждого $s_i, s_j \in S_d$ выражение s_i $is_component_of$ s_j значит, что s_i определяется несколькими признаками, один из которых s_j. Граф признаков изображается в виде дерева (рис. 1). Узлы, которые представляют простые признаки

– терминальные (закрашены). Одной аркой показано, что составленный признак определяется одновременным проявлением составных признаков, а двумя арками – проявлением хотя бы одного составного признака.

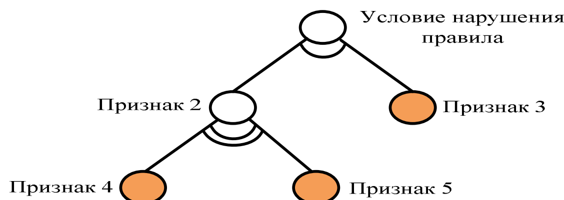

Рис. 1. Граф признаков дефекта проектирования

Определение метрик для вычисление интенсивности простых признаков

Для каждого простого признака $s \in S$ выбрать метрику $M_s : E \rightarrow \Re$, с помощью которой можно оценить интенсивность признака s. Если оценивается признак, для которого невозможно подобрать существующую метрику, тогда необходимо определить новую метрику.

Установка порогового значения для каждой из метрик

Пороговое значение $Z_s \in \Re$ разделяет область значений метрики M_s на две подобласти. В зависимости от того, какой подобласти принадлежит результат измерений, принимается решение о присутствии признака $s \in S$. Существуют следующие основные подходы определения порогов для метрик [14]: использование статистики и использование общепринятой семантики.

При использовании статистики пороги метрик определяются на основе результатов ряда измерений метрики и семантики соответствующих простых признаков. Такой подход необходимо применять в том случае, если простой признак ссылается на большие/малые или на очень большие/малые величины и не устанавливает точного порога. Например, для метрики размера, используемой для оценки интенсивности такого простого признака как «модуль очень большой», только используя статистические данные, можно определить какие значения метрики являются очень большими, а какие нет.

Большинство метрик программного обеспечения имеют не нормальный закон распределения значений, поэтому для обработки ряда измерений рекомендуется использовать статистический метод «ящик с усами» (box-plot) [15] (рис. 2), предназначенный для обнаружения аномальных значений в наборе данных и успешно применяемый для решения задач обеспечения качества программного обеспечения [14].

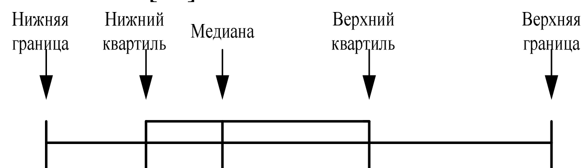

Рис. 2. Ящик с усами

Медиана med делит ранжированный набор данных на две равные части: 50 % «нижних» единиц набора данных будут иметь значение не больше, чем медиана, а «верхние» 50 % – значения не меньше, чем медиана. Верхний квартиль uq – медиана единиц набора данных, имеющих значение больше чем med, а нижний квартиль lq – медиана единиц набора данных, имеющих значение меньше чем med. Длина «ящика» bl - расстояние от верхнего квартиля к нижнему, то есть $bl = uq - lq$. Верхняя ut и нижняя lt границы определяются следующим образом: $ut = uq + 1.5bl$, $lt = lq - 1.5bl$. Полученные значение должны быть урезаны до ближайшей значимой единицы набора данных, чтобы избежать не имеющих смысла понятий, таких как отрицательное количество строк кода. Значения единиц набора данных, превышающие квартили являются аномалиями, а значения, превышающие границы – выбросами. В соответствии с этим, пороговое значение метрики целесообразно устанавливать таким образом:

– если простой признак ссылается на большую величину, то порогом оценивающей его метрики является верхний квартиль;

– если простой признак ссылается на малую величину, то порогом оценивающей его метрики является нижний квартиль;

– если простой признак ссылается на очень большую величину, то порогом оценивающей его метрики является верхняя граница;

– если простой признак ссылается на очень малую величину, то порогом оценивающей его метрики является нижняя граница.

© А.С.Нечай, 2010

Для получения ряда измерений метрики – входных данных, для определения порога на основе статистической информации, используются такие основные подходы:

– измерение элементов выборки из генеральной совокупности программных систем с последующим обобщением результатов статистической обработки, полученных данных на всю генеральную совокупность;

– измерение элементов диагностируемой программной системы.

В зависимости от используемого подхода, полученный в результате статистической обработки ряда измерений порог может быть соответственно обобщенный или относительный. Эмпирическое исследование обобщенных и относительных порогов в работе [16] показывает что:

– пороги должны быть относительны к программному продукту;

– пороги могут сохраняться между разными версиями программной системы.

В связи с этим, в диссертации используются пороги, относительные ко всем версиям диагностируемого программного обеспечения, то есть входные данные для определения порога получаются путем измерения элементов всех версий диагностируемого программного обеспечения.

При использовании общепринятой семантики, пороги метрик определяются на основе общих знаний и семантики соответствующих простых признаков. Такой подход необходимо применять в том случае если простой признак явно ссылается на определенное значение порога или порог может быть определен исходя из общих знаний, наблюдений, требований в рамках конкретного проекта. Например, для метрики максимальной вложенности условных операторов, используемой для оценки интенсивности такого простого признака как «модуль сложный», порог можно определить как граница краткосрочной памяти человека – 7 [17].

Построение функций модели дефекта проектирования

В связи с этим, в диссертации используются пороги, относительные. Задачей данного шага является построение функций модели дефекта проектирования φ_{d} и μ_{d} на основе полученного графа признаков. Для этого необходимо выбрать функции, с помощью которых оценивать интенсивности как простых, так и составленных признаков дефектов проектирования. Выбор функций должен соответствовать следующим критериям:

– простоты – в связи с тем, что полученные модели дефектов проектирования будут использоваться для диагностики больших программных систем и их версий, выбираемые функции должны иметь минимальное количество операций, чтобы минимизировать длительность вычислений;

– соответствия – выбираемые функции должны соответствовать моделируемым сущностям, в данном случае передавать смысл интенсивности признака дефекта проектирования.

Среди критериев нет критерия точности, это связано с тем, что предлагаемый в диссертации метод ориентирован на отслеживание динамики развития дефекта проектирования, а не на его обнаружение или оценку степени развития.

Функции для оценки интенсивности простых признаков. В зависимости от того, является ли порог верхней или нижней границей значений оценивающей метрики, определим две функции. Если порог является верхней границей, то функция определения интенсивности простого признака имеет вид:

$$I_s = HigherThan\ (M_s(e), Z_s) = \frac{M_s(e)}{Z_s} \cdot 100\% \ .$$

Формула не имеет смысла, если порог $Z_s = 0$, в этом случае значение функции принимается равным 1000%.

Если порог является нижней границей, то функция определения интенсивности простого признака имеет вид:

$$I_s = LowerThan\ (M_s(e), Z_s) = \frac{Z_s}{M_s(e)} \cdot 100\% \ .$$

Формула не имеет смысла, если значение метрики $M_s(e) = 0$. В этом случае значение функции принимается равным 1000%. Для обеих функций верхнее значение ограничивается 1000% с целью избегания очень больших значений.

В данном случае, под интенсивностью простого признака понимается процент превышения порога значения метрики в определенном направлении, который передает сущность интенсивности признака как количественную характеристику его проявления. Каждая из функций имеет один

© А.С.Нечай, 2010

основной арифметический оператор, а именно, оператор деления. Таким образом, определенные функции соответствуют критерию соответствия и простоты. Для оценки интенсивности простого признака, разумеется, могут быть использованы и другие функции.

Функции для оценки интенсивности составленных признаков. Комбинирование значений метрик или функций может быть выполнено разными путями. Это может быть неформальный подход, например, используя линейчатую диаграмму, которая показывает сразу несколько метрик. В случае анализа больших и сложных программ, содержащих сотни классов, указанный подход может оказаться неэффективным, поэтому необходим формальный подход, например, определение непрямой метрики с помощью формулы. Так, для определения функции φ_{d} необходимо определять формулы для вычисления интенсивности составленных признаков дефекта путем комбинирования формул для вычисления интенсивности составных признаков. Для данной задачи предлагается использовать агрегирующие функции [18]. Функцию $min\,(I_{s,1}, I_{s,2})$ – нахождение минимального значения аргументов, применим для определения интенсивности составленного признака, если он определяется одновременным проявлением составных признаков (соответствующие узлы графа объединены одной аркой). Функцию $max\,(I_{s,1}, I_{s,2})$ – нахождение максимального значения аргументов, применим для определения интенсивности составленного признака, если он определяется проявлением хотя бы одного из составных признаков (соответствующие узлы графа объединены двумя арками). Данные функции были выбраны в связи с тем, что они передают смысл составленных признаков нарушения правила проектирования, являясь при этом простыми, следовательно, удовлетворяют критериям простоты и соответствия. Процесс комбинирования признаков обратный процессу построения дерева признаков. Первым комбинируются простые признаки, затем, вверх по дереву, комбинируются все остальные признаки, пока не будет достигнут признак, представляемый вершиной дерева.

Результирующая функция является функцией φ_{d}. Для иллюстрации построения функции φ_{d} воспользуемся ранее рассмотренным графом признаков. Пусть метрики $M_{s,3}, M_{s,4}, M_{s,5}$ выбраны для оценки интенсивностей соответственно признаков 3, 4 и 5. Причем, признак 3 проявляется, если метрика $M_{s,3}$ превышает порог $Z_{s,3}$, признак 4 проявляется, если метрика $M_{s,4}$ превышает порог $Z_{s,4}$, признак 5 проявляется, если метрика $M_{s,5}$ меньше порога $Z_{s,5}$. Тогда построение функции начинается из определения формул для оценки интенсивностей простых признаков 3, 4 и 5 (рис.3).

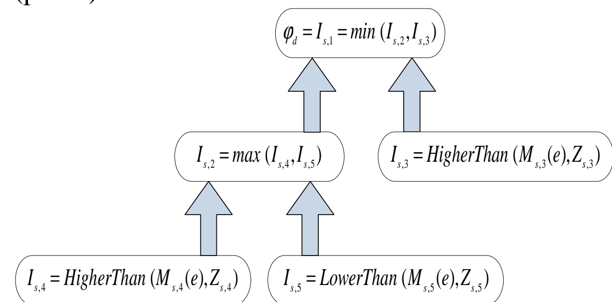

Рис. 3. Пример построения функции φ_{d}

Затем формулы для признаков 4 и 5 комбинируются в формулу для оценки интенсивности признака 2 с помощью агрегирующей функции $max\,(I_{s,4}, I_{s,5})$ (узлы графа признаков 4 и 5 объединены двумя арками). На последнем шаге формулы для признаков 2 и 3 комбинируются в формулу для оценки интенсивности признака 1 с помощью агрегирующей функции $min\,(I_{s,2}, I_{s,3})$ (узлы графа признаков 2 и 3 объединены одной аркой). Полученная функция для оценки интенсивности признака 1 является функцией φ_{d}:

$$\varphi_d(e) = min\,(max\,(HigherThan\,(M_{s,4}(e), Z_{s,4}), LowerThan\,(M_{s,5}(e), Z_{s,5})), HigherThan\,(M_{s,3}(e), Z_{s,3}))$$

Для построения функции μ_{d} применим агрегирующую функцию $mean\,(I_{s,1}, I_{s,2})$ – нахождение среднего арифметического значения аргументов. В примере, рассмотренном выше – три простых признака. Поэтому функция μ_{d} будет иметь следующий вид:

$$\mu_d(e) = mean\,(HigherThan\,(M_{s,4}(e), Z_{s,4}), LowerThan\,(M_{s,5}(e), Z_{s,5}), HigherThan\,(M_{s,3}(e), Z_{s,3}))$$

Построение модели дефекта «God Class»

Для оценки работоспособности модели построим модель дефекта проектирования «God Class» [11]. По сравнению с другими классами, класс, пораженный дефектом «God Class», выполняет значительно больше функций в системе, то есть централизует системную логику в своих методах. Класс с дефектом «God Class» делегирует лишь небольшую часть работы набору простых классов и использует их данные. Данный дефект по своей природе является идентификационным дефектом класса [3]. Правило, несоответствием которому является данный дефект можно сформулировать следующим образом: «класс не должен быть сложным и плохо сцепленным и при этом получать доступ к данным других классов». В результате анализа правила можно выделить признаки его нарушения, которые должны проявляться одновременно: «класс очень сложный», «класс имеет слабое сцепление» и «класс получает доступ к чужим данным» (рис.4).

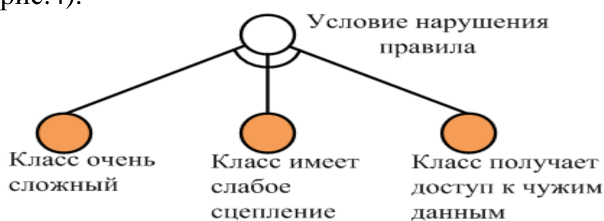

Рис. 4. Граф признаков дефекта «God Class»

Метрики и пороги для оценки простых признаков дефекта приведены в табл.1.

Таблица 1

Таблица признаков дефекта «God Class»

Простой признак	**Метрика**	**Порог**	**Функция для оценки признака**
Класс очень сложный	WMC	Upper tail (36)	*HigherThan*
Класс имеет слабое сцепление	TCC	0,33	*LowerThan*
Класс получает доступ к чужим данным	ATFD	3	*HigherThan*

Для метрики WMC значение порога выставлено на основании того, что соответствующий признак ссылается на «очень большую» величину.

Для нормализованных метрик, таких как TCC, в работе [14] рекомендуется выбирать значение порога из множества {3/4, 2/3, 1/2, 1/3, 1/4} для легкости его отнесения к семантике (например «две трети» или «четверть»). Класс плохо сцепленным можно считать, если значение метрики TCC меньше 0.5, поэтому значением порога можно выбрать одно из двух: «треть» и «четверть». В связи с тем, что модель строится для мониторинга и завышенные значения порога не приведут к пропуску дефекта, а лишь к его уменьшению на фоне других, выбрано более строгое значение порога – «треть», или 0.33. Идеальным значением порога для метрики ATFD [14] можно считать 0, то есть инкапсуляция не нарушается совсем. Однако, исходя из опыта, можно сказать, что доступ к нескольким чужим атрибутам не является проблемой для класса, что и отражено в установленном значении порога.

Таким образом, применив агрегирующие функции для комбинации функций оценки интенсивности простых признаков, получим следующие функции модели дефекта проектирования «God Class»:

$$\varphi_d(e) = min\ (HigherThan\ (WMC(e), ut),\ LowerThan\ (TCC(e), 0.33),\ HigherThan\ (ATFD(e), 3))$$

$$\mu_d(e) = mean\ (HigherThan\ (WMC(e), ut),\ LowerThan\ (TCC(e), 0.33),\ HigherThan\ (ATFD(e), 3))$$

где e – класс объектно-ориентированного программного обеспечения.

Применение модели дефекта «God Class»

Исследование дефектов проектирования «God Class» посредством построенной модели проводилось в инструменте для UML моделирования, написанный на языке программирования Java под названием ArgoUML. С его помощью возможно создавать и сохранять все стандартные UML диаграммы. Инструмент так же имеет возможность выполнять обратное проектирование скомпилированного Java кода и генерировать UML диаграммы на его основе. Исходные коды ArgoUML доступны под BSD Open Source License. ArgoUML выбран благодаря тому, что имеет значительный размер и сложность, известную широкому кругу читателей предметную область (UML-моделирование), а так же доступный исходный код основных и рабочих релизов датируемых с октября 2002 по август 2009 года. Характеристики последней стабильной версии, используемой в исследовании ArgoUML – 0.28.1 от 16.08.09 приведены в табл. 2.

© А.С.Нечай, 2010

Таблица 2

Характеристики ArgoUML

Характеристика	**Значене**
Язык программирования	Java
Количество пакетов	120
Количество классов	2089
Количество методов	17247
Количество строк кода	174212

Были выбраны классы, которые по результатам применения метода из [6] поражены дефектом «God Class». Свойства дефектов данных классов были рассчитаны с применением полученной модели. Всего было обнаружено 60 таких классов. Половина из них, пораженная наиболее развитыми дефектами представлена в табл. 3, значения всех свойств округлены.

Таблица 3

Классы, пораженные дефектами «God Class»

Имя класса	**WMC**	**TCC**	**ATFD**	φ_d	μ_d
CoreHelperMDRImpl	745	0,01	35	1000	1000
CoreFactoryMDRImpl	337	0,03	24	800	912
ModelAccessModelInterpreter	176	0	99	489	830
UMLActivityDiagram	129	0	83	358	786
ModelManagementHelperMDRImpl	128	0,08	20	356	479
AttributeNotationUml	119	0	68	331	777
ArgoDiagramImpl	110	0,01	32	306	769
FacadeMDRImpl	803	0	9	300	767
FigNodeModelElement	294	0,02	9	300	706
StateMachinesFactoryMDRImpl	104	0,05	11	289	439
CommonBehaviorFactoryMDRImpl	101	0,02	19	281	638
SequenceDiagramGraphModel	89	0,03	56	247	749
DeploymentDiagramGraphModel	88	0,13	41	244	499
ExplorerPopup	86	0	78	239	746
ExtensionMechanismsHelperMDRImpl	83	0,1	12	231	320
ImportCommon	83	0,02	44	231	744
UMLDeploymentDiagram	83	0,01	55	231	744
UserDefinedProfile	80	0,07	33	222	564
ModuleLoader2	76	0,09	15	211	359
ToDoList	73	0,09	15	203	357
Modeller	261	0,1	6	200	418
ModelEventPumpMDRImpl	105	0,16	6	200	233
MDRModelImplementation	72	0,02	34	200	733
Main	72	0,16	73	200	469
ProjectBrowser	150	0,04	6	200	481
Modeler	196	0,11	6	200	348
PGMLStackParser	72	0,12	6	200	225
CollaborationsFactoryMDRImpl	67	0,02	19	186	606
UMLMutableGraphSupport	67	0,03	30	186	729
TodoParser	64	0,11	9	178	259

Из таблицы видно, что все классы активно получают доступ к данным других классов, имеют очень большую сложность, и их сцепление близко к нулю. Класс CoreHelperMDRImpl имеет наиболее развитый дефект. При ручном просмотре его кода заметны большие размер и количество его методов. Класс содержит 166 методов и только 2 атрибута, что говорит о том, что методы мало связанны, а класс формирует скорее библиотеку, реализующую интерфейс CoreHelper. Классы в таблице отсортированы по убыванию степени развития их дефектов проектирования. Имея свойства дефектов, поразивших классы, можно расставить приоритеты и начать ручную проверку и возможную реструктуризацию с наиболее «горячих» точек. Так, например, классы TodoParser и CoreHelperMDRImpl оба считаются пораженными дефектами проектирования. Однако, применяя предложенные модели, становится очевидным, что CoreHelperMDRImpl имеет проблемы намного серьезней, и это подчеркивает их размещение в таблице (№1 и №30).

В самом деле, CoreHelperMDRImpl содержит в 6 раз больше методов и почти в 7 длиннее (в терминах количества строк кода). Только один класс CoreHelperMDRImpl имеет размер небольшого приложения – 3,324 строчек кода.

Кроме того, количественно выраженные свойства дефектов позволяют выявлять их на стадиях зарождения и отслеживать их развитие. По результатам такого мониторинга, дефекты, остающиеся стабильными на протяжении своей истории могут не рассматриваться как опасные, поскольку их стабильность говорит о том, что они не создают сложностей при сопровождении, а также работы по сопровождению не ухудшают структуру системы. Таким образом, внимание инженеров по сопровождению не рассеивается на 60 подозрительных классов, а концентрируется на нескольких самых развитых и нестабильных. Использование предложенных моделей в контексте мониторинга описано в работе [19].

Заключение

В статье предложена модель дефекта проектирования объектно-ориентированного программного обеспечения, состоящая из двух функций и реализующая идею о представлении дефекта как самостоятельной сущности.

© А.С.Нечай, 2010

Предложены критерии подбора функций для комбинирования признаков дефекта.

Дальнейшие изыскания в данном направлении могут быть направлены на уточнение и добавление новых критериев подбора функций, которые могли бы обеспечить выбор функций более точно отражающих сущность моделируемых свойств дефекта и не требующих ограничивать области значений (в работе мы ограничиваем область значений [0, 1000]).

Безусловно, признаки дефектов проектирования могут иметь разный вес при определении степени развития дефекта, поэтому модель можно расширить весовыми коэффициентами для отображения данного факта. Способ получения таких коэффициентов и их практическую значимость еще предстоит установить.

Литература

1. *Lehman, M. M.* On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle / M.M. Lehman // *The Journal of Systems and Software*. – 1980. – vol. 1. – P. 213-221.
2. *Izurieta C.* How Software Designs Decay: A Pilot Study of Pattern Evolution / Clemente Izurieta, James M. Bieman // Proceedings of the First International Symposium on Empirical Software Engineering and Measurement (ESEM'07), September 20-21 2007. – Washington, 2007. – P. 449-451.
3. *Нечай О.С.* Методи та засоби виявлення дефектів проектування об'єктно-орієнтованого програмного забезпечення / О.С. Нечай, М.О. Сидоров // Вісник НАУ. – 2009. – №3. – С. 200-205.
4. *Ciupke O.* Automatic detection of design problems in object-oriented reengineering / Oliver Ciupke // Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS'99). – Washington: IEEE Computer Society, 1999. – P. 18–32.
5. *Hovemeyer D.* Finding bugs is easy / David Hovemeyer, William Pugh // ACM SIGPLAN Notices. – 2004. – Vol.39, No.12. – P.92-106.
6. *Marinescu R.* Measurement and Quality in Object-Oriented Design: Ph.D thesis / R. Marinescu. – "Politehnica" University of Timisoara, 2002. – 155 p.
7. *Moha N.* A Domain Analysis to Specify Design Defects and Generate Detection Algorithms / Naouel Moha, Y. Guéhéneuc, F. Le Meur, L. Duchien // Proceedings of the 11th Intern. Conf. on Fundamental Approaches to Software Engineering. – Springer-Verlag, March-April 2008. – P. 276–291.
8. *Khomh F.* A Bayesian Approach for the Detection of Code and Design Smells / F. Khomh, S. Vaucher, Y.-G. Gu´eh´eneuc, H. Sahraoui // Proceedings of the 9th International Conference on Quality Software. – Washington : IEEE Computer Society, 2009. – P. 302-313.
9. *Mantyla M.* Bad Smells " Humans as Code Critics / Mika V. Mantyla, Jari Vanhanen, Casper Lassenius // in Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04). – Washington: IEEE Computer Society, 2004. – P. 399–408.
10. *Kim S.* Which warnings should I fix first? / Sunghun Kim, Michael D. Ernst // Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE'07). – New York: ACM, 2007. – P. 45–54.
11. *Riel A.* Object Oriented Design Heuristics/ Arthur J. Riel. – Addison-Wesley Professional, 1996 . – 400 p.
12. *Chidamber S.* A metrics suite for object oriented design / S.R. Chidamber, C.F. Kemerer // IEEE Transactions on Software Engineering. – 1994. – Vol.20, No. 6. – P. 476-493.
13. *Bieman J.M.* Cohesion and reuse in an object-oriented system / J.M. Bieman, B.K. Kang // Proceedings of ACM symposium on Software Reusability. – New York : ACM Press, 1995. – P. 259-262.
14. *Lanza M.* Object-Oriented Metrics in Practice / Michele Lanza, Radu Marinescu. – Berlin : Springer-Verlag, 2006 . – 206 p.
15. *Chambers J.* Graphical Methods for Data Analysis / John M. Chambers. – Chapman & Hall/CRC, 1983 . – 336 p.
16. *Crespo Y.* Relative Thresholds: Case Study to Incorporate Metrics in the Detection of Bad Smells / Y. Crespo, C. L´opez, R. Marticorena // Proceedings of 10th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering. – Lugano : Universit´a della Svizzera italiana Press, 1996. – P. 109-118.
17. *Pinker S.* How the Mind Works / Steven Pinker. – W.W. Norton & Co, 1999. – 672 p.
18. Calvo T. Aggregation Operators: New Trends and Applications / Tomasa Calvo, Gaspar Mayor, Radko Mesiar. – Physica-Verlag Heidelberg, 2002. – 352 p.
19. Нечай О.С. Метод діагностики об'єктно-орієнтованого програмного забезпечення / О.С. Нечай // Вісник НАУ. – 2009. – №5. – С. 100–111

Сведения об авторе

Нечай Александр Сергеевич - ассистент каф. инженерии программного обеспечения Национального авиационного университета научное направление – проектирование и эволюция программного обеспечения.
e-mail: alexander.nechay@livenau.net.

Стаття надійшла до редакції 15.03.2010

© А.С.Нечай, 2010