ВЕСТНИК ТОМСКОГО ГОСУДАРСТВЕННОГО УНИВЕРСИТЕТА
2015 Управление, вычислительная техника и информатика № 2 (31)

ОБЗОР

УДК 004.415.2
DOI 10.17223/19988605/31/9

О.А. Змеев, Л.С. Иванова

ПОИСК АРТЕФАКТОВ ПРОЕКТИРОВАНИЯ. ОБЗОР ПОДХОДОВ

Рассмотрены подходы, предложенные исследователями разных стран, для обнаружения паттернов, антипаттернов и недостатков проектирования в различных источниках информации о системе. Проведена систематизация рассмотренных подходов по ряду критериев. Приведен перечень существующих реализаций.
Ключевые слова: паттерн; антипаттерн; анализ данных; UML.

С развитием информационных технологий вопрос качества программного обеспечения становится все более острым. Проектируемые системы становятся все больше и сложнее, что затрудняет как добавление нового функционала, так и поиск и исправление различных ошибок. Автоматизация процесса поиска аномалий, антипаттернов и недостатков способствует улучшению качества разрабатываемого программного обеспечения (ПО), существенному уменьшению времени и стоимости работ. Кроме того, анализ разрабатываемого ПО на наличие различных шаблонов проектирования, отсутствие ошибок позволяет более объективно оценить квалификацию разработчика.

В данной статье проводится обзор литературы, посвященной проблеме обнаружения различных артефактов (паттернов, недостатков, ошибок, антипаттернов и др.) в исходном коде ПО, документации и других источниках информации. Осуществляется систематизация найденных подходов по ряду признаков.

Первый раздел посвящен перечню объектов для анализа. Во втором разделе рассмотрены различные источники информации для анализа. Третий раздел содержит перечень наиболее популярных методов обнаружения артефактов. В четвертом разделе проведен обзор существующих реализаций рассмотренных подходов.

1. Объекты анализа

Подходы, предлагаемые для поиска артефактов, можно классифицировать по целому ряду признаков. Начать рассмотрение следует с классификации по объектам анализа. В качестве цели для поиска могут выступать следующие артефакты:

1. Паттерны проектирования – архитектурные решения, представляющие собой решение некоторой часто возникающей проблемы проектирования в типичном контексте. Например, реализация паттерна Декоратор (Decorator) [1] позволяет посредством оборачивания динамически изменять функционал объекта без порождения громоздкого набора подклассов.

Среди авторов, исследования которых посвящены анализу кода и / или документации, поиск паттернов проектирования является наиболее популярной темой [2–17].

Обнаружение паттернов в исходном коде является важной частью обратного проектирования. Автоматизация данного процесса позволяет существенно увеличить качество собранной информации и уменьшить затраты, как временные, так и материальные.

Авторы статьи [3] предложили подход как для поиска реализаций паттернов, так и для их проверки на соответствие стандартному представлению [1].

2. Антипаттерны, нарушения принципов проектирования, недостатки кода.

Антипаттерны – неудачные архитектурные решения часто возникающих проблем проектирования. В качестве примера можно упомянуть Божественный объект (God object, Blob) [18]. Создание объекта с очень широким функционалом противоречит принципу ООП «разделяй и властвуй» [19].

Поиску данных артефактов посвящены исследования [20–27]. Первый подход [26] по этой тематике подразумевал ручной поиск недостатков в UML-диаграммах [28], в более современных исследованиях используется автоматизированный поиск.

Обнаружение антипаттернов и других недостатков проектирования позволяет повысить качество разрабатываемых приложений. Кроме того, антипаттерны могут быть причиной ошибок, которые не поддаются обнаружению с помощью классических методов тестирования.

3. Микропаттерны.

Поиск паттернов проектирования в автоматическом режиме является нетривиальной задачей в силу своей сложности и нечеткой формализации условий, идентифицирующих их. Это может приводить к ошибочным результатам при попытках распознавания.

С другой стороны, если опуститься на более низкий семантический уровень, который приближен к конкретной реализации (технологии), то паттерны могут быть описаны в четких терминах данной технологии. Подобного рода паттерны были названы отслеживаемыми (англ. traceable patterns) [29]. Данные паттерны могут покрывать различные по величине модули, начиная с фрагментов кода, заканчивая пакетами (в контексте модулей Java). Отслеживаемые паттерны, ограничивающиеся рамками одного класса / интерфейса, были именованы в [29] как микропаттерны.

Авторами статьи [29] был выдвинут набор микропаттернов, которые можно описать простыми средствами, предлагаемыми языком программирования Java. Данный набор представляет собой основные способы проектирования отдельных классов, которые используются разработчиками. В качестве примера можно привести паттерн Запись (англ. Record) – класс, в котором все поля имеют модификатор доступа public и не имеют ни одного объявленного метода.

Это, в свою очередь, даёт фундамент для определения более сложных конструкций, область которых выходит за рамки конкретного класса.

2. Исходные данные для анализа

Вторым основанием для классификации подходов для обнаружения артефактов является источник данных для проведения анализа. В литературе выделяют три типа анализа:

1. Статический анализ – анализ исходного кода и документации.

Статический анализ можно разделить на ряд этапов: на первом этапе осуществляются семантический разбор источника и генерация некоторого внутреннего представления классов и отношений между классами. Затем данное представление анализируется (методы анализа представлены в разделе 3) на наличие искомых артефактов. На финальном этапе результаты анализа передаются пользователю.

Большинство исследований, посвященных DPD (Design Pattern Detection – обнаружение паттернов проектирования), основано на статическом анализе. Методики анализа исходного кода приложения описаны в литературных источниках [2, 4, 6–12, 14, 16, 17, 20–25, 27, 29]. В качестве языка программирования для написания исходного кода наиболее популярными являются Java и C++. Канадскими исследователями был изучен вопрос распознавания паттернов в системах на языке Эйфель в статье [17]. Для тестирования реализаций большинство исследователей использовали известные библиотеки с открытым исходным кодом на соответствующем языке.

Главным недостатком данного подхода является ориентированность на определенный язык программирования. Исходный код приложений, написанный на других популярных языках программирования (C#, Objective C, PHP и др.), не может быть проанализирован с помощью инструментов, предназначенных для анализа исходного кода на языке Java.

Положительной стороной данного подхода является возможность проведения анализа на любом этапе разработки программного обеспечения, подразумевающем наличие исходного кода.

Некоторые подходы подразумевают использование UML-диаграмм [28] в качестве документации для анализа. Примеры методик анализа диаграмм описаны в источниках [3, 13, 15, 26]. Анализ диаграмм UML позволяет устранить ориентированность на определенный язык, однако необходимо поддерживать диаграммы в актуальном состоянии и отображать на них максимально возможное количество информации. Кроме того, необходимо учитывать формат хранения данных, использующийся в case-инструментах для создания UML-диаграмм.

2. Динамический анализ – анализ приложения во время выполнения.

В отличие от статического анализа, динамический подразумевает анализ поведения приложения в процессе работы. Происходит сбор информации о сообщениях, посылаемых между объектами, с учетом времени. На основе этих данных строится внутреннее представление, в котором осуществляется поиск артефактов по поведенческим признакам.

Достоинством динамического анализа является способность распознавать артефакты, одинаковые статически, но разные по поведению. Пример динамического анализа описан в работе [5].

3. Комбинированный анализ – комбинация статического и динамического анализа.

Данному виду анализа посвящены исследования [5]. Статический анализ исходного кода применяется для выявления «кандидатов», динамический анализ – для проверки «кандидатов» на соответствие тому или иному паттерну проектирования. Данный подход более точен, однако объединяет достоинства и недостатки двух вышеописанных подходов.

3. Методы обнаружения артефактов

В исследуемых подходах предлагаются следующие стратегии обнаружения:

1. Метрики программного обеспечения.

Метрикой ПО называют меру, позволяющую получить численное значение некоторого свойства ПО. В качестве примера рассмотрим метрику, используемую авторами статьи [23], для обнаружения антипаттерна Божественный класс (God object) – взвешенное число методов (Weighted Method Count). Формула данной метрики

$$WMC = \sum_{i=1}^{n} c_i \,, \tag{1}$$

где c_i – цикломатическое число Маккейба (2). Для каждого метода строится ориентированный граф, при этом вершины графа соответствуют участкам кода с последовательными вычислениями без операторов ветвления и цикла, дуги соответствуют ветвям выполнения программы. Каждая вершина должна быть достижима из начальной, конечная вершина достижима из любой другой вершины.

$$c_i = e_i - n_i + 2p_i \tag{2}$$

где e_i – количество дуг графа, построенного для i-го метода, n_i – количество вершин такого графа, p_i – число компонент связности такого графа.

Принадлежность класса к антипаттерну Божественный объект авторы [23] определяют по следующему правилу:

$$GodClass(S) = S' \left| \begin{array}{c} S' \subseteq S, \forall C \in S' \\ (WMC(C), TopValues(25\%)) \wedge \\ \wedge (ATFD(C), HigherThan(1)) \wedge (TCC(C), BottomValues(25\%)), \end{array} \right.$$

где *ATFD(C)* – число классов, атрибуты которого запрашиваются в методах класса *C*, *TCC(C)* – относительное число непосредственно связанных методов класса *C*. *(WMC(C), TopValues(25%))* означает, что значение *WMC(C)* должно входить в 25% наибольших значений *WMC* среди всех классов множества *S*. *HigherThan(1)* означает, что *ATFD(C)* должен быть больше или равен 1. *(TCC(C), BottomValues(25%))* означает, что значение *TCC(C)* должно входить в 25% наименьших значений *TCC* среди всех классов множества *S*.

Инструмент для анализа MARPLE [2] также использует метрики при анализе исходного кода и выявлении паттернов.

2. Графы.

В тематике DPD графы чаще всего используются для представления связей между классами, объектами классов. В некоторых исследованиях [11, 24] используется абстрактный синтаксический граф (Abstract Syntax Graph – ASG [30]) как способ представления информации о системе.

2.1 Мера схожести.

Примером использования графов в обнаружении артефактов является подход, предложенный греческими исследователями Николасом Цанталисом и др. [16]. В данном подходе для обнаружения паттернов проектирования используется мера схожести между графами. В качестве узлов графа выступают классы, в качестве ребер – отношения между классами (ассоциации, генерализации и др.). Для вычисления схожести между графами используется следующий метод: графы, соответствующие искомому паттерну (G_A с n_A вершин) и рассматриваемому подмножеству классов (G_B с n_B вершин), представлены в виде матриц смежности. Данные матрицы обрабатываются с помощью специального итеративного алгоритма: Z_0 матрица $n_B \times n_A$, заполненная единицами; на каждой итерации значение вычисляется по формуле (3) до тех пор, пока результат не сойдется.

$$Z_{k+1} = \frac{BZ_k A^T + B^T Z_k A}{\left\| BZ_k A^T + B^T Z_k A \right\|_1}, \tag{3}$$

где A, B – матрицы смежности графов G_A и G_B соответственно.

Результат работы алгоритма – матрица S, равная последнему значению Z_k. Элемент s_{ij} описывает схожесть вершины j из G_A с вершиной i из G_B. Затем все матрицы S, полученные для каждого типа отношений, суммируются, результат нормируется. Схожесть искомого паттерна и рассматриваемого подмножества классов определяется экспертной оценкой на основе полученных данных.

Обнаружение подобия между графами позволяет находить паттерны, отличающиеся от их стандартного представления [1]. Кроме того, авторы заявляют, что их подход обнаруживает иерархии паттернов.

2.2. Максимальный изоморфный подграф, дерево решений.

Нахождение подобия между графами может быть сведено к задаче нахождения максимального изоморфного подграфа. Сложность поиска повышается в силу перебора всех возможных изоморфических перестановок для каждого подграфа. В качестве оптимизации авторы статьи [13] используют дерево принятия решений, проход по которому может породить любую возможную перестановку подграфа. Далее вместо полного набора изоморфных перестановок подграфов необходимо использовать соответствующее каждому подграфу дерево решений. Таким образом, для того чтобы определить сходство паттерн-графа и некоторого подграфа системы-графа, необходимо пройтись по соответствующему дереву решений и получить ответ о возможности порождения данным деревом перестановки подграфа, равного паттерн-графу.

2.3. Максимальный изоморфный подграф, генетический алгоритм.

Использование графов и генетического алгоритма для обнаружения паттернов предложили индийские исследователи Р. Синх Рао и М. Гупта [15]. В качестве источника данных они используют UML-диаграммы классов, которые преобразуются в графы по специальным правилам: вводятся дополнительные отношения наследования и агрегации по принципу транзитивности; строится граф, вершины которого соответствуют классам, ребра – отношениям между классами. Для каждой вершины определяется вектор $t = (t_1,t_2,t_3,t_4)$, строящийся по правилам: t_1=1, если соответствующий класс связан ассоциацией с другим классом, иначе 0; t_2=1, если соответствующий класс связан агрегацией с другим классом. Значение t_3 определяется наличием отношения наследования, t_4 – зависимостью. Аналогичные вектора $e = (e_1,e_2,e_3,e_4)$ вводятся для ребер графа (значения зависят от типа отношения, представленного данным ребром). Хромосома для генетического алгоритма – матрица C размерности $n \times m$, где n – число вершин графа, соответствующего паттерну, m – число вершин графа, соответствующего рассматриваемому подмножеству классов. Хромосомы для первой итерации строятся случайным образом. Функция приспособленности имеет вид

$$F = F_{nc} + F_{ec},$$

$$F_{nc} = |t_1 - t'_1| + |t_2 - t'_2| + |t_3 - t'_3| + |t_4 - t'_4|,$$

$$F_{ec} = |e_1 - e'_1| + |e_2 - e'_2| + |e_3 - e'_3| + |e_4 - e'_4|.$$

Скрещивание происходит путем соединения двух матриц (некоторых столбцов и строк). Мутация заключается в случайной перестановке элементов матрицы из одной строки в другую. Отбор осуществляется по минимальным значениям функции приспособленности. Результатом работы генетического алгоритма является матрица C – матрица соответствия вершины i графа-паттерна и j – графа-подсистемы.

Данный подход предложен впервые, рабочий прототип находится в разработке, поэтому пока нет никаких данных о результатах тестирования.

3. Визуальный анализ.

В качестве полуавтоматического метода обнаружения артефактов в статье [21] был предложен подход, в котором большое количество табличных данных наглядно представляются через визуальные атрибуты геометрического объекта (цвет, высота, поворот прямоугольного столбика и т.д.). Над данными геометрическими объектами могут проводиться различные операции (фильтрация, группировка и т.д.) для облегчения принятия окончательного решения экспертами.

4. Байесовская сеть доверия.

Авторы статьи [22] используют байесовскую сеть для обнаружения антипаттернов. Задача поиска сводится к задаче классификации с двумя возможными исходами: антипаттерн и неантипаттерн. В качестве входных данных используются вектора значений ряда метрик (число объявленных методов, число атрибутов и др.). Для формирования сети, соответствующей искомому антипаттерну, авторы используют методику Goal Question Metric [31]. Методика подразумевает разделение шагов на три уровня: концептуальный (цели), операционный (вопросы), уровень качества (метрики). На концептуальном уровне необходимо определить объект поиска. Операционный уровень вводит ряд вопросов, использующихся для определения объекта поиска, например симптомы антипаттернов. Уровень качества подразумевает ответы на вопросы в измеряемом виде, например метрики для измерения свойств программного обеспечения. На основании данной информации строится сеть доверия, причем входные вершины соответствуют вопросам операционного уровня, выходная вершина содержит вероятность того, что класс является антипаттерном.

5. Java-аннотации.

В статье [14] для обнаружения паттернов проектирования предложено использовать Java-аннотации. По мнению авторов, такие аннотации, как @abstract, @instantiation и другие, могут указывать на реализации паттернов Одиночка, Адаптер и др. Для обнаружения возможных кандидатов в вышеупомянутой статье также применяются регулярные выражения и SQL-запросы.

6. Предикаты.

Применение предикатов для принятия решения о наличии паттерна рассмотрено в источниках [4, 6, 8, 9]. Принадлежность к определенному паттерну определяется посредством комплексной формулы логики предикатов. Исследуемые входные данные (какого-либо типа, например исходный код, структура диаграммы классов UML и т.д.) представляются как набор термов, подставляемых в целевую формулу. На выходе вполне естественный ответ о принадлежности исходного набора термов области истинности целевого предиката. Подход удобен с точки зрения анализа и реализации своей четкой формализацией требований, однако этот же аспект создает существенные проблемы на этапе формирования этих требований. К примеру, в статье [8] для решения подобных проблем искомые паттерны и анализируемый исходный код преобразуются в конструкции на языке OWL (Web Ontology Language [32]).

7. Операции с битовыми массивами.

Необычный способ поиска паттернов в исходном коде предложили исследователи из Монреаля [10]. Они используют итеративный алгоритм, который с помощью побитовых логических операций над векторами обнаруживает реализацию паттерна в приложении. Для исходного кода создается UML-подобное графическое представление, на основе которого строится ориентированный граф (вершины – классы, ребра – отношения между классами). Полученный граф автоматически достраивается до эйлерова графа, строится минимальный эйлеров цикл. На основе эйлерова цикла формируется строчное представление исходной системы, которое преобразуется в набор битовых векторов. Аналогичные операции проводятся для искомых паттернов. Логические операции над набором векторов системы и набором векторов искомого паттерна позволяют обнаружить реализацию паттерна в системе.

8. Разбор текстового представления визуального языка

Метод, описанный в статье [7], делится на два этапа. Первый этап заключается в следующем: для исходного кода строится представление на специальном визуальном языке, отражающем отношения между классами. Это представление преобразуется в строковое выражение. Паттерн задается определенной грамматикой, которая определяет некоторое допустимое подмножество возможных порождаемых выражений. Проверяется принадлежность текущего выражения данному подмножеству на основе LR-анализа. Принадлежность говорит о возможном существовании паттерна в рассматриваемой подсистеме. Пример грамматики для паттерна Адаптер (Adapter) приведен ниже:

a) *AdapterPattern* $\rightarrow$ 1_2 INHERITANCE 1_1 *Adapter* 1_1 ASSOCIATION 2_1 *Adaptee*

b) *Target* $\rightarrow$ CLASS

Δ:{$Target_1$ = $CLASS_1$}

c) *Adapter* $\rightarrow$ CLASS

Δ:{$Adapter_1$ = $CLASS_1$}

d) *Adaptee* $\rightarrow$CLASS

Δ:{$Adaptee_1$ = $CLASS_1$}

Правила порождения b – d определяют нетерминальные *Target*, *Adapter*, *Adaptee* как терминальный символ CLASS, правило a определяет паттерн Адаптер как нетерминальный *Target*, связанный терминальным символом INHERITANCE с нетерминальным *Adapter*, который, в свою очередь, связан терминальным ASSOCIATION с нетерминальным *Adaptee*.

На втором этапе осуществляется проверка участков исходного кода – кандидатов, обнаруженных на первом этапе. Данный подход, как утверждают авторы, зарекомендовал себя для обнаружения структурных паттернов.

Помимо вышеперечисленных, исследователями предложены различные методы ручного поиска артефактов [26].

4. Существующие реализации

На основе некоторых из описанных подходов [2–8, 11, 12, 14, 17, 22, 23, 25, 27] авторами были разработаны соответствующие реализации. Большинство из них являются плагинами для IDE Eclipse и осуществляют анализ исходного кода на языке Java. Для тестирования таких плагинов использовались открытые исходные коды популярных фреймворков (JHotDraw и др.). Таким образом, сравнение качества и скорости обнаружения артефактов возможно только для нескольких реализаций.

Автор инструмента для обнаружения паттернов PDE [5] в своей диссертации провел сравнение инструмента с такими аналогами, как FUJABA [11] и PINOT [12]. Помимо того факта, что аналоги не поддерживают часть паттернов GoF [1], они продемонстрировали худшие результаты при распознавании таких паттернов, как Прототип (Prototype), Компоновщик (Composite) и Состояние (State) [1].

Инструмент DRT [14] в сравнении с PINOT [12] выдал практически одинаковые результаты, выигрывая в определении Фабричного метода и проигрывая для паттерна Посетитель. Сравнение с FUJABA [11] продемонстрировало полное превосходство DRT [14].

Подходы для обнаружения антипаттернов также имеют несколько реализаций. Разработчики инструмента BDTEX [22] утверждают, что их приложение обнаруживает реализации антипаттерна Блоб (Blob, God object) [18] лучше, чем инструмент Detex [25].

Помимо плагинов для анализа исходного кода, в сети Интернет доступна реализация подхода для рефакторинга паттернов проектирования в UML-диаграммах [3]. Данная реализация встроена в инструмент для создания UML-диаграмм ArgoUML.

Проблема тестирования и сравнения различных реализаций DPD-подходов стала настолько острой, что итальянскими исследователями [33] было создано веб-приложение для сравнительного анализа инструментов обнаружения паттернов. На данный момент в нем содержится девять Java-фреймворков как источников данных для поиска и четыре инструмента анализа для сравнения.

Заключение

Анализ литературы показал, что тематика DPD достаточно популярна. Написано большое количество печатных работ, предложена масса подходов для поиска не только паттернов, но и различных недостатков. Разработано несколько рабочих решений, позволяющих проводить анализ исходного кода. Также существует решение для анализа UML-диаграмм.

Предложенные исследователями методики имеют как достоинства, так и недостатки. Основным недостатком большинства предложенных подходов является зависимость от конкретного языка программирования. Необходимо проводить работу в области анализа UML-диаграмм и других независимых от языка реализации источников информации.

ЛИТЕРАТУРА

1. *Gamma E., Helm R., Johnson R., Vlissides J.* Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
2. *Arcelli Fontana F., Zanoni M.* A tool for design pattern detection and software architecture recognition // Information Sciences. 2011. V. 181. P. 1306–1324.
3. *Bergenti F., Poggi A.* Improving UML Designs Using Automatic Design Pattern Detection // Proc. 12th. International Conference on Software Engineering and Knowledge Engineering. 2000. URL:http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.3764
4. *Beyer D., Lewerentz C.* CrocoPat: efficient pattern analysis in object-oriented programs // Proceedings of the International Workshop on Program Comprehension (IWPC'03). 2003. P. 294–295.
5. *Birkner M.* Object-oriented design pattern detection using static and dynamic analysis in java software / MB-PDE Java Software Design Pattern Detection Engine. URL: https://mb-pde.googlecode.com/files/MasterThesis.pdf
6. *Blewitt A., Bundy A., Stark I.* Automatic verification of design patterns in Java // ASE '05 Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. ACM, New York, 2005. P. 224–232.
7. *De Lucia A., Deufemia V., Gravino C., Risi M.* Design pattern recovery through visual language parsing and source code analysis // The Journal of Systems and Software. 2009. No. 82. P. 1177–1193.
8. *Dietrich J., Elgar C.* Towards a web of patterns // Web Semantics: Science, Services and Agents on the World Wide Web. 2007. No. 5(2). P. 108–116.
9. *Fabry J., Mens T.* Language-independent detection of object-oriented design patterns // Computer Languages, Systems & Structures. 2004. V. 30. P. 21–33.
10. *Gueheneuc Y., Hamel S., Kaczor O.* Efficient identification of design patterns with bit-vector algorithm // Proceedings of European Conference on Software. Maintenance and Reengineering (CSMR'06). Bari, 2006. P. 175–184.
11. *Towards* pattern design recovery / J. Niere et. al // Proceedings of International Conference on Software Engineering (ICSE'02). Orlando, 2002. P. 338–348.
12. *Olsson, R., Shi, N.* Reverse engineering of design patterns from java source code // Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE'06). Tokyo, 2006. P. 123–134.
13. *Pande A., Gupta M., Tripathi A.K.* A decision tree approach for design patterns detection by subgraph isomorphism // Communications in Computer and Information Science. 2010. V. 101. P. 561–564.
14. *Rasool G., Philipow I., Mader P.* Design pattern recovery based on annotations // Advances in Engineering Software. 2010. V. 41. P. 519–526.
15. *Singh Rao R., Gupta M.* Design Pattern Detection by Multilayer Neural Genetic Algorithm // International Journal of Computer Science and Network. 2014. No. 3(1). P. 9–14.
16. *Tsantalis N., Chatzigeorgiou A., Stephanides G., Halkidis S.T.* Design pattern detection using similarity scoring // IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. 2006. No. 32(11). P. 896–909.
17. *Wang W., Tzerpos V.* DPVK – an eclipse plug-in to detect design patterns in Eiffel systems // Electronic Notes in Theoretical Computer Science. 2004. V. 107. P. 71–86.
18. *Riel A.J* Object-Oriented Design Heuristics. Addison-Wesley, 1996.
19. *Top* Down Design in An Object Oriented World // University of SAN FRANCISCO. Department of computer science. URL: http://www.cs.usfca.edu/~parrt/course/601/lectures/top.down.design.html
20. *Christopoulou A., Giakoumakis E.A., Zafeiris V.E., Soukara V.* Automated refactoring to the Strategy design pattern // Information and Software Technology. 2012. No. 54. P. 1202–1214.
21. *Dhambri K., Sahraoui H., Poulin P.* Visual detection of design anomalies // Software Maintenance and Reengineering. 2008. P. 279–283.
22. *Khomh F., Vaucher S., Gueheneuc Y.-G., Sahraoui H.* BDTEX: a cgm-based Bayesian approach for the detection of antipatterns // The Journal of Systems and Software. 2011. No. 84. P. 559–572.
23. *Marinescu R.* Detection strategies: metrics-based rules for detecting design flaws // Software Maintenance. 2004. P. 350–359.
24. *Meyer M.* Pattern-based reengineering of software systems // WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering. Washington, 2006. P. 305–306.

25. *Moha N., Gueheneuc G., Duchien L., Le Meur A.F.* Décor: a method for the specification and detection of code and design smells // Software Engineering, IEEE Transactions on. 2009. No. 36(1). P. 20–36.
26. *Travassos G., Shull F., Fredericks M., Basili V.R.* Detecting defects in object-oriented designs: using reading techniques to increase software quality / Computer Science. Universuty of Maryland. URL: http://www.cs.umd.edu/projects/SoftEng/ESEG/ papers/oopsla99.pdf
27. *Wieman R.* Anti-pattern Scanner: an approach to detect anti-patterns and design violations / The Software Evolution Research Lab. URL: http://swerl.tudelft.nl/twiki/pub/Main/PastAndCurrentMScProjects/Thesis_RubenWieman2011.pdf
28. *OMG* Unified Modeling Language (OMG UML), Infrastructure / Documents Associated With Unified Modeling Language (UML), V.2.4.1. UML: http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/
29. *Gil J., Maman I.* Micro Patterns in Java Code // OOPSLA '05 Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. ACM, New York, 2005. P. 97–116.
30. *Abstract* semantic graph // Wikipedia – free encyclopedia. URL: http://en.wikipedia.org/wiki/Abstract_semantic_graph
31. *Basili R., Weiss D.M.* A methodology for collecting valid software engineering data // IEEE Transactions on Software Engineering. 1984. No. 10 (6). P. 728–738.
32. *Web* Ontology Language // Wikipedia – free encyclopedia. URL: http://en.wikipedia.org/wiki/Web_Ontology_Language
33. *Arcelli F., Caracciolo A., Zanoni M.* A Benchmark for Design Pattern Detection Tools: a Community Driven Approach // Special theme: Evolving Software. 2012. No. 88. P. 32.

Змеев Олег Алексеевич, д-р физ.-мат. наук, профессор. E-mail: ozmeyev@gmail.com
Иванова Лидия Сергеевна. E-mail: lida@redlg.ru
Томский государственный университет Поступила в редакцию 14 апреля 2015 г.

Zmeev Oleg A., Ivanova Lidia S. (Tomsk state university, Russian Federation).
Design artifacts detection. Review of the approaches.
Keywords: design pattern; antipattern; data analysis; UML.

DOI 10.17223/19988605/31/9

In this paper, the review of the literature dedicated to the detection of the different artifacts (patterns, defects, errors, antipatterns and etc.) in software source code, documentation and other information sources is given. The systematization of the approaches on several grounds is achieved.

In the study approaches the following artifacts can be used as the purpose for the search:

1. Design patterns.

Patterns detection in the source code is important part of the reverse design. The automation of this process makes it possible to significantly increase the quality of the collected information and to decrease both time and material costs.

2. Antipatterns, design principle violations, code defects.

Antipatterns and other design defects detection makes it possible to increase the quality of developed applications.

3. Micro pattern is traceable patterns, which are limited by the one class/interface. Their detection creates foundation for determining more complex constructions.

Secondly, the classification of approaches is based on the data source for conducting the analysis. There are 3 types of the analysis in the literature:

1. The static analysis is the analysis of the source code and documentation.

Programming languages Java and C++ are the most popular for source code writing. Also, some researchers use UML-diagrams as documentation for analysis.

2. The dynamic analysis is that of application in runtime. Java-bytecode is used as the source data.

3. The combined analysis is combination of static and dynamic analysis. The static analysis of the source code is used for the detection of “candidates”, dynamic analysis is used for checking “candidates” if they correspond to one or another design pattern.

The following detection strategies are proposed in the study approaches:

1. Metrics.
2. Graphs.
3. Visual analysis.
4. Bayesian Belief Networks.
5. Java-annotations.
6. Predicates.
7. Bit arrays operations.
8. Analysis of textual representation of visual language.

Besides those enumerated above, researchers proposed the different methods of the manual detection of artifacts.

The corresponding realizations were developed by authors on the base of several described approaches. The most of them are IDE Eclipse plugins and analyses Java source code. Open source code of popular frameworks (JHotDraw and etc.) is used for testing such plugins.

Approaches for antipatterns detection so have several realizations.

Besides plugins for source code analysis, the approach realization for design patterns refactoring in UML-diagrams is accessible in the Internet.

The analysis of literature showed that the DPD issue is sufficiently popular. The procedures proposed by researchers have both advantages and disadvantages. The main disadvantage in the majority of the approaches proposed is dependence on the concrete programming language. It is necessary to perform work in the domain of the analysis of UML- diagrams and others language information sources, which are independent of the realization.

REFERENCES

1. Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
2. Arcelli Fontana, F. & Zanoni, M. (2011) A tool for design pattern detection and software architecture recognition. *Information Sciences*. 181. pp. 1306-1324. DOI : 10.1016/j.ins.2010.12.002
3. Bergenti, F. & Poggi, A. (2000) Improving UML Designs Using Automatic Design Pattern Detection. *Proc. 12th. International Conference on Software Engineering and Knowledge Engineering*. 2000. [Online] Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.3764
4. Beyer, D. & Lewerentz, C. (2003) CrocoPat: efficient pattern analysis in object-oriented programs. *Proceedings of the International Workshop on Program Comprehension (IWPC'03).* pp. 294-295.
5. Birkner, M. Object-oriented design pattern detection using static and dynamic analysis in java software. MB-PDE Java Software Design Pattern Detection Engine. [Online] Available from: https://mb-pde.googlecode.com/files/MasterThesis.pdf.
6. Blewitt, A., Bundy, A. & Stark, I. (2005) Automatic verification of design patterns in Java. *ASE '05 Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM New York. pp. 224-232.
7. De Lucia, A., Deufemia, V., Gravino, C. & Risi, M. (2009) Design pattern recovery through visual language parsing and source code analysis. *The Journal of Systems and Software*. 82. pp. 1177-1193. DOI: 10.1016/j.jss.2009.02.012
8. Dietrich, J. & Elgar, C. (2007) Towards a web of patterns. *Web Semantics: Science, Services and Agents on the World Wide Web*. 5(2). pp. 108-116. DOI: 10.1016/j.websem.2006.11.007
9. Fabry, J. & Mens, T. (2004) Language-independent detection of object-oriented design patterns. *Computer Languages, Systems & Structures*. 30. pp. 21-33. DOI: 10.1016/j.cl.2003.09.002
10. Gueheneuc, Y., Hamel, S. & Kaczor, O. (2006) Efficient identification of design patterns with bit-vector algorithm. *Proceedings of European Conference on Software. Maintenance and Reengineering (CSMR'06).* Bari. pp. 175-184.
11. Niere, J. et.al. (2002) Towards pattern design recovery. *Proceedings of International Conference on Software Engineering (ICSE'02).* Orlando. pp. 338-348.
12. Olsson, R. & Shi, N. (2006) Reverse engineering of design patterns from java source code. *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. Tokyo. pp. 123-134.
13. Pande, A., Gupta, M. & Tripathi, A.K. (2010) A decision tree approach for design patterns detection by subgraph isomorphism. *Communications in Computer and Information Science*. 101. pp. 561-564. DOI: 10.1007/978-3-642-15766-0_95
14. Rasool, G., Philipow, I. & Mader P. (2010) Design pattern recovery based on annotations. *Advances in Engineering Software.* 2010. 41. pp. 519-526. DOI: 10.1016/j.advengsoft.2009.10.014
15. Singh Rao, R. & Gupta, M. (2014) Design Pattern Detection by Multilayer Neural Genetic Algorithm. *International Journal of Computer Science and Network.* 3(1). pp. 9-14.
16. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G. & Halkidis, S.T. (2006) Design pattern detection using similarity scoring. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*. 32(11). pp. 896-909. DOI: 10.1109/TSE.2006.112
17. Wang, W. & Tzerpos, V. (2004) DPVK – an eclipse plug-in to detect design patterns in Eiffel systems. *Electronic Notes in Theoretical Computer Science*. 107. pp. 71-86. DOI: 10.1016/j.entcs.2004.02.049
18. Riel, A.J. (1996) *Object-Oriented Design Heuristics*. Addison-Wesley.
19. *Top Down Design in An Object Oriented World*. University of San Francisco. Department of computer science. [Online] Available from: http://www.cs.usfca.edu/~parrt/course/601/lectures/top.down.design.html.
20. Christopoulou, A., Giakoumakis, E.A., Zafeiris, V.E. & Soukara, V. (2012) Automated refactoring to the Strategy design pattern. *Information and Software Technology*. 54. pp. 1202-1214. DOI: 10.1016/j.infsof.2012.05.004
21. Dhambri, K., Sahraoui, H. & Poulin, P. (2008) Visual detection of design anomalies. *Proc. of the 12th European Conference on Software Maintenance and Reengineering*. pp. 279-283. DOI: 10.1109/CSMR.2008.4493326
22. Khomh, F., Vaucher, S., Gueheneuc, Y.-G. & Sahraoui, H. (2011) BDTEX: a cgm-based Bayesian approach for the detection of antipatterns. *The Journal of Systems and Software*. 84. pp. 559-572.
23. Marinescu, R. (2004) Detection strategies: metrics-based rules for detecting design flaws. *Proc. of the 20th IEEE International Conference on Software Maintenance.* pp. 350-359. DOI: 10.1109/ICSM.2004.1357820
24. Meyer, M. (2006) Pattern-based reengineering of software systems. *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*. Washington. pp. 305-306. DOI: 10.1109/WCRE.2006.42
25. Moha, N., Gueheneuc, G., Duchien, L. & Le Meur, A.F. (2009) Décor: a method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*. 36(1). pp. 20-36. DOI: 10.1109/TSE.2009.50
26. Travassos, G., Shull, F., Fredericks, M. & Basili, V.R. (1999) Detecting defects in object-oriented designs: using reading techniques to increase software quality. *Proc. of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages,*

and applications. University of Maryland. [Online] Available from: http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/oopsla99.pdf. DOI: 10.1145/320384.320389
27. Wieman, R. (2011) *Anti-pattern Scanner: an approach to detect anti-patterns and design violations*. Master's Thesis. The Software Evolution Research Lab. [Online] Available from: http://swerl.tudelft.nl/twiki/pub/Main/PastAndCurrentMScProjects / Thesis_RubenWieman2011.pdf.
28. *OMG Unified Modeling Language (OMG UML), Infrastructure. Documents Associated With Unified Modeling Language (UML), V.2.4.1*. Available from: http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/
29. Gil, J. & Maman, I. (2005) Micro Patterns in Java Code. *OOPSLA '05 Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM New York. pp. 97-116. DOI: 10.1145/1094811.1094819
30. Wikipedia – free encyclopedia. *Abstract semantic graph*. [Online] Available from: http://en.wikipedia.org/wiki/ Abstract_semantic_graph.
31. Basili, R. & Weiss, D.M. (1984) A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*. 10(6). pp. 728-738. DOI: 10.1109/TSE.1984.5010301
32. Wikipedia – free encyclopedia. *Web Ontology Language*. [Online] Available from: http://en.wikipedia.org/wiki/ Web_Ontology_Language.
33. Arcelli, F., Caracciolo, A. & Zanoni, M. (2012) A Benchmark for Design Pattern Detection Tools: a Community Driven Approach. *Special theme: Evolving Software*. 88. pp. 32.