РАЗРАБОТКА АНАЛИЗАТОРА АРХИТЕКТУРЫ Java-ПРИЛОЖЕНИЙ

Д. А. Тарасенко

Белорусский государственный университет, г. Минск;
Dmitriy.tarasienko@gmail.com;
науч. рук. – М. И. Давидовская, ст. преп.

Данная статья посвящена способам оценки архитектуры кода Java-приложений. Рассмотрены такие понятия как архитектура программного обеспечения (ПО), модульность, компоненты и сложность программной системы. Проанализированы основные метрики архитектуры кода и представлены алгоритмы их подсчета. В практической части рассмотрен алгоритм распределения классов по компонентам и реализовано приложение-анализатор. Данное приложение позволяет получить оценку по метрикам архитектуры кода программных систем и предложить вариант оптимизации программного проекта с разбиением на компоненты и уровни архитектуры.

***Ключевые слова*:** архитектура кода; анализатор; компонент; Java; модуль; метрики.

На сегодняшний день язык Java является одним из наиболее распространенных средств разработки ПО. Причём это часто крупные проекты, разработанные большими командами, рассчитанные на длительную эксплуатацию. Для таких проектов очень важно наличие качественной архитектуры. В противном случае поддержка большой кодовой базы становится весьма затратной, а реализация новых функций затруднительна.

Архитектуру ПО важно не только изначально правильно спроектировать, но и постоянно поддерживать в актуальном состоянии, а также контролировать соответствие ей реального кода. На текущий момент большинство существующих средств позволяют визуализировать связи между отдельными частями кода, не предоставляя при этом конкретных количественных метрик, позволяющих отслеживать изменения качества кода во времени. С другой стороны, существуют также средства статического анализа, умеющие выявлять конкретные фрагменты кода, потенциально влияющие на правильность и безопасность функционирования приложения. Наличие большого числа предупреждений статического анализатора, относящихся к определенной части кода, позволяет косвенно судить о качестве конкретной реализации, но плохо коррелирует с дефектами архитектуры.

Модульность — один из ключевых элементов архитектуры ПО. Для облегчения управления программной системой она может разбиваться

на модули — некоторые подпрограммы, которые могут вызываться из другого модуля системы. Высокая степень независимости модулей между собой достигается за счёт усиления внутренних взаимосвязей и ослабления связей между модулями.

Существуют оценки, базирующиеся на связности и сцеплении каждой отдельной пары модулей. Рассмотрим подробнее характеристики связности и сцепления каждого модуля, используя понятия силы связности (СС) и силы сцепления (СЦ) [2].

Для подсчета сложности программной системы в диапазоне от 0 до 1, изменим эти коэффициенты, разделив их на 10. Пусть зависимость первого порядка между парой модулей определяется по формуле:

$$d_{ij} = \begin{cases} 0{,}15(s_i + s_j) + 0{,}7c_{ij}, \text{если} c_{ij} \neq 0 \\ 0, \text{если} c_{ij} = 0 \\ 1, \text{если} i = j \end{cases} \quad (1)$$

Величина d_{ij} определяет вероятность того, что модуль j придётся изменить, если будет изменён модуль i. Величины s_i и s_j равны связности данных модулей, а c_{ij} — сцеплению между ними. Если сцепления нет, то $d_{ij} = 0$.

Компоненты представляют собой независимую систему, которая может быть использована повторно за счёт независимого развёртывания. Для выявления компонентов используется метод декомпозиции сложной системы, как и в случае с модульным подходом. Преимущество подхода заключается в том, что как только компоненты будут определены, каждый из них может быть назначен отдельной команде разработчиков, при этом проектировщики являются свободным в реализации внутренней организации компонент, каждый из которых является отдельной подсистемой [1].

Для эффективности обработки и упрощения представления программы на этапе использования обработчиков узлов синтаксического дерева, построенного библиотекой Spoon, построим собственное представление графа зависимостей с использованием списков связности. В результатах обработки для удобства создадим два представления, одно из которых включает в себя зависимости классов, другое — пакетов, так как неизвестно, какой уровень компонентов и само понятие компонента рассматривает пользователь. Для исследования данных представлений на наличие циклов используем стандартный механизм, основанный на поиске в глубину: из каждой вершины ориентированного графа

запускается поиск в глубину, если в ходе поиска найден путь в стартовую вершину, значит, граф содержит цикл.

Одной из основных задач при проектировании приложения либо системы является определение компонентов, то есть, классов, пакетов и внешних зависимостей, которые должны быть распределены по отдельным модулям. Для выделения компонент воспользуемся рядом принципов:

- Классы, которые имеют общие зависимости либо зависят друг от друга, логичнее располагать в одном компоненте.
- Абстракции могут быть вынесены в отдельный компонент для обеспечения выполнения принципа открытости/закрытости.
- Если изменения в наборе классов могут быть инициированы одним актором, данные классы с более высокой вероятностью принадлежат одному компоненту при наличии зависимостей между ними.

Приведённые выше принципы позволяют разработать алгоритм разбиения по компонентам, который состоит из следующих шагов:

7. Построить пары компонентов (классов либо пакетов), которые имеют зависимости друг от друга в виде наследования, типа полей, переменных либо вызове статического метода другого класса.
8. Удалить из построенных пар те, где внедрённая зависимость является абстракцией.
9. Удалить из зависимостей те, где параметр неустойчивости превышает некоторое пороговое значение.
10. Построить систему непересекающихся множеств, элементами которой является некоторое представление компонент, а принадлежность классов одному множеству определяется наличием зависимостей между ними.
11. Представить построенные множества пользователю как возможный вариант распределения компонентов по более массивным (например, модулям).

Шаг 2 обусловлен тем, что абстракции могут быть помещены в отдельный компонент (за исключением наследования). Шаг 3 же следует из того, что классы, которые инициализируются за счёт вызова фабричных методов или присваивания, генерируют зависимость вызывающего класса от конкретных реализаций, избежать которую невозможно [3].

Метод оценки вероятности того, что в модуле потребуются изменения при изменении некоторого другого модуля, также имеет большое значение в отношении классов, поскольку позволяет спроектировать механизм распределения классов на компоненты,

основанный на данной вероятности (так, имеет смысл располагать классы, изменения в которых с высокой долей вероятности зависят друг от друга, в одном компоненте). Главным преимуществом алгоритма, базирующегося на данной концепции, является возможность искусственно регулировать приблизительное число компонент, на который алгоритм осуществит разбиение. За счет разных показателей сцепления между компонентами он учитывает множество факторов, таких как связь исключительно с использованием статических переменных и методов, наличие управляющих переходов, использование примитивных либо сложных типов данных.

По формуле 1 для каждой пары компонентов требуется рассчитать значения связности для каждого класса и сцепления для каждой рассматриваемой пары. Так, связность уровня CC = 5 означает, что некоторый класс в рамках своих методов вызывает все публичные методы класса-зависимости. В свою очередь, CC = 7 означает, что вызываемый метод имеет те же параметры, что и некоторый другой метод, который уже вызывался у данного класса. Значение CC = 9 зависит от возвращаемого при вызове метода значения, CC = 10 означает, что класс имеет не более одного вызываемого метода. Таким образом, значения связность для каждого из компонентов могут быть получены на этапе построения графа без дополнительной обработки — при добавлении каждой зависимости по вызову метода необходимо пересчитывать значение связности и увеличивать его, если необходимо.

В случае сцепления значения СЦ = 1 и СЦ = 9 данные, касающиеся компонентов, такие как наличие статических методов и полей, заполняются на этапе построения модели по мере разбора классов. Значение СЦ = 3 означает, что оба класса имеют зависимость от некоторого внешнего, причем выражается она в доступе к статическому его полю либо методу. Значение СЦ = 5 означает, что классы имеют в своем коде выражение, содержащее статическую переменную в правой части, либо вызывают метод, который на этапе разбора обозначен в данных о компоненте как метод, в ходе выполнения которого может быть задано значение статической переменной. Аналогичным образом определяется значение СЦ = 10. Значение СЦ = 7 выявляется по наличию в теле метода некоторого ветвления, которое в условии использует параметр, переданный в метод. Это может означать как прямое обращение к логической переменной, так и цепочку вызовов методов, где в качестве объекта либо параметра вызова выступает параметр метода, от которого внешний компонент имеет зависимость.

Анализатор, разработанный в представленном исследовании, на данном этапе включает в себя стандартный функционал для

компонентов: подсчёт базовых характеристик в виде устойчивости, абстрактности, числа входящих и исходящих зависимостей, цикломатической сложности; обнаружение циклических зависимостей среди пакетов и классов, а также способен определить простейшее разбиение анализируемого проекта на компоненты и уровни архитектуры. Согласно обнаруженным недостаткам для успешной работы анализатора возможно расширить его функционал за счёт предоставления рекомендаций по улучшению архитектуры и внесения изменений в проект самим анализатором.

Библиографические ссылки

1. *Мартин Р.* Чистая архитектура. Искусство разработки программного обеспечения. — СПб.: Питер, 2018. — 352 с.
2. *Назаров С. В*. Архитектура и проектирование программных систем. – М.: ИНФРА-М, 2013. — Гл. 2, 4, 6. — С. 115-320.
3. *Тарасенко Д. А., Полойко Д. К.* Разработка анализатора архитектуры java-приложений/ Д. А. Тарасенко, Д. К. Полойко // НАУЧНЫЕ ИССЛЕДОВАНИЯ XXI ВЕКА. — 2022. — № 1 (15). — С. 76–80.