**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Кафедра технологий программирования**

ТАРАСЕНКО

Дмитрий Алексеевич

**РАЗРАБОТКА АНАЛИЗАТОРА АРХИТЕКТУРЫ JAVA-ПРИЛОЖЕНИЙ**

Дипломная работа

Научный руководитель:

Старший преподаватель

М.И. Давидовская

|  |
| --- |
| Допущена к защите  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2022 г  Зав. кафедрой технологий программирования  доктор технических наук, профессор,  Заслуженный деятель науки РБ А.Н. Курбацкий |

Минск, 2022

содержание

содержание 2

ВВЕДЕНИЕ 7

глава 1. ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ ПРИЛОЖЕНИЙ 10

1.1 Типы архитектуры ПО 10

1.1.1 Типы архитектуры ПО 12

1.2 Модульность 15

1.3 Представление архитектуры приложений 17

1.4 Парадигма ООП 18

1.5 Принципы SOLID 19

глава 2. численные метрики архитектуры программного обеспечения 23

2.1 Сложность программной системы и методы оценки сложностей 23

2.2 Механизмы внедрения зависимостей 27

2.3 Принципы организации компонентов 32

глава 3. проектирование анализатора архитектуры java-приложений 42

3.1 Обзор анализаторов архитектуры Java-приложений 42

3.2 Функциональные требования анализатора Java-приложений 43

3.3 Проектирование анализатора Java-приложений 46

глава 4. реализация анализатора архитектуры java-приложений 51

4.1 Библиотека Spoon 51

4.2 Алгоритм обработки графа 53

4.3 Модификация анализатора в случае интеграции технологий внедрения зависимостей 55

4.4 Анализ результата разработки на данном этапе 56

4.5 Методы изменения кода для улучшения его характеристик 58

4.5.1 Инверсия зависимостей 58

4.5.2 Использование остовных деревьев 61

4.5.3 Разбиение компонентов по уровням 63

4.5.4 Анализ вероятности изменения зависимого компонента 65

4.6 Реализация алгоритмов анализа строения компонентов 69

ЗАКЛЮЧЕНИЕ 75

список литературы 77

ПРИЛОЖЕНИЯ 78

перечень условных обозначений

|  |  |
| --- | --- |
| ООП | Объектно-ориентированное программирование |
| ПО  JSON | Программное обеспечение  JavaScript Object Notation |
| API | Программный интерфейс приложения (Application Programming Interface) |
| VCS | Системы контроля версия (Versions Control Systems) |
| IDE | Интегрированная среда разработки (Integrated development environment) |
| CRP | Принцип совместного повторного использования (Common Reuse Principle) |
| ISP | Принцип разделения интерфейсов (Interfaces Segregation Principle) |
| CCP | Принцип согласованного изменения (Common Closure Principle) |
| REP | Принцип эквивалентности повторного использования и выпусков (Reuse/Release Equivalence Principle) |
|  |  |
|  |  |
|  |  |

**Реферат**

Дипломная работа, 82 c., 18 рис., 7 таблиц.

**Ключевые слова:** АРХИТЕКТУРА, АНАЛИЗАТОР, КОМПОНЕНТ, JAVA, МОДУЛЬ, МЕТРИКИ.

**Объект исследования** — объектом исследования является архитектура кода на языке программирования Java и принципы её оптимального построения. В качестве предмета исследования рассматривается разработка и исследование характеристик анализатора архитектуры приложения на языке программирования Java.

**Цели работы —** рассмотреть концепции и методы анализа архитектуры кода, реализовать решения для их применения, а также разработать приложение для анализа архитектуры Java-приложений и вывода статистических показателей оценки архитектурного анализа.

**Методы исследования —** а) теоретические: изучение литературы, посвященной архитектуре приложений на языке Java, метрикам для анализа архитектуры; б) практические: сравнительный анализ существующих анализаторов архитектуры приложений, обобщение опыта работ в области анализа архитектуры Java-приложений, проведение экспериментов, проектирование спецификации и разработка приложения для анализа архитектуры Java-приложений.

**Результатами являются —** анализатор архитектуры Java-приложений.

**Область применения** — разработка систем на языке программирования Java с отслеживание качества архитектуры и возможностью оценки её состояния для дальнейших изменений.

**Рэферат**

Дыпломная работа, 82 с., 18 мал., 7 табліц

**Ключавыя словы**: АРХІТЭКТУРА, АНАЛІЗАТАР, КАМПАНЕНТ, JAVA, МОДУЛЬ, МЕТРЫКА.

**Аб'ект даследавання** **—** аб’ектам даследавання з’яўляюцца архітэктура кода на мове праграмавання Java і прынцыпы яе аптымальнай пабудовы. У якасці прадмета даследавання разглядаецца распрацоўка і даследаванне характарыстык аналізатара архітэктуры прыкладання на мове праграмавання Java..

**Мэты працы** **—** разгледзець метады аналізу архітэктуры кода, распрацаваць метады іх сумеснага прымянення, а таксама распрацаваць прыкладанне для вырашэння задачы, апісанай на этапе праектавання.

**Метады даследавання** **—** а) тэарэтычныя: вывучэнне літаратуры, прысвечанай метрыкам архітэктуры, даследаванне існуючых аналізатараў архітэктуры; б) практычныя: правядзенне эксперыментаў, праектаванне спецыфікацыі і распрацоўка прыкладання.

**Вынікамі з'яўляюцца** **—** аналізатар архітэктуры Java-прыкладанняў.

**Вобласць ужывання** **—** Распрацоўка сістэм на мове праграмавання Java з адсочваннем якасці архітэктуры і магчымасцю ацэнкі яе стану для далейшых зменаў.

**Abstract**

Graduate work, 82 p., 18 illustrations, 7 tables.

**Keywords**: ARCHITECTURE, ANALYZER, COMPONENT, JAVA, MODULE, METRICS.

**Object of research —** the object of research is the architecture of the code in the Java programming language and the principles of its optimal construction. The subject of the study is the development and study of the characteristics of the analyzer of the application architecture in the Java programming language.

**Purpose —** to consider methods of code architecture analysis, to develop methods of their joint application, as well as to develop an application to solve the problem described at the design stage.

**Methods of research** **—** a) theoretical: a study of the literature on architecture metrics, study of existing architecture analyzers; b) practical: conducting experiments, designing specifications, and developing applications.

**The results are** **—** Java-applications architecture analyzer.

**Scope** **—** development of systems in the Java programming language with monitoring the quality of the architecture and the ability to assess its condition for further changes.

ВВЕДЕНИЕ

Ежедневно программное обеспечение пополняется тысячами новых и повторных программных реализаций существующий решений. Данные решения разрабатываются как отдельными ИТ-специалистами, фрилансерами, студентами при обучении, так и небольшими командами разработчиков, крупными ИТ-компаниями. При этом цикл разработки программного продукта насчитывает ряд стадий, в которые входят проектирование, разработка, тестирование и сопровождение. Сопровождение является самой длительной стадией цикла разработки программного обеспечения, так как может длиться неограниченный промежуток времени.

На этапе сопровождения программного продукта основной задачей разработчика является внесение изменений в уже функционирующий продукт с целью удовлетворения новых требований. Таким образом, одним из критериев успешности продукта на этапе его сопровождения выделяют скорость внесения изменений в его функционал. Скорость внесения изменений зависит от того, как разработчик понимает структуру продукта, поддается ли она изменениям, возможна ли реализация запрошенных требований. Последний критерий требует изменения требований, в то время как первые два зависят от архитектуры программного обеспечения, которая задается на этапе проектирования. В случае, если стадии проектирования и разработки продукта были ориентированы исключительно на скорость выхода продукта на рынок, на этапе сопровождения программное обеспечение потребует больших людских и временных ресурсов.

Понимание вышеописанных принципов привело к разработке критериев архитектуры программного обеспечения. Данные критерии заключаются как в правилах, которым должны следовать разработчики при написании (например, широко применяющиеся при объектно-ориентированном программировании принципы SOLID), так и в конкретных количественных метриках, позволяющих оценить уже разработанную архитектуру [2, с. 75]. На основе этих критериев производится анализ архитектуры с помощью программных средств, которые в дальнейшем будем называть анализаторами архитектуры приложений.

Архитектура приложений включает в себя архитектуру модулей приложения, архитектуру баз данных и архитектуру кода, которая выражается в наличии определенным способом взаимодействующих компонент. В рамках объектно-ориентированного программирования в роли компонент как правило выступают классы либо наборы классов, которые сильно связаны между собой.

Существует несколько анализаторов архитектуры приложений, реализованных на языке программирования Java: JDepend, Sotograph, встроенные средства Intellij Idea. Каждый из данных анализаторов имеет отличительные черты, которые позволяют утверждать, что на данный момент сложно найти программное решение анализатора архитектуры приложений на языке Java, который удовлетворял бы всем возможным требованиям со стороны пользователя. Также, относительно небольшое число существующих решений демонстрирует, что метрики программного обеспечения нередко определяются собственным представлением сотрудников, осуществляющих контроль за качеством кода, либо коммерческими решениями компаний, к которым нет доступа у сторонних разработчиков.

Некоммерческие исследования в данной области позволят распространить возможности в анализе архитектуры на широкую аудиторию разработчиков, доработать концепции, позволяющие облегчить и удешевить долговременную поддержку и разработку приложений, разработать новые и усовершенствовать существующие алгоритмы, позволяющие на этапе разработки выявить проблемы архитектуры и предложить варианты их решений. В дальнейшем это позволит унифицировать ряд требований к архитектуре кода, а также получить знания, которые можно использовать в ряде активно развивающихся на данный момент отраслей: технологиях, отвечающих за машинное проектирование и написание кода, дальнейших алгоритмических исследованиях анализа кода, собственно, промышленной разработке приложений.

Использование анализаторов предоставляет возможность перейти от абстрактно-индивидуальной концепции качества архитектуры к математически заданным правилам, которые точно указывают на уязвимые места приложения не с точки зрения правильности его функционирования, как поступает большинство анализаторов кода, а с точки зрения простоты дальнейшего сопровождения приложения. В рамках данной работы рассмотрим создание анализатора архитектуры приложений, реализованных на языке программирования Java, который ориентирован на анализ структуры кода приложения за счет выявления компонент приложения, вычисления зависимостей между ними и определения метрик архитектуры для каждого из компонентов.

В главе 1 приведены типы архитектуры программного обеспечения, рассмотрены понятия модульности, свойства, которым должны удовлетворять модуля, а также сложность программной системы и методы её оценивания. Также будут рассмотрены принципы SOLID с целью задать их однозначное определение, которое будет использоваться при определении критериев для численных метрик. Описана концепция ООП, которая позволяет реализовывать архитектуру на основе принципов SOLID.

В главе 2 рассмотрены критерии и метрики качества архитектуры программного обеспечения. Подробно описаны преимущества и недостатки архитектур, определяющихся конкретными значениями метрик. Рассмотрены современные принципы организации зависимостей в приложениях.

В главе 3 рассмотрены существующие анализаторы архитектур, их возможности и недостатки. Описывается процесс разработки требования и проектирование на различных уровнях анализатора архитектуры, который реализован в качестве примера практического применения принципов, описанных в первых двух главах.

В главе 4 описывается результат разработки анализатора архитектуры, описана сторонняя библиотека, которая применяется при его разработке, рассматриваются и аргументируются алгоритмы, применяющиеся при его разработке, объясняются его характеристики, полученные в результате анализа его исходного кода, а также его недостатки и возможные методы улучшения.

глава 1. ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ ПРИЛОЖЕНИЙ

* 1. Типы архитектуры ПО

Согласно стандартам IEEE Std 1472000 и IEEE 1471, «архитектура — это базовая организация системы, воплощенная в её компонентах, их отношениях между собой и с окружением, а также принципы, определяющие проектирование и развитие системы» (Назаров С. В., Архитектура и проектирование программных систем, с. 41). IEEE 1471 также указывает на необходимость использования архитектуры системы при решении следующих задач:

1. анализ альтернативных проектов системы;
2. планирование перепроектирования системы;
3. общение по поводу системы между различными организациями, вовлеченными в её разработку;
4. выработка критериев оценки системы при принятии её на эксплуатацию;
5. разработка документации по использованию системы и её сопровождению;
6. проектирование и разработка отдельных элементов системы;
7. управление конфигурациями системы;
8. планирование бюджета и использования других ресурсов, которые требует система;
9. проведение анализа качества системы [3, с. 42].

В контексте данной работы рассматривается архитектура программного обеспечения. Программная архитектура представляет собой универсальную абстракцию системы, доступную всем заинтересованным в системе лицам. Она содержит сведения о том, какие решения были приняты на начальных этапах разработки системы, определяет ограничения реализации и организационную структуру, так как задает декомпозицию обязанностей. Формируется архитектура под влиянием разработчиков, опыта и привычек архитекторов, технической базы и окружения. Рассмотрим три основных этапа в процессе преобразования архитектуры от набросков до комплексного вида, наполненного полной информацией о системах:

1. «Архитектурный образец (шаблон) — описание типов элементов и отношений, ограничений на их использование» [3, с. 50]. Основываясь на данных ограничениях, подбирают набор соответствующих им вариантов архитектуры. Отличают два вида образцов: собственно, образцы проектирования, и каркасы. Отличие каркаса заключается в его расширяемости в некоторой конкретной области. Образцы описываются с использованием известных метрик качества, что позволяет архитектору выбрать подходящий по сформированным требованиям. Наряду с термином архитектурного шаблона иногда применяют архитектурный стиль [3, с. 50].
2. «Эталонная модель — это разделение системы по функциональным возможностям и потокам данных» [3, c. 51]. По сути, являются описанием декомпозиции решения некоторой проблемы на взаимодействующие компоненты, которые способны в совокупности её решить. Так как эталонные модели являются подтвержденными решениями, их применение ограничено уже сформировавшимися предметными областями.
3. Эталонная архитектура представляет собой отображение эталонной модели на программные компоненты, которые реализуют компоненты решения, описанные в эталонной модели. Если эталонная модель описывает декомпозицию функций и потоков данных, то этап эталонной архитектуры задает отображение сформировавшегося результата декомпозиции на компоненты системы, причем отображение не обязательно однозначное. Однозначным данное отображение является тогда, когда каждая отдельная функция либо поток данных реализованы в системе каждый в своем программном компоненте. В таком случае эталонная модель задает напрямую схему взаимодействия между программными компонентами системы [3, с. 51].

Рассмотрим подробнее набор свойств архитектуры:

* Архитектура определяет структуру, одну из основных характеристик системы. Структурный элемент системы может быть подсистемой, ресурсной библиотекой, базой данных, вычислительным узлом и т. д.
* Архитектура задает поведение системы. Поведение определяется характеристиками взаимодействий между структурными элементами системы.
* Концентрируется на значимых элементах. Элементы считаются значимыми, если они имеют продолжительное и устойчивое действие либо определяют значимые свойства (надежность, масштабируемость).
* Уравновешивает потребности заинтересованных лиц. Так как для системы определяется ряд функциональных и нефункциональных требования, некоторые из которых могут быть противоречивыми, задачей разработчика архитектуры является выработка компромиссного решения, основывающегося на основных факторах, определяющих трудоемкость разработки, её основные цели и цели проекта.
* Воплощает решение на основе логического обоснования. Данная информация используется разработчиком для избегания ненужных повторений своих действий.
* Может соответствовать определенному архитектурному стилю. Архитектурный стиль определяет номенклатуру компонентов и типов соединительных звеньев, а также набор условий, на которых они взаимодействуют друг с другом
* Определяет ограничения реализации. Реализация является представлением архитектуры только если соответствует проектным решениям. Архитектура задает обязательства каждого элемента относительно других элементов, а также накладывает определенные ограничения, связанные с распределением ресурсов.
* Способствует или сдерживает реализацию атрибутов качества системы. При этом сама по себе архитектура не гарантирует ни функциональности, ни качества.
* Облегчает анализ изменений и их организацию. Архитектура предполагает разделение всех изменений на архитектурные, локальные и нелокальные. Соответственно, внесение данных изменений требует корректирование взаимодействий между элементами системы, отдельного элемента либо нескольких элементов.
* Облегчает эволюционное макетирование. Становится возможным создание исполняемой системы на ранних этапах разработки, что позволяет выявить потенциальные проблемы.
* Позволяет более точно рассчитать стоимость и сроки создания системы.
* Оказывает влияние на окружение.
* Оказывает влияние на структуру и задачи компании-разработчика.
* Помогает в обучении.
* Представлена в каждой системе.

**1.1.1 Типы архитектуры ПО**

Существует ряд вариантов архитектуры программного обеспечения. Наиболее популярными из них в современной разработке являются многослойная, многоуровневая, сервис-ориентированная и микросервисная архитектуры.

Многослойная архитектура подразумевает разделение ПО на слои, лежащие друг на друге, при этом каждый слой выполняет определенную обязанность. Выделяют слой представления, содержащий пользовательский интерфейс и обеспечивающий положительную оценку пользователем графической составляющей системы, слой бизнес-логики, который отделяет графическое представление от вычислительных задач, что позволяет своевременно изменять внутреннюю реализацию вычислений, не влияя на другие слои, и слой передачи данных, который отвечает за взаимодействие с постоянными хранилищами.

К преимуществам многослойной архитектуры можно отнести простую реализацию, абстракцию за счет отделения слоев друг от друга, изолирование слоев от изменений в других слоях и управляемость программного обеспечения за счет слабой связанности. Основные её недостатки заключаются в слабой масштабируемости, монолитной структуре, которая затрудняет внесение модификаций, и обязательное посещение данными каждого слоя, даже если их представление не зависит от бизнес-логики.

Многоуровневая архитектура разделяет систему на уровни по принципу взаимодействия «клиент-сервер». Каждый уровень обладает следующими свойствами:

* На каждом уровне ничего не известно о существовании других уровней. Данное свойство позволяет существенно сократить число связей между элементами системы и, как следствие, упростить её.
* На каждом уровне ничего не известно о внутренней структуре других уровней. Связь между уровнями осуществляется через четко установленные сопряжения.
* Каждый уровень представляет собой группу модулей (раздельно компилируемых подпрограмм). При этом некоторые модули могут являться внутренними, недоступными с других уровней. Сопряжение между модулями задается их именами.
* Каждый уровень использует собственные ресурсы и либо скрывает их от других уровней, либо предоставляет некоторую их абстракцию, призванную защитить сопряжения от изменений внутри конкретного модуля.
* Каждый уровень может обеспечивать некоторую абстракцию данных в системе.
* Связи между уровнями ограничены явными аргументами, передаваемыми между уровнями. Использование некоторых глобальных данных недопустимо, при этом также нежелательно раздельное использование данных различными модулями в границах одного уровня.
* Каждый уровень должен иметь высокую прочность и слабое сцепление. Это означает, что каждая функция, выполняемая уровнем абстракции, должна быть представлена единственным входом, при этом аргументы, передаваемые между уровнями, должны быть отдельными элементами данных с как можно более простыми структурами.

К преимуществам многоуровневых систем относится возможность их как вертикального, так и горизонтального масштабирования в рамках одного уровня, высокая производительность. При этом системы, использующие данную архитектуру, требуют больших затрат по времени и ресурсам.

В основе сервис-ориентированной архитектуры лежат принципы многократного использования функциональных фрагментов информационных элементов, ликвидации дублирования функциональности в программной среде и унификации типовых операционных процессов. Компоненты программы могут быть распределены по разным сетевым узлам и предлагаются как независимые сервисы-приложения.

Сервис-ориентированная архитектурная модель состоит из пяти элементов:

* сервисы;
* сервисная шина;
* сервисный репозиторий;
* безопасность;
* управление.

Клиент отправляет запрос по сети, этот запрос обрабатывается сервисной шиной, которая отвечает за маршрутизацию. С помощью сервисного репозитория она направляет запрос в специально выделенный сервис, который может взаимодействовать с другими сервисами и базами данных для построения ответа. Управление и безопасность задают требуемые параметры транзакции.

Сервисы делятся на два вида: атомарные, которые предоставляют функциональность, не подлежащую декомпозиции, и композиционные, которые являются композицией нескольких атомарных сервисов, предоставляя более продвинутую функциональность. Также отличают типы сервисов: организационные, доменные, вспомогательные, интегрированные, сервисы приложений и безопасности, в зависимости от выполняемой ими задачи.

При использовании микросервисной архитектуры приложение представляет собой набор небольших сервисов, каждый из которых функционирует в собственном процессе. Данные сервисы могу развертываться независимо друг от друга с помощью полностью автоматизированного механизма, при этом централизованное управление ими минимально. Они могут быть реализованы на разных языках программирования и использовать разные средства хранения данных.

Архитектура состоит из следующих пяти компонентов:

* сервисы;
* сервисная шина;
* внешняя конфигурация;
* шлюз API;
* контейнеры.

Микросервисная архитектура обладает следующими свойствами:

* Компонентизация через сервисы (сервисы являются изолированными компонентами, каждый из которых несет единственную ответственность).
* Организация вокруг бизнес-возможностей.
* Ориентирована на конкретные продукты.
* Умные конечные точки и глупые каналы (Smart Endpoints and dump pipes; подразумевается, что бизнес-логики перемещается в конечные точки, что позволяет использовать простые протоколы передачи данных).
* Децентрализованное управление бизнес-логикой.
* Децентрализованное управление данными.
* Автоматизация инфраструктуры.
* Защита от сбоев.

К преимуществам микросервисной архитектуры можно отнести слабую связанность благодаря высокой степени изоляции, повышенную модульность, то, что сбои в отдельном сервисе не влияют на работу системы, высокую гибкость и масштабируемость, простоту модификаций, оптимальную систему обработки ошибок. При этом возникает повышенный риск сбоя при обмене данными между сервисами, усложняется управление, требуется решение проблем, свойственных распределенным архитектурам (например, балансировка нагрузки), комплексное тестирование и большой запас времени на реализацию.

* 1. Модульность

Для облегчения управления программной системой, она разбивается на модули — «замкнутые программы, которые можно вызвать из другого модуля системы» [3, с. 252]. Модуль состоит из интерфейса и реализации, при этом различные модули могут быть реализованы на различных языках программирования и откомпилированы отдельно. Высокая степень независимости модулей между собой достигается за счет усиления внутренних взаимосвязей и ослабления связей между модулями.

Модульное программирование имеет следующие преимущества:

* Упрощается разработка.
* Становится возможной параллельная разработка системы различными разработчиками.
* Упрощается настройка и модификация системы.
* Возникают контрольные точки, позволяющие контролировать стадии жизненного цикла системы.
* Обеспечивается полное тестирование.
* Программные системы не требуют единовременной тяжеловесной загрузки в оперативную память.

При вышеописанных преимуществах модульное программирование может требовать дополнительного времени исполнения программы или памяти, а также усложняет организацию взаимодействия между компонентами.

Модуль должен обладать следующими свойствами:

1. Модуль может быть активизирован средствами операционной системы или вызовом функционала из другого модуля.
2. На модуль можно ссылаться по имени.
3. Может обращаться к другим модулям.
4. Должен возвращать управление тому модулю, который передал управление ему.
5. Должен иметь один вход и один выход.
6. Не должен содержать большой объем кода и функционала.
7. Не должен сохранять историю своих вызовов, чтобы облегчить отладку.
8. Структура принятия решения должна определять модули, на которых прямо влияет решение, как подчиненные к модулю, который данное решение принимает.
9. Минимизация объема доступных данных для каждого модуля.
10. Нужно избегать внутренних процедур. Безопасность же лучше достигать за счет ограничения видимости и прав доступа к модулю. [3, с.253].

«Модульность — это свойство программного обеспечения, которое позволяет создать сколь угодно сложную программу (либо, согласно другому определению, свойство, которым обладает система, которая может подвергаться декомпозиции на ряд слабо зависящих друг от друга модулей)» [3, c.254]. Однако, большое количество модулей также налагает ограничения на систему, что описывает рисунок 1.1:

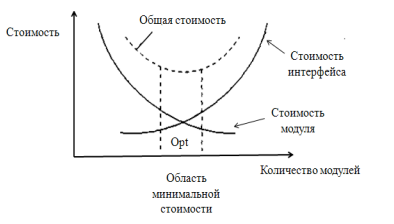


Рисунок 1.1 График зависимости стоимости системы от количества модулей

Таким образом, существует некоторое оптимальное количество модулей для системы. Точного способа рассчитать данное количество без эксперимента не существует, однако есть ряд свойств, которыми должен обладать оптимальный модуль, чтобы приблизить количество модулей системы к оптимальному: его должно быть проще использовать, чем разработать, его интерфейс при этом должен быть проще содержащейся в нем бизнес-логики, при этом модуль должен скрывать информацию от модулей, которые в ней не нуждаются (принцип информационной закрытости). Информационная закрытость позволяет разрабатывать модуля независимыми группами разработчиков и облегчить модификацию системы.

* 1. Представление архитектуры приложений

Существует ряд подходов к представлению архитектуры приложений:

1. Модульно-интерфейсный подход.
2. Объектно-ориентированный подход.
3. Компонентный подход.

Первым рассмотрим модульно-интерфейсный подход. Модульная программная система представляет собой древовидную структуру, узлы которой соответствуют программным модулям, а направленные дуги — статической подчиненности модулей. При определении набора модулей необходимо учитывать их соответствие ряду свойств: модуль вызывает модулем, расположенным выше в иерархии, и после окончания вычислений возвращает ему управление, принятие основных решений, связанных с алгоритмом вычислений, должно быть вынесено на самый высокий уровень и если в разных местах алгоритма встречается одна и та же функция, то она выносится в отдельный модуль.

Модульно-интерфейсный подход является методом структурного проектирования, который подразумевает, что каждый основной модуль системы может выполняться отдельно. Спецификации модулей и их интерфейсов дают структурную основу для проектирования каждого модуля и системы в целом. Спецификация модуля в таком случае состоит из функциональной спецификации, описывающей функциональность модуля, и синтаксической спецификации его входов, позволяющей определить, как правильно обраться к функциональности данного модуля. Существуют два основных метода разработки модульной структуры: нисходящей и восходящей разработок.

Что касается объектно-ориентированного подхода, Киммел П. отмечает две методики разработки систем согласно данному подходу: «потребление» и «производство». Первая методика заключается в активном использовании сторонних библиотек и классов, в то время как вторая подразумевает построение собственных решений. Основной задачей при проектировании является выявление классов предметной области — сущностей, описывающих задачу предметной области. Объект в данном случае — это сущность, которая используется при выполнении некоторой функции. Для решения вопросов взаимодействия с внешними библиотеками либо клиентскими приложениями вводятся понятия интерфейсных классов, предназначенных для связи элементов с внешними, классы-контролеры, управляющие другими классами, и классы-сущности, которые представляют собой данные. При этом классы-сущности включают в себя логические сущности, которые являются результатом выполнения смешанных запросов к базе данных.

При компонентном подходе компоненты представляют собой независимую систему, которая может быть использована повторно за счет независимого развертывания. 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. Парадигма ООП

Основополагающими понятиями парадигмы объектно-ориентированного программирования выделяют инкапсуляцию, наследование и полиморфизм. При этом стоит учитывать, что с появлением объектно-ориентированных языков программирования парадигма ООП претерпела ряд изменений, которые ослабили требования к соблюдению принципов, ей свойственных. Рассмотрим подробнее каждое из понятий.

Инкапсуляция — это механизм, позволяющий ограничить доступность данных и методов класса из сторонних классов. Объектно-ориентированные языки программирования поддерживают эффективный способ инкапсуляции, что позволяет ограничить компонент, состоящий из связанных методов и данных. Реализуется это с помощью приватных членов данных и общедоступных членов-функций определенных классов. При этом инкапсуляция применяется и не в объектно-ориентированных языках (например, языке программирования С) за счет использования структур. За счет введения в объектно-ориентированные языки ключевых слов, задающих область видимости полей и методов, удалось восстановить ограничения, которые пропали из-за того, что пользователям классов открывается доступ к их внутренней структуре.

В языке программирования Java отсутствует возможность разделить объявление и определение класса, так как отсутствует деление на заголовки и реализации, что сильно ослабляет инкапсуляцию. Таким образом, реализация парадигмы ООП перестает зависеть от инкапсуляции, при этом ряд объектно-ориентированных языков практически не имеют принудительной инкапсуляции.

Наследование в объектно-ориентированных языках выражается в повторном объявлении группы методов либо переменных в ограниченной области видимости. Преимущество объектно-ориентированных языков перед реализациями, заменяющими наследование в иных языках, в неявном приведении объектов к родительскому типу.

Полиморфизм подразумевает, что объекты могут представлять различное поведение в зависимости от контекста. Для реализации полиморфизма используются ссылки на методы (либо указатели на функции в языках С и С++). Проблема явного использования ссылок заключается в соглашениях, связанных с их использованием. Ссылку необходимо проинициализировать и вызывать метод только посредством её использования. Реализация полиморфизма на уровне языка решает эту проблему.

Основное преимущество, которое дает полиморфизм, это возможность использования инверсии зависимостей. До появления встроенной реализации полиморфизма приложение с точки зрения архитектуры представляло собой дерево вызовов функций, в котором зависимости исходного кода следовали за потоком управления. Таким образом, однозначно существует функция, которая зависит от всех остальных, причем изменение реализации на самом низком уровне влечет изменения реализаций на всех вышестоящих. При инверсии зависимостей уже реализация низкоуровневой функции зависит от поведения, прописанного ей на более высоком уровне. Таким образом, факт поддержки полиморфизма в объектно-ориентированных языках означает, что любую зависимость в исходном коде можно инвертировать, что в свою очередь предоставляет разработчику абсолютный контроль над направлением зависимостей.

Описанные выше возможности позволяют инвертировать зависимости таким образом, что программный интерфейс будет зависеть от бизнес-правил, а не наоборот. Соответственно, бизнес-правила можно скомпилировать в независимую единицу развертывания. Таким образом, если в программный интерфейс будут внесены изменения, потребуется лишь повторно развернуть компонент. Данный принцип называется независимостью развертывания. При этом, если система состоит из нескольких модулей, модули могут разрабатываться независимо друг от друга. Данный принцип называется независимостью разработки.

* 1. Принципы SOLID

Принципы SOLID задают правила компоновки данных и методов на уровне классов, однако применимы не только к объектно-ориентированному коду. Цель принципов — создать программные компоненты, которые будут удовлетворять следующим свойствам:

* Позволяют изменения.
* Выглядят достаточно просто и понятно, чтобы не затруднять внесение изменений разработчиком.
* Образуют основу для компонентов более высокого уровня, которые могут использоваться во многих программных компонентах.

SRP (Single Responsibility Principle) — Принцип единственной ответственности. Представляет собой следствие закона Конвея: структура программной системы является лучшей, если формируется в основном под влиянием социальной структуры организации, использующей эту систему, поэтому каждый программный модуль имеет одну и только одну причину для изменения. Причиной для изменения как правило является некоторый актор — лицо, которое настаивает на изменениях.

Признаки нарушения принципа SRP:

* Непреднамеренное дублирование. Подразумевается, что один и тот же класс содержит методы, которые содержат в себе определенную логику, которую задают разные акторы. Опасность заключается в том, что данные методы могут совместно использовать некоторые приватные методы, в которых могут потребоваться изменения по требованию одной из сторон.
* Слияния. Обычно встречается в классах с большим количеством методов. Данная проблема подобна той, когда разные разработчики вносят изменения в один и тот же класс. Предположим, разные акторы потребовали одновременно внести изменения в класс. После выполнения требований изменения оказались несовместными, из-за чего понадобилось слияние.

Решение данных проблем сопряжено с перемещением функций в отдельные классы. Недостаток данного похода заключается в том, что требуется создавать несколько экземпляров различных классов и следить за ними. Как правило, такая проблема решается применением паттерна фасад, инкапсулирующего реализацию каждого из методов в отдельным классе за общим интерфейсом. Таким образом, можно создать объект, который предоставляет все тот же функционал что и до внесения изменений, но при этом в случае изменения требований к реализации потребуются изменения в не зависящих друг от друга классах. В случае, если разработчик предпочитает реализовывать бизнес-правила на уровне, наиболее близком к данным, интерфейс доступа к данным реализуется на уровне фасада.

OCP (Open-Closed Principle) — принцип открытости/закрытости. Его суть состоит в том, что простая для изменения система должна предусматривать простую возможность изменения её поведения добавлением нового кода вместо изменения существующего. Реализация данного принципа достигается комбинацией реализации принципа SRP с инверсией зависимостей. Важно добиться того, чтобы все отношения компонентов были однонаправленны. Если компонент А необходимо защитить от изменений, которые могут потребоваться в компоненте Б, то нужно сделать так, чтобы компонент Б зависел от компонента А.

Введем понятия интерактора — класса, который реализует бизнес-логику, контроллера — класса, который отделяет логику интерактора от логики презентатора — класса, который задает логику предоставления пользователю представлений — моделей данных, предназначенных для предоставления пользователю. В таком случае контроллер должен быть защищен от изменений в презентаторах, презентаторы от изменений в представлениях, а интерактор в принципе не должен зависеть ни от одного другого компонента. Таким образом, можно ввести понятие уровня — так, интерактор занимает самый высокий уровень, после чего следуют контроллеры, презентаторы и представления. Чем выше уровень, тем лучше защищены компоненты. При этом реализация уровней достигается за счет применения интерфейсов, которые избавляют классы от транзитивных зависимостей. Транзитивные зависимости сами по себе нарушают принцип CRP, который будет рассмотрен ниже, поэтому, даже при том, что защита интерактора от изменений в контроллере имеет наивысшую важность при проектировании архитектуры приложения, также необходимо защитить контроллер от изменений, которые могут быть внесены в интерактор, скрыв от контроллера реализацию интерактора.

LSP (Liskov Substitution Principle) — принцип подстановки Барбары Лисков. Принцип утверждает, что для создания программных систем из взаимозаменяемых компонентов эти компоненты должны соответствовать контракту, который позволяет заменять эти компоненты друг другом. В условиях наследования это означает, что в случае, если один из классов объявлен подклассом другого, то данные типы должны быть заменяемыми. Предположим, класс-наследник обрабатывает данные некоторым отличным от отработки данных в наследуемом классе образом, например, использует только вместе некоторые поля, которые в классе-родителе могут быть использованы отдельно. Тогда при подстановке подкласса пользователь будет ожидать иного поведения от экземпляра. Другим примером нарушения LSP является несоответствие протоколов связи модулей приложения посредством REST-технологий. В случае, если некоторые реализации модулей изменят протокол доступа, потребуются изменения во всех зависимых модулях в виде дополнительного обработчика.

ISP (Interface Segregation Principle) — принцип разделения интерфейсов. Данный принцип призывает избавляться от неиспользуемых зависимостей. Предположим, имеется класс, который имеет несколько публичных методов. В таком случае любой класс, имеющий от него зависимость, зависит от всех публичных методов данного класса, так как изменения любого из этих методов влекут изменения класса. Решением в данном случае является распределение данных методов по интерфейсам и внедрение зависимости от интерфейса вместо реализации. Данная проблема характера только для языков со статической типизацией, так как в языках с динамической типизацией подобные объявления определяются во время выполнения, что лишает необходимости перекомпиляции кода.

DIP (Dependency Inversion Principle) — принцип инверсии зависимостей. Код, реализующий бизнес-правила высокого уровня, не должен зависеть от низкоуровневых реализаций. Напротив, детали должны зависеть от более высокого уровня. В языках со статической типизацией это означает, что инструкции import должны ссылаться только на модули с исходным кодом, содержащим интерфейсы, абстрактные классы и другие абстракции. Очевидно, повсеместное использование данного принципа невозможно, так как стандартная библиотека Java содержит множество конкретных классов. С другой стороны, эти классы довольно стабильны — вероятность их изменения довольно мала.

Стоит отметить, что каждое изменение абстракции влечет за собой изменение всех её реализаций. Таким образом, интерфейс должен быть как можно более стабилен, в то время как его реализации более изменчивы. Таким образом, стабильной называется архитектура, в которой вместо зависимостей от переменчивых конкретных реализаций используются зависимости от стабильных абстракций. Соответственно, можно выделить следующий правила построения стабильной архитектуры:

* Не добавлять зависимостей от изменчивых конкретных классов. Данное правило универсально с точки зрения системы типизации языка.
* Не наследовать изменчивые конкретные классы. Данное правило является прямым следствие из предыдущего, так как наследования является самой жесткой формой зависимости.
* Не переопределять конкретные функции. При переопределении конкретных функций, которые используют внешние зависимости, данные зависимости наследуются.
* Не использовать зависимостей от имен конкретных изменчивых сущностей.

Таким образом, модульность систем облегчила как процесс их разработки, так и проектирования. Разбиение программы на модули позволяет оценить каждый модуль по ряду характеристик, после чего, наладив взаимодействие между ними, получить систему, обладающую требуемыми характеристиками.

**ВЫВОДЫ**

1. Рассмотрены типы архитектуры ПО, дано определение модульности.
2. Рассмотрены преимущества модульности относительно других подходов, описаны свойства и представления модулей.
3. Кратко описаны нововведения парадигмы ООП, ее преимущества.
4. Приведены описания принципов SOLID в контексте практического значения для анализа архитектуры кода.

глава 2. численные метрики архитектуры программного обеспечения

* 1. Сложность программной системы и методы оценки сложностей

Сложность системы имеет широкое влияние на её реализацию, так как от неё зависит продуктивность разработчиков и надежность системы — чем выше сложность, тем сложнее добиться надежности. При оценке сложности системы выделяют три группы метрик: метрики размера программ, метрики сложности потока управления программ, метрики сложности потока данных программ. В качестве метрики размера программ, как правило, принимается число строк исходного текста, причем под строкой понимают любой оператор программы.

В простейшем случае сложность системы можно определить как сумму сложностей её модулей, при этом сложность модуля может определяться различными способами. М. Холстед ввел две метрики сложности модуля: длину N и объем V, где

|  |  |  |
| --- | --- | --- |
|  |  | (2.1) |

где n1 — число различных операторов модуля, n2 — число различных операндов, а

|  |  |  |
| --- | --- | --- |
|  |  | (2.2) |

т. е., объем модуля равен количеству символов для записи всех операндов и операторов текста программы.

Метрики размера программ игнорируют внутренние связи между элементами в системе, поэтому были введены метрики сложности потока управления. Как правило, эти метрики оперируют либо плотностью управляющих переходов внутри программ, либо взаимосвязями этих переходов. Т. Джилб ввел метрику, которая определяет логическую сложность программы как её насыщенность операторами ветвления, при этом вводятся две характеристики:

1. CL — абсолютная сложность программы, равная количеству операторов условия.
2. cl — относительная сложность программы, равная отношению CL к общему числу операторов.

Позже метрику дополнили характеристикой CLI, равной максимальной вложенности операторов в программе.

Т. МакКейб выработал метрику ацикломатической сложности, равной

|  |  |  |
| --- | --- | --- |
|  |  | (2.3) |

где E — количество дуг, а N — вершин в управляющем графе программной системы. Дальнейшее уточнение сложности потребовало, чтобы модуль мог оцениваться как локальная структура с учетом исключительно внутренних элементов и взаимосвязей между ними.

Разобьем вершины управляющего графа на две группы: принимающие вершины, у которых положительная степени вершины не превосходит 1, и вершины отбора, у которых положительная степень больше либо равна 2. Чтобы получить оценку по методу граничных условий, необходимо разбить управляющий граф на максимальное число подграфов, которые обладают следующими свойствами:

1. Вход в граф осуществляется только через некоторую вершину отбора.
2. Каждый подграф включает некоторую вершину, в которую можно попасть из любой другой вершины подграфа (его нижнюю границу).

Число вершин, образующих такой подграф, называют скорректированной сложностью вершины отбора. Все скорректированные сложности вершину суммируются, образуя абсолютную граничную сложность программы. После этого относительная граничная сложность программы определяется как

|  |  |  |
| --- | --- | --- |
|  |  | (2.4) |

где v — общее число вершин графа программы, Sa — абсолютная граничная сложность программы.

Результатом предварительного проектирования является иерархическая структура модулей, пример которой приведен на рисунке 1.2.

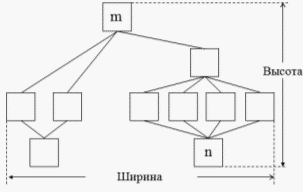


Рисунок 1.2 Пример иерархической структуры модулей

Данная структура обладает характеристиками высоты и ширины, которые равны количеству уровней управления и максимальному числу модулей на одном уровне соответственно. Локальными характеристиками модуля в подобной структуре являются коэффициент объединения по входу (Fan\_in), равный количеству модулей, которые прямо управляют данным модулем, и коэффициент разветвления по выходу (Fan\_out), который равен числу модулей, которые прямо управляются данным. Качество структуры можно определить как степень её близости к древовидной, так как древовидная структура обладает наилучшими характеристиками. Невязка, задающая данную величину, равна

|  |  |  |
| --- | --- | --- |
|  |  | (2.5) |

где et = n-1 — число ребер дерева, построенного на данном количестве вершин,

|  |  |  |
| --- | --- | --- |
|  |  | (2.6) |

число ребер полного графа, построенного на данном количестве вершин, e — число ребер оцениваемой структуры. Данная невязка задает грубую оценку структуры, более точная оценка вводится за счет использования характеристик сцепления и связности: хорошая структура должна иметь низкое сцепление и высокую связность, при этом большое значение Fan\_in свидетельствует о высоком сцеплении, а большое значение Fan\_out — о сложности вызывающего модуля. Однако эти характеристики не учитывают веса связи, поэтому были введены характеристики ifan\_in(i) и ifan\_out(j), которые задают количество элементов и структур данных, из которых i-ый модуль получает информацию и которые обновляются j- ым модулем соответственно. Данные информационные коэффициенты суммируются со структурными sfan\_in(i) и sfan\_out(j), которые учитывают только вызовы модулей. Итоговая формула выглядит как

|  |  |  |
| --- | --- | --- |
|  |  | (2.7) |

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | (2.7.1) |
|  | |  |  |
|  |  | | (2.7.2) |
|  |  | |  |

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):

Связность — мера зависимости частей одного модуля друг от друга.

1. Связность по совпадению (СС = 0). В модуле отсутствуют ярко выраженные внутренние связи.
2. Логическая связность (СС = 1). Части модуля обладают функциональным подобием (например, по-разному реализуют функции).
3. Временная связность (СС = 3). Части модуля не связаны, но необходимы в один и тот же период работы системы.
4. Процедурная связность (СС = 5). Части модуля связаны порядком выполняемых ими действий, реализующих некоторый сценарий поведения.
5. Коммуникативная связность (СС = 7). Части модуля работают с одними и теми же структурами данных.
6. Информационная (последовательная) связность (СС = 9). Выходные данные одной части используются как входные во второй части модуля.
7. Функциональная связность (СС = 10). Части модуля реализуют вместе одну функцию.

Сцепление — мера взаимозависимостей модулей по данным.

1. Полностью независимые модули (СЦ = 0). Модули, не вызывающие друг друга, и не использующие общих данных.
2. Сцепление по данным (СЦ = 1). Один модуль вызывает второй, входные и выходные параметры — простые элементы данных.
3. Сцепление по образцу (СЦ = 3). Модули ссылаются на одну и ту же глобальную структуру данных.
4. Сцепление по общей области (СЦ = 5). Модули разделяют одну и ту же общую глобальную структуру данных.
5. Сцепление по управлению (СЦ = 7). Один модуль явно управляет поведением другого за счет передачи ему флагов либо команд управления.
6. Сцепление по внешним ссылкам (СЦ = 9). Один модуль обладает доступом к данным другого модуля через внешнюю точку входа.
7. Сцепление по кодам (содержанию) (СЦ = 10). Один модуль прямо ссылается на содержимое другого модуля.

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:

|  |  |  |
| --- | --- | --- |
|  |  | (2.8) |

Величина dij определяет вероятность того, что модуль j придется изменить, если будет изменен модуль i. Величины si и sj ­равны связности данных модулей, а cij — сцеплению между ними. Если сцепления нет, то dij = 0. Матрица D может быть представлена неориентированным графом, однако она не отображает полную модель программы, так как в ней не отражены явно влияния зависимостей более высоких порядков. Для полного изображения зависимостей между любой парой модулей вычисляется полная матрица зависимостей:

1. Находятся все пути в графе за исключением циклов между парой модулей.
2. Вероятность для найденных путей вычисляется как произведения вероятностей соответствующих дуг.
3. Вычислить зависимости между модулями, используя вероятности для путей, но при этом пути не являются взаимно исключающими.

Результатом вычисления является вероятность того, что при изменении некоторого модуля понадобится изменить другой. Число модулей, которые потребуется изменить при изменении данного, равно сумме элементов строки, соответствующей данному модулю, за исключением диагонального. Также можно вычислить вероятность, что при изменении данного элемента не придется изменять остальные как

|  |  |  |
| --- | --- | --- |
|  |  | (2.9) |

Оценка сложности данной системы равна сумме всех элементов матрицы, разделенной на общее число модулей.

* 1. Механизмы внедрения зависимостей

Зависимость класса от другого класса (интерфейса, структуры и т. д.) подразумевает использование реализации класса-зависимости (его объектов, методов, полей) внутри зависимого класса. При этом существует несколько способов внедрить данную зависимость:

1. Создавать зависимость в самом зависимом классе.
2. Внедрение зависимости путем передачи экземпляра пользовательского класса в конструктор.
3. Делегировать внедрение зависимостей внешней технологии.

При использовании первого способа, т. е. создания зависимости в самом зависимом классе, подразумевается прямое использование конструктора, вызов соответствующего метода фабрики объектов либо иной другой способ создания объекта конкретного класса. В качестве преимуществ данного подхода можно выделить простоту написания кода (объект создается ровно в том месте, которое нужно разработчику), а также полный контроль класса над создаваемыми объектами. Класс не имеет зависимости от компонентов, конструирующих объект, потому что сам им является.

Недостатками данного подхода являются тесная связь двух классов, которая в случае низкой устойчивости зависимости приводит к необходимости постоянно изменять зависимый класс, при этом любая зависимость от данного класса включает в себя исходную зависимость, то есть, любая зависимость от зависимого в данной связке класса является транзитивной. Также при жесткой зависимости класс не может быть заменен на иную реализацию, а тестирование данной связи по отдельности практически невозможно — в случае, если необходимо протестировать класс, имеющий жесткие зависимости, придется создать экземпляры всех классов-зависимостей, при этом даже при обнаружении ошибок на этапе тестирования выяснить класс, имеющий ошибку, намного сложнее. Также данная связь самим своим существованием с высокой долей вероятности нарушает принцип единой ответственности — у разных классов могу быть разные поводы для изменения. Кроме того, у зависимого класса падает показатель связности, так как он включает в себя логику, явно не связанную с его функционалом — как минимум, создание экземпляров класса-зависимости.

Во втором способе, т. е. внедрении зависимости путем передачи экземпляра пользовательского класса в конструктор, данная зависимость является слабой, так как класс-зависимость может быть заменен без изменения кода зависимого класса (например, передачей объекта, являющегося экземпляром его расширения). Также данный способ внедрения делает возможным модульное тестирование, так как вместо экземпляра класса-зависимости при тестировании может быть передан экземпляр с иным поведением либо и вовсе экземпляр класса-заглушки.

Однако, данный подход имеет свои недостатки: в первую очередь, данный подход лишает код преимуществ полиморфизма, а именно возможности не представлять программу в виде последовательного выполнения ряда функций, на которые представлены указатели, из-за чего приходится создавать некоторый класс, который обладает минимальной устойчивостью и занимается как раз созданием и внедрением объектов в основаниях данных цепочек внедрения. Также код по-прежнему нарушает принцип внедрения единой ответственности, так все равно находится класс, отвечающий в том числе за создание объектов.

В третьем способе, который позволяет делегировать внедрение зависимостей внешней технологии, код приложения работает с внедряемыми классами без посредников, не существует компонентов, отвечающих за создание и внедрение экземпляров. Преимущества — наиболее простой код, при этом классы легко заменимы в реализациях, меняется только код провайдера, причем провайдер ничего не значит о конкретных реализациях, использующих данные зависимости. Также классы тестируемы, поскольку вместо конкретных экземпляров достаточно заменить зависимости от них, а это возможно сделать без изменения тестируемого класса.

Конкретные реализации компонентов могут сконцентрироваться исключительно на реализации собственной логики. Однако, есть и недостатки: в первую очередь, высокий порог вхождения: написание кода требует изучения соответствующих дополнительных технологий. Кроме того, добавляется зависимость от самой технологии, причем добавляется во все использующие её компоненты. В случае, если технология не является легковесной либо включает в себя дополнительные зависимости, проект как в рамках разработки, так и после развертывания увеличивается в размерах, увеличивается время компиляции и требуемые характеристики системы для использования итогового приложения.

Все основные технологии в языке Java поддерживают два типа внедрения зависимостей: путем внедрения её напрямую в поле данного типа либо путем внедрения в конструктор. При этом данные типы внедрения не имеют ничего общего с рассмотренными выше способами, так как не подразумевают создания объекта средствами зависимого класса.

Средства фреймворка Spring предоставляют способ внедрения зависимостей. Определим понятие бина как некоторого экземпляра класса. Для конфигурации бинов могут использоваться xml-файлы, которые содержат в себе информацию о классе, которому принадлежит экземпляр, необходимом списке параметров, включающем, если это требуется, ссылки на другие экземпляры, а также уникальный идентификатор, использующийся для их разграничения. После создания бины помещаются в так называемый контейнер, представляющий собой HashMap, в котором в качестве ключа хранится уникальный идентификатор из объявления бина, а в качестве значения — созданный экземпляр.

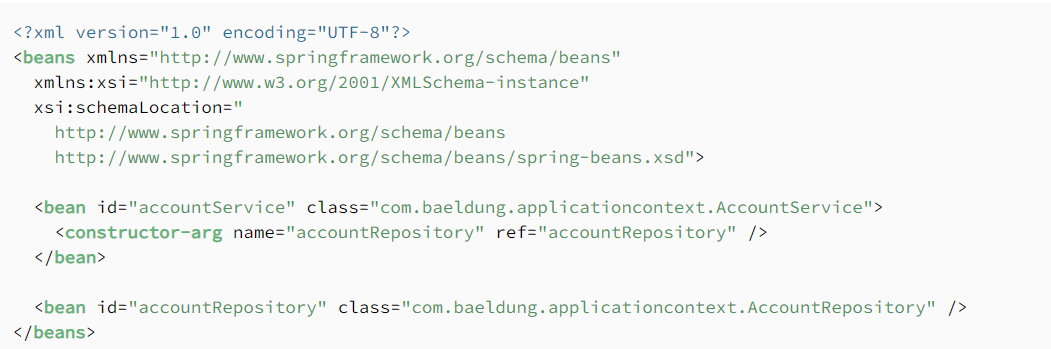


Рисунок 2.1 Пример xml-конфигурации контекста Spring

Для использования бинов необходимо создать экземпляр интерфейса ApplicationContext, который на основе конфигурации предоставляет доступ к контейнеру, после чего достать из данного контейнера необходимый бин, передав его уникальный идентификатор. Начиная с версии Spring 4.0 в качестве идентификатора может выступать типизация класса. Существуют следующие виды контекстов:

* AnnotationConfigApplicationContext — предоставляет возможность загрузить бины, сконфигурированные с помощью использования аннотаций.
* ClassPathXmlApplicationContext — загружает бины, сконфигурированные с помощью xml, размещенного в пределах ресурсных директорий проекта.
* FileSystemApplicationContext — аналогичен предыдущему за исключением того, что в качестве директории конфигурационных файлов может быть указан абсолютный путь в системе.
* AnnotationConfigWebApplicationContext — также предоставляет возможности конфигураций на основе аннотаций за исключением того, что обеспечивает поддержку некоторых специфических для веб-приложений аннотаций и стандартных ресурсных директорий.
* XmlWebApplicationContext — аналогичен предыдущему, использует xml-файлы для конфигурации.

С момента выхода Spring Boot процесс конфигурации был упрощен за счет выполнения многих стандартных операций на этапе старта приложения, а также добавления в стандартную сборку многих зависимостей. Были устранены некоторые недостатки указанного выше способа конфигурации: отсутствие необходимости в xml-файлах, добавлена возможность автоматического подбора бинов с помощью использования аннотации Autowired. Это позволило устранить дублирование идентификаторов в коде приложения и в файлах конфигурации, а также ослабить видимую зависимость приложения от контекста, поскольку в большинстве случаев разработчик не нуждается в управлении контекстом напрямую. Однако, стоит отметить ряд недостатков данного решения: тяжеловесность (большое количество зависимостей, увеличивающих вес и длительность старта и сборки приложения), зависимость от фреймворка, которая в данном случае представляет собой зависимость от множества модулей, а также неявность выполняемых действий.

Bootique представляет собой более легковесный фреймворк, чем Spring. Внедрение зависимостей происходит с помощью Bootique DI, который вместо конфигурационных xml-файлов предоставляет возможность напрямую задать экземпляр класса (класс), который является реализацией некоторого интерфейса либо его самого в так называемых модулях (имплементация интерфейса BQModule), по которым и происходит разграничение области видимости бинов. Для ручного доступа к аналогу контекста используется объект класс Injector, который позволяет аналогичным образом достать экземпляр класса, внедренный для определенного типа.

Для задания внедрений используется реализация метода configure интерфейса BQModule. Для задания сложны объектов используется аннотация Provides вкупе с фабричными методами, расположенными в той же реализации BQModule. С помощью вызова метода bind класса Binder возможно указать как конкретную реализацию интерфейса (класс), так и задать конкретный созданный с помощью фабричных методов экземпляр. Также для конкретного класса можно задать квалификатор — уникальное значение, позволяющее разграничить экземпляры одного и того класса между собой. При этом в качестве квалификатора во избежание дублирования строк может использоваться аннотация, создаваемая разработчиком. Также существует поддержка типизированных классов, причем под разные типизации возможно задать разные реализующие объекты.

Также стоит отметить возможность использования интерфейса Provider для возвращения бинов из фабричных методов. С помощью использования подобного способа внедрения можно избегать циклических зависимостей, от которого в ядре фреймворка существует проверка по умолчанию, которая не даст коду выполниться: при использовании Provider для разрыва циклической зависимости на этапе вызова метода get будет уже сконфигурирован объект, который содержит данный Provider в качестве поля либо внедренного параметра. Внедрять параметры можно аналогично Spring в сеттеры либо конструкторы, используя аннотацию Inject.

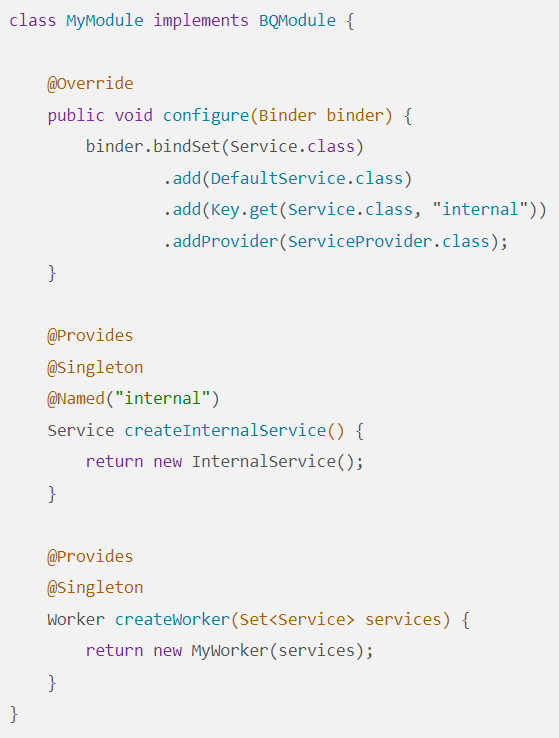


Рисунок 2.2 Пример конфигурации контекста Bootique

Таким образом, путем использования Bootique возможно упростить взаимодействие с зависимостями, при этом он более легковесный, не требует дублирования строк и позволяет разработчику напрямую управлять содержимым контекста. Однако, остается ряд общих архитектурных недостатков: существует некоторый класс (реализация BQModule), которая зависит от всех конкретных реализаций; зависимость от фреймворка; возможность напрямую задать экземпляр конкретного класса вместо его абстракции. При этом имеет место нестандартное архитектурное решение: вместо использования в качестве кода более высокого уровня абстракций, которые расширяют являющиеся заменимыми конкретные реализации, абстракция в данном случае достигается за счет подстановки в приложение различных модулей, представляющих собой реализации BQModule и их классы-зависимости. На низком уровне это выглядит аналогично стандартным решениям: класс может зависеть от некоторой абстракции, которая может быть представлена конкретным классом и извлечена из контекста.

* 1. Принципы организации компонентов

Один из основных вопросов при организации компонентов — к какому компоненту отнести тот или иной класс. Сложность данного вопроса заключается в том, что ответ на него практически полностью зависит от контекста. Рассмотрим ряд принципов, которые определяют связность компонентов:

* REP (Reuse/Release Equivalence Principle) — принцип эквивалентности повторного использования и выпусков. Суть данного принципа заключается в том, что при выпуске новой версии продукта разработчики обязаны рассылать описание новой версии, чтобы пользователи смогли определить, требуется ли им переходить на неё и какие изменения это повлечет. С точки зрения архитектуры программного продукта это означает, что классы и модули, составляющие компонент, должны принадлежать связной группе. При этом данные классы и модули должны выпускаться вместе.
* CCP (Common Closure Principle) — принцип согласованного изменения. Данный принцип эквивалентен принципу SRP с тем отличием, что применяется относительно компонента. Таким образом, данный принцип означает, что компонент должен включать в себя те классы и модули, требования к которым могут изменяться по запросу одного и того же актора. В случае, если классы связаны настолько, что могут изменяться только вместе, они должны принадлежать одному компоненту. Данный принцип схож с принципом OCP, поскольку предписывает объединять в один компонент классы, закрытые для одного и того же вида изменений. С точки зрения OCP данный вид изменений определяется за счет ожиданий и опыта разработчика.
* CRP (Common Reuse Principle) — принцип совместного повторного использования — гласит, что компонент должен включать классы и модули, используемые совместно. В компонентах ожидается увидеть классы, имеющие множественные зависимости друг от друга. Основное отличие данного принципа от предыдущих в том, что он четко определяет, какие классы не должны включаться в один компонент. В случае, если компонент зависит хотя бы от одного класса другого компонента, он зависит от всего компонента. Это означает, что в компонент должны включаться классы таким образом, чтобы при зависимости от одного из них нельзя было не зависеть от других. Таким образом, независимые друг от друга классы должны лежать в разных компонентах. Данный принцип следует из обобщения принципа ISP для классов.

Данные принципы противоречат друг другу, так как два из них стараются сделать компоненты как можно крупнее, в то время как третий требует разделения компонентов на более мелкие. В случае уделения большого внимания принципам REP и CCP происходит слишком много ненужных выпусков компонентов, но достигается удобство пользователей и сопровождения. В случае комбинации принципов CCP и CRP усложняется повторное использование компонентов, но устраняются лишние выпуски при все том же удобстве сопровождения. Комбинация же CRP и REP дает изменениям возможность затронуть слишком большое число компонентов при удобстве пользователей. Таким образом, задача архитектора состоит в комбинации всех трех принципов так, чтобы наилучшим образом удовлетворить требования разработчиков и клиентов.

Еще одним важным принципом организации компонентов является принцип ацикличности зависимостей (ADP; Acyclic Dependencies Principle). Проблема циклических зависимостей компонентов заключается в том, что при модификации одного из компонентов требуются изменения во всех остальных вне зависимости от уровня компонента. Кроме того, предположим, различные разработчики внесли изменения в абстракции двух различных модулей, связанных между собой (возможно, через некоторое количество связей) ациклической зависимостью, после чего дальнейшее их совместное применение стало невозможно. В таком случае требуется масштабная интеграция изменений в проект, причем данная интеграция затрагивает все модули, связанные с данными. Данное требование вынуждает разработчиков уменьшить длительность разработки перед интеграциями, чтобы избежать больших изменений, что негативно сказывается на эффективности команды.

Решением в данном случае является разделение компонентов на разные выпуски. В случае внесения изменений в компонент, предыдущая версия сохраняется в репозитории, после чего команда продолжает заниматься разработкой новых версий. Разработчики других компонентов могут продолжить использовать старую версию пока не будут готовы провести интеграцию. В таком случае ни одна из команд разработчиков не зависит от другой. Однако стоит учитывать, что не для всех компонентов такое возможно.

Другим, более универсальным решением, является избавление от циклической зависимости. Существует два метода разрыва цикла:

* Применение принципа инверсии зависимостей. Пусть компонент А зависит от некоторых классов в компоненте Б. В таком случае мы можем создать для этих классов абстракции, которые будут помещены в компонент А. Таким образом, зависимость будет направлена в противоположную сторону.
* Создать новый компонент. В таком случае все классы, от которых зависит компонент А, будут вынесены в некоторый компонент В, от которого будут зависеть компоненты А и Б.

Данное решение предполагает зависимость структуры компонентов от требований к продукту. Из этого можно сделать вывод, что оптимальной является стратегия проектирования архитектуры «снизу вверх», что позволяет определять требования, которые будут появляться в процессе разработки конкретных компонентов, и изменять согласно им структуру их взаимодействия.

SDP (Stable Dependencies Principle) — принцип устойчивых зависимостей — гласит, что зависимости должны быть направлены в сторону устойчивости. Дизайн не может быть статичным, соответственно, некоторые компоненты изначально проектируются как изменяемые. В случае, если устойчивые компоненты зависят от изменяемых, изменяемые компоненты также будет сложно изменять. В случае, если от компонента зависят некоторые компоненты, то говорят, что он несет ответственность за эти компоненты. В случае, если компонент не зависит от других компонентов, такой компонент называется независимым. Компонент, которые независим, считается устойчивым. Точную меру устойчивости рассмотрим подробнее в следующем пункте.

SAP (Stable Abstractions Principle) — принцип устойчивости абстракций. Опять же, рассмотрим в следующем пункте [[2](#Чистая_архитектура)].

**2.4 Метрики архитектуры программного обеспечения**

Для оценки сложности программного обеспечения используются такие метрики, как цикломатическая сложность и связность. Отметим, что для сбора данных по данным метрикам используется статический анализ кода и конфигурационных файлов, при котором не требуется запуск программы.

Для определения цикломатической сложности требуется построить граф потока управления программы либо модуля, который на данный момент анализируется. Узлы графа представляют собой группы команд, которые выполняются последовательно вне зависимости от потока управления. Цикломатическая сложность характеризуется количеством линейно независимых путей выполнения программного кода. Модель может быть построена двумя путями: в первом случае ориентированные дуги означают передачу управления в потоке между блоками кода, в то время как во втором узлами графа являются точки входа и выхода анализируемого участка кода, а дуги обозначают маршрут выполнения программы, при этом все точки входа соединены со всеми точками выхода. В таком случае граф является сильносвязным, а цикломатическая сложность исследуемого участка кода равна цикломатическую числу данного графа потока управления. В случае, когда программа содержит по одной точке входа и выхода, цикломатическая сложность равна числу узлов ветвления, увеличенному на 1. Верхнее ограничение по данному параметру, разработанные Томасом Дж. Маккейбом, составляет значение 10 для модуля, однако в некоторых случаях может быть повышено до 15. При достижении данного значения модуль рекомендуется разбивать на модули меньшего объема.

Параметр «связность» определяется как мера того, насколько класс соответствует поставленной задаче, для которой он был создан, и насколько разные области данная задача затрагивает. Класс, с параметром связности, равным 1, выполняет одну команду. Класс с параметром 0 выполняет все команды, которые можно найти в программе, независимо от их семантики.

Для измерения связности существует ряд разнообразных метрик:

* Метрики, базирующиеся на интерфейсе (Interface-based metrics). Данные метрики оценивают связность по информации, которую можно найти в сигнатурах методов класса. Наиболее часто используемые из них приведены в таблице 2.1.

Таблица 2.1 — Метрики, базирующиеся на интерфейсе

|  |  |  |
| --- | --- | --- |
| Название метрики | Краткое описание | Формула |
| CAMC | Связность между методами класса (Cohesion among methods of classes), определяет связность методов по передаваемым параметрам | (2.1),  l — число типов параметров, k — число методов, a — общее число различных типов параметров методов класса |
| NHD | Нормализованной расстояние Хэмминга (Normalized Hamming distance), определяет разницу между методами по передаваемым параметрам | xj — число методов, которые имеют тип параметра j |
| MMAC | Method-method through Attributes Cohesion, подсчитывает разницу между парами методов в зависимости от их атрибутов | = |

* Метрики, основанные на коде (Code-based metrics). Данные метрики, в свою очередь, могут быть классифицированы на концептуальные и структурные. Концептуальные метрики используют методы извлечения информации, основанные на исследовании концепций исходного кода, для подсчета связности. Если методы класса считаются концептуально связанными, то класс имеет высокий показатель связности. Концептуальные метрики приведены в таблице 2.2.

Таблица 2.2 — Метрики, основанные на коде

|  |  |
| --- | --- |
| Название | Формула и определение |
| LORM | Логическая связность методов (Logical relatedness of methods), определяется отношением общего числа внутренних зависимостей класса к числу возможных внутренних зависимостей, где внутренняя зависимость — вызов из одного метода класса другого метода этого класса, соответственно, число возможных зависимостей определяется как , где N — число методов класса |
| C3 | Концептуальная связность классов (Conceptual cohesion of classes), базируется на анализе тестовой информации в методах, включая комментарии и названия переменных. Пусть — набор методов класса, — вектора, относящиеся к mi соответственно. Тогда  *,* где  , а , |
| LCSM | Отсутствие концептуального сходства между методами (Lack of conceptual similarity between methods), пусть |
| MWE | Используется, когда можно каким-то образом выделить темы (семантические группы), которые распределены (либо могут быть распределены) по тексту методов. Для определения данных групп может использоваться алгоритм LDA (Latent Dirichlet Allocation), после чего, используя информацию об энтропии групп, становится возможно вычислить вероятность, с которой данный метод принадлежит данной группе. После определения данных вероятностей — вероятность того, что тема ti имеет место в методе d,  *, ,*  *, |t|* — число тем, n — число методов. |

Структурные методы основаны на использовании структурных данных, извлеченных из исходного кода. Разница между структурными метриками заключается в определениях отношений между методами. Выделяют четыре подтипа структурных методов со своими методами оценки связности:

1. Метрики, основывающиеся на непересекающихся компонентах (Disjoint component-based metrics), которые подсчитывают число непересекающихся наборов методов или атрибутов в классе. Данные методы представлены в таблице 2.3:

Таблица 2.3 — Метрики, основанные на непересекающихся компонентах

|  |  |
| --- | --- |
| Метрика | Определение |
| LCOM1 | Число пар методов, которые не используют одинаковые атрибуты |
| TLCOM | Число пар методов, которые не используют напрямую либо транзитивно общий атрибут |
| LCOM3 | Число связанных компонент в графе, узлы которого представляют собой методы, а ребра — совместное использование хотя бы одного атрибута |
| LCOM4 | Схож с предыдущим, но ребра также добавляются, если один метод вызывает другой или наоборот. |

1. Метрики, основанные на попарных соединениях (Pairwise connection-based metrics). Данные метрики представляют собой функцию от чисел связанных и непересекающихся пар методов. Введем понятия прямых и косвенных связей между методами. Два метода имеют прямую связь, если содержат хотя бы одну совместно используемую переменную. Методы, которые неявно использую общую переменную (путем вызова методов, её использующих хотя бы одной из сторон), имеют связь косвенную. Три наиболее используемых метрики приведены в таблице 2.4:

Таблица 2.4 — Метрики, основанные на попарных соединениях

|  |  |
| --- | --- |
| Метрика | Определение |
| LCOM2 | Пусть P — число пар методов, которые не разделяют атрибуты, Q — число тех, которые разделяют. Тогда |
| TCC | Пусть NDC — число пар напрямую связанных методов, NP — наибольшее возможное число прямых или косвенных связей. Тогда |
| LCC | Пусть NIC — число пар косвенных связей в классе. Тогда |

1. Метрики, основанные на характеристиках соединения (Connection magnitude-based metrics). Данные методы подсчитывают некоторый индекс совместного использования, используя число обращения к атрибуту класса среди его методов и общее число методов, но при этом не учитывают, какие конкретно методы используют тот или иной атрибут. Некоторые из них приведены в таблице 5:

Таблица 2.5 — Метрики, основанные на характеристиках соединения

|  |  |
| --- | --- |
| Метрика | Определение |
| LCOM5 | Пусть l — число атрибутов, k — число методов, a — суммарное число обращений к атрибутам класса из его методов. Тогда |
| Coh | Определения совпадают с изложенными выше, |
| CC | Пусть Ii и Ij — наборы атрибутов, к которым обращаются методы mi и mj­ соответственно. Тогда сходство данных методов, обозначаемое |

1. Метрики, основанные на декомпозиции (Decompose-based metrics). Данные методы представляют собой рекурсивное удаление из класса необходимых элементов, которые делают его методы связанными. Некоторые методы приведены в таблице 2.6:

Таблица 2.6 — Метрики, основанные на декомпозиции

|  |  |
| --- | --- |
| Метрика | Определение |
| ICBMC | Пусть G — граф зависимостей между методами. Тогда  Q — число ребер в срезе графа G, Nm(G) — число методов в графе G (без специальных), Nv(G) — число атрибутов в графе G |
| CBMC | *,* , где M(G) — число связывающих методов в графе G (методов, при удалении которых граф перестает быть связным), N(G) — число методов в графе G, не являющихся специальными, Gi — одна из компонент графа. Тогда |

Сцепление (зацепление; coupling) — способ и степень взаимозависимости между классами. Данный параметр оценивается подобно рассмотренному выше сцеплению модулей, за исключением того, что оценивается взаимосвязанность классов между собой [[6](#Метрики_связности)].

Исходящие зависимости (efferent coupling) — число классов, от которых зависит данный. В данном случае равен числу классов, методы которых вызываются из данного. Данная характеристика, как и входящие зависимости (Afferent coupling) обычно вычисляется для пакетов, в таком случае рассматривается число пакетов, от которых зависит данный либо которые зависят от данного.

Сложность (Complexity) — оценивается как характеристика программы или модуля, также рассмотрена выше в контексте модульности программы.

Устойчивость (Instability) — демонстрирует, насколько просто класс может быть подвержен изменениям. Пусть класс имеет число входящих зависимостей Fan-in и число выходящих Fan-out. Тогда мера неустойчивости I равна

|  |  |  |
| --- | --- | --- |
|  |  | (2.10) |

Если метрика равна 1, то другие компоненты не зависят от данного, и при этом данный компонент зависит от всех остальных. Данный компонент должен быть готов к изменениям в сторонних компонентах, но при этом изменения в нем самом никак не затрагивают остальные. При этом если метрика равна 0, то имеет место обратная ситуация: изменения в других компонентах не затрагивают данный, но при этом изменения в данном компоненте нежелательны. Принцип устойчивых зависимостей гласит, что все зависимости должны быть направлены в сторону большей устойчивости.

Абстрактность. Данная метрика определяется как число абстрактных классов и интерфейсов в компоненте, деленное на общее число классов. Таким образом, можно определить прямую, на которой должны располагаться компоненты согласно графику, отражающему зависимость между устойчивостью и абстрактностью:

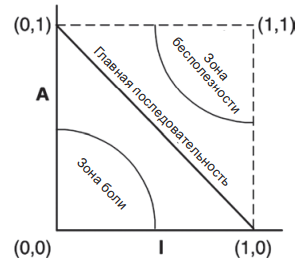


Рисунок 2.3 Определение оптимальной прямой для компонента

Таким образом, существует ряд характеристик, позволяющих с математической точностью оценить качество кода по ряду параметров. Задачей архитектора является нахождение компромисса между различными характеристиками создаваемой системы в зависимости от специфики задачи и требований, предъявляемых к системе.

**ВЫВОДЫ**

1. Рассмотрены основные численные метрики архитектуры классов.
2. Описаны современные методы внедрения зависимостей и принципы организации компонентов, их влияние на метрики.
3. Приведены расчетные формулы для метрик, методики их подсчета.

глава 3. проектирование анализатора архитектуры java-приложений

3.1 Обзор анализаторов архитектуры Java-приложений

Существует несколько уже реализованных решений для анализа архитектуры Java-приложений:

1. JDepend.
2. Code Maat.
3. CodeMR.

JDepend ориентирован на построение набора метрик конкретного пакета классов. Предоставляет информацию о числе классов и интерфейсов, входящим и исходящим зависимостям по отношению к другим пакетам, абстрактности, устойчивости и дистанции от главной последовательности пакета. Также может определить наличие циклов в зависимостях между пакетами. Генерирует отчеты в виде файлов с расширениями txt либо xml. Интерфейс представлен на рисунке 3.1:

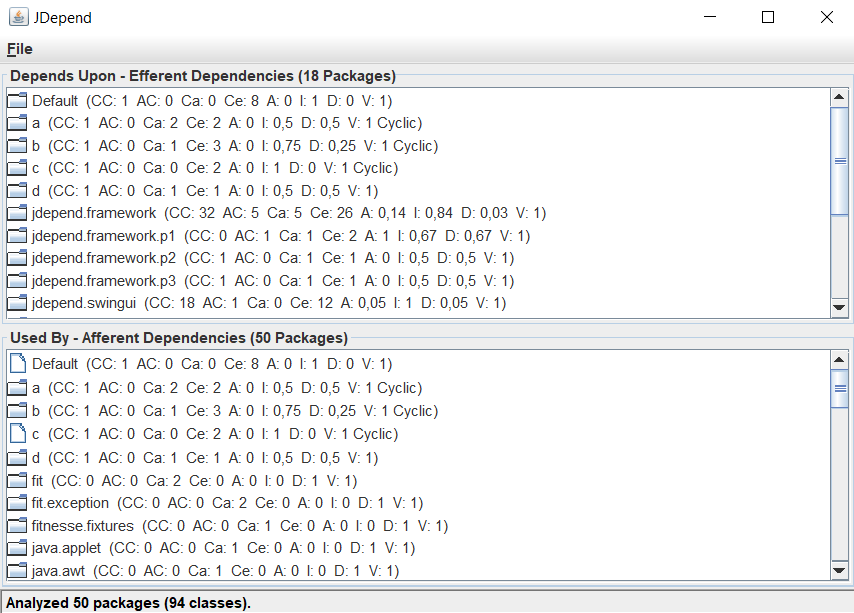


Рисунок 3.1 Интерфейс JDepend

К недостаткам данного решения можно отнести отсутствие компонентов кроме пакетов, что допускает множество проблем в локальных классах. Также не учитывается семантика классов либо методов, нет рекомендаций для изменений или подробного описания свойств того или иного компонента.

Code Maat в основном предназначен для анализа внесенных в проект изменений за счет использования логов некоторых систем контроля версий. Анализируются авторы изменений, возраст кода, и, самое важное — сцепление между модулями. В качестве модулей выступают Java-классы, также производится расчет вероятности того, что при изменении определенного модуля понадобятся изменения в другом, который имеет входящую или исходящую зависимость от данного. Имеет только консольный интерфейс.

К недостаткам данного анализатора можно отнести то, что компонентами выступают только классы, единственная характеристика архитектуры — сцепление, а также то, что в случае отсутствия изменений в некоторых модулях за определенный период (т. е., в случае отсутствия упоминания о них в логах системы контроля версий) анализатор перестает учитывать зависимости с ними.

CodeMR является, пожалуй, эталонным решением. Он включает в себя анализатор структуры пакетов, а также подсчитывает для каждого из модулей связность, сцепление, сложность, а также ряд иных характеристик, рассмотренных в главе 2. Также данный анализатор имеет плагины для интеграции с наиболее популярными IDE, предлагает настраиваемую визуализацию полученных данных. Также позволяет настроить границы характеристик и уведомить пользователя о том, если некоторые компоненты программы им не удовлетворяют.

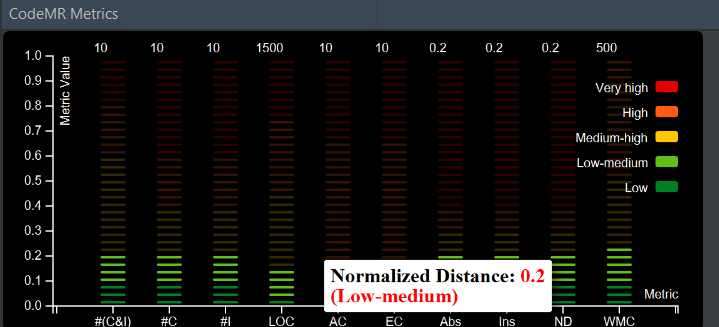


Рисунок 3.2 Плагин CodeMR

К недостаткам можно отнести отсутствие рекомендаций по изменению кода, ограничение функционала подсчетом метрик. Также стоит учитывать, что решение является коммерческим, поставляется на платной основе и скрывает подробности реализации.

3.2 Функциональные требования анализатора Java-приложений

Спектр требований, предъявляемых к анализатору java-приложений, зависит от числа метрик архитектуры, которые должен поддерживать анализатор, того, какие варианты компонентов он должен поддерживать, а также должен ли он давать рекомендации и обладать графическим интерфейсом.

Сформулируем функциональные требования следующим образом:

1. Анализатор должен представлять собой модуль, функционирующий ‘независимо от использующей его системы, и предоставлять на выходе некоторым образом сформированные отчеты, которые позволят системе строить их представление удобным для пользователя образом.
2. Анализатор должен уметь вычислять метрики компонентов анализируемой системы, которые описаны в главе 2.
3. Анализатор должен уметь обнаруживать циклические зависимости в анализируемой системе и предоставлять информацию об обнаруженном цикле.
4. Анализатор должен уметь работать как с отдельными файлами, в таком случае его функциональность ограничивается представлением отдельных классов в виде компонент, так и с некоторыми пакетами классов.
5. Анализатор должен поддерживать различные определения компонентов системы, а именно класс, пакет, модуль.

При разработке требований использовались методы опроса предполагаемой аудитории на тему наиболее востребованных функций, ожидаемых от анализатора, а также исследование функционала существующих решений.

При этом стоит отметить, что хотя анализатор сам по себе может использоваться как отдельное приложение, но при этом для пользователя больший интерес представляет система, использующая его как независимый модуль, так как именно с ней приходится взаимодействовать. В большинстве случаев, такой системой является анализатор более широкого профиля, который включает в себя функциональность, которая выходит за рамки анализа архитектуры приложения. Так как анализатор, реализованный в качестве практической составляющей данной работы, ориентирован на использование его в качестве составляющей подобных систем, было проведено исследование, направленное на выявление требований к системам данного профиля, с целью выяснить наличие либо отсутствие противоречий между требованиями.

В качестве первого шага исследования был проведен опрос среди предполагаемой аудитории подобных систем (практикующих разработчиков) на тему требований, которые они предъявляют к подобным системам. Результат опроса представлен на рисунке 3.3:

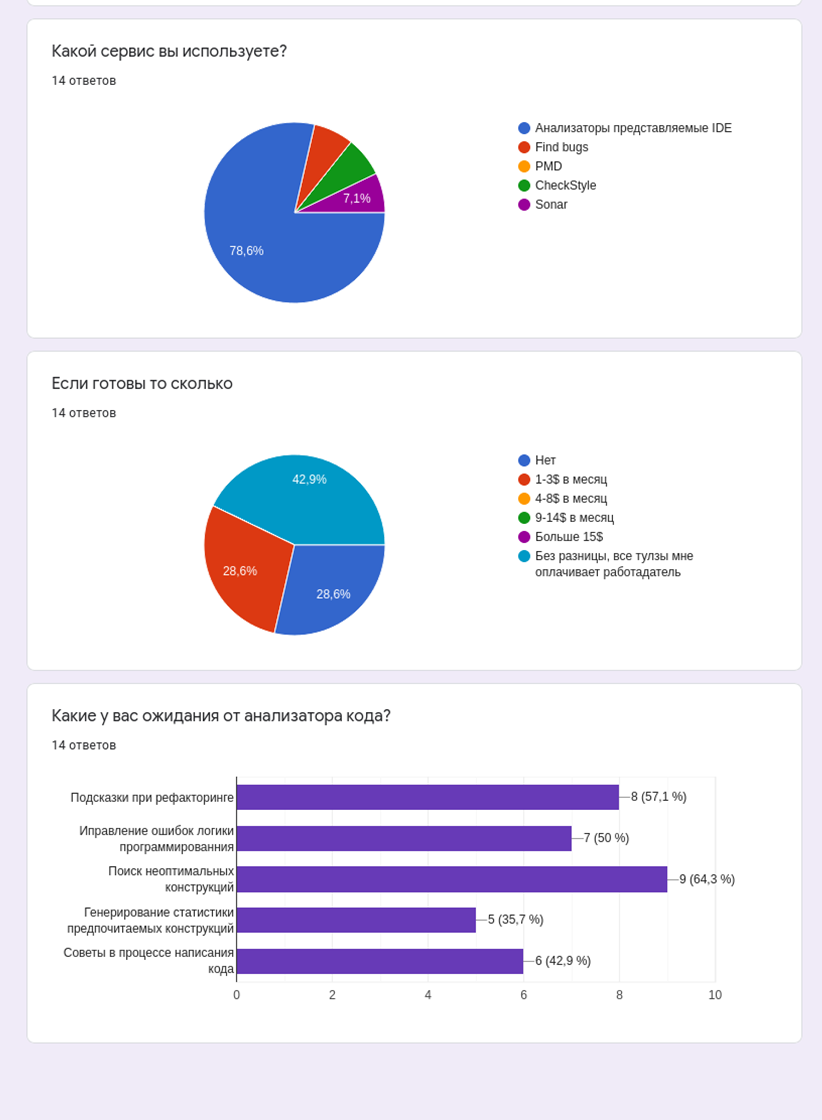


Рисунок 3.3 Результат опроса о требованиях

В результате исследования были получены следующие профили задач системы:

1. Регистрация.
2. Аутентификация.
3. Добавление нового файла.
4. Добавление репозитория.
5. Просмотр всех ранее добавленных файлов и репозиториев для анализа.
6. Просмотр общей статистики.
7. Просмотр статистики по конкретному документу.

Как видно из профилей задач и требований к анализатору архитектуры, рассмотренных выше, реализация анализатора влияет на вопрос реализации решений задач под пунктами 3, 4, 7, также за счет решений, которые будут применены при разработке анализатора, определяется вид решения пунктов 6,7. Рассмотрим подробнее требования, предъявляемые к системе. Для этого рассмотрим ряд её характеристик:

* Заинтересованные стороны. В данном случае в качестве заинтересованной стороны выступают разработчики программного обеспечения на языке программирования Java.
* Видение продукта заинтересованными лицами: в первую очередь, интересует быстрое получение метрик предоставленного кода, возможность загрузки файлов либо кода из репозитория, а также внятное представление измеренных характеристик.
* Конфликты и противоречия. В качестве конфликта между запросами пользователя и предоставляемым функционалом можно отметить отсутствие открытого API в ряде VCS систем, наличие собственных метрик качества кода в различных компаниях, а также разногласия по дизайну и формату предоставляемых проектов.

Основной вариант использования с точки зрения пользователей выглядит подобным образом:

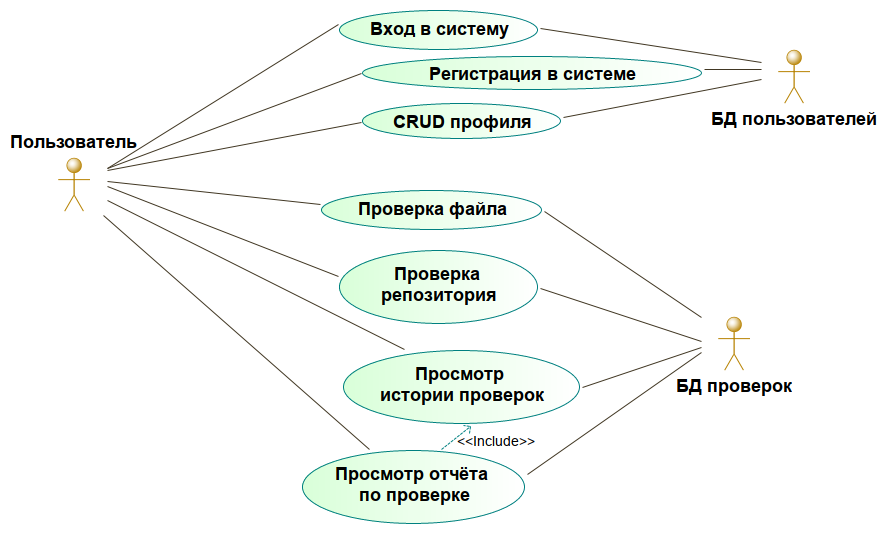


Рисунок 3.4 Диаграмма варианта использования

Таким образом, основными задачами для удовлетворения требований системы, которые стоят перед анализатором, является обеспечение высокой скорости вычислений, поддержка загрузки одновременно нескольких файлов и предоставление результатов вычислений в формате, который включает в себя все необходимые данные для его преобразования в формат, выбранный системой для отображения пользователю.

3.3 Проектирование анализатора Java-приложений

Построение анализатора, удовлетворяющего требованиям из параграфа 3.2 можно разбить на следующие подпункты:

* Реализация парсера файлов, содержащих в себе код на языке Java, и преобразующего код в представление, которое возможно обработать с целью получения характеристик, описанных в параграфе 3.2.
* Реализация бизнес-логики, отвечающей за обработку представления java-кода и подсчет метрик архитектуры.
* Реализация интерфейса анализатора, который позволит вызывать его функционал из некоторой системы либо использовать его независимо.

Рассмотрим каждый из пунктов более подробно, определив влияние требований на каждом этапе:

К первому этапу предъявляются требования, которые касаются возможности обработки проекта, состоящего из нескольких файлов, а также скорости обработки. С точки зрения архитектора, наиболее оптимальным вариантом является построение максимально гибкой масштабируемой системы, которая подразумевает разделение данного модуля на довольно большое число компонентов: компонент, отвечающий за разбор кода из файлов, компонент, отвечающий за хранение построенной модели, компонент, отвечающий за вычисления, проводимые над данной моделью, и, наконец, компонент, отвечающий за предоставление данных пользователю (в данном случае, совпадающий с контроллером).

Однако данное разделение подразумевает проведение вычислений над обработанными на первом шаге данными, что означает, что обращение к данным произойдет удвоенное число раз: при построении модели для вычислений и при дальнейшем её использовании. Таким образом, для обеспечения требования по оптимальному времени расчетов, необходимо производить вычисление той части статистики, которая может быть выполнена без построения модели, на этапе обработки кода.

Данный факт подразумевает собой внедрение зависимости компонента обработки файлов не только от модели представления кода, но и от модели статистики, а также от вычислительного модуля. Сразу отметим, что несмотря на ухудшение структуры из-за утери графом зависимостей древовидной структуры, зависимости сохраняют направление, однако требуется инверсия зависимостей, направленной на модель статистики, так как она может быть подвержена изменениям. При этом, с учетом высокой ожидаемой связности модуля (модуль направлен на выполнение одного действия, которое несмотря на возможное разбиение на подпункты не подразумевает их выполнение отдельно друг от друга) внедрение новых зависимостей не должно повлиять на ухудшение масштабируемости модуля.

Реализация бизнес-логики подразумевает собой реализацию внешнего контроллера, который управляет последовательностью действий, а также вычислительного компонента. Контроллер по определению должен зависеть от класса (компонента), который занимается обработкой файлов и компонента, который занимается вычислениями с построением модели. Также контроллер может зависеть от модели статистики, для обеспечения её передачи пользователю, однако в это может быть выражено зависимостью либо от названия модели, либо от её представления.

Компонент, занимающийся вычислениями, зависит от модели представления файлов и модели статистики. Здесь стоит отметить, что любые изменения в статистической модели (например, добавление новых метрик, которые необходимо представить пользователю) неизбежно вызывают изменения вычислительного компонента, так как даже при использовании инверсии зависимостей модели понадобится расширение, если, конечно, не использовать для хранения результатов некоторый обобщенный формат (например, использовать словарь ключ-значение, ключ в котором представляет собой название метрики, значение — её представление в читаемом формате. Очевидно, что подобная организация не требует изменения структуры модели при изменении вычислений), однако использование подобных конструкций сказывается на читаемости кода и влияет на удобство использования модели.

Таким образом, модель статистики с вычислительным компонентом объединяются в один компонент, при этом компонент изменяемый. Однако, две из трех зависимостей от данного компонента не могут быть изменены при его расширении (зависимость компонента, отвечающего за преобразование файлов, подразумевает заполнение модели определенным заранее заданным набором данных, в то время как контроллер вызывает обобщенный метод наподобие «вычислить» вычислительного модуля, подробная реализация которого от него скрыта. Таким образом, от модели зависит лишь её преобразователь в требуемый формат). Это означает, что компоненту можно очертить довольно четкие границы, удостоверившись, что общие методы, которые могут быть вызваны от него независимо, вынесены в отдельный компонент.

Интерфейс анализатора, который занимается представлением данных пользователю, может быть реализован в виде части статистической модели, которая предоставляет некоторый обобщенный формат хранимых в ней данных, либо представлять собой контроллер, поскольку контроллер будет в любом случае иметь от неё транзитивную зависимость. Однако, реализация данного интерфейса контроллером, подразумевает усиление зависимости, так как теперь контроллер будет напрямую зависеть от внутренней реализации модели статистики, опять-таки при условии отказа от обобщенного представления.

Итоговая диаграмма компонентов может быть представлена следующим образом:

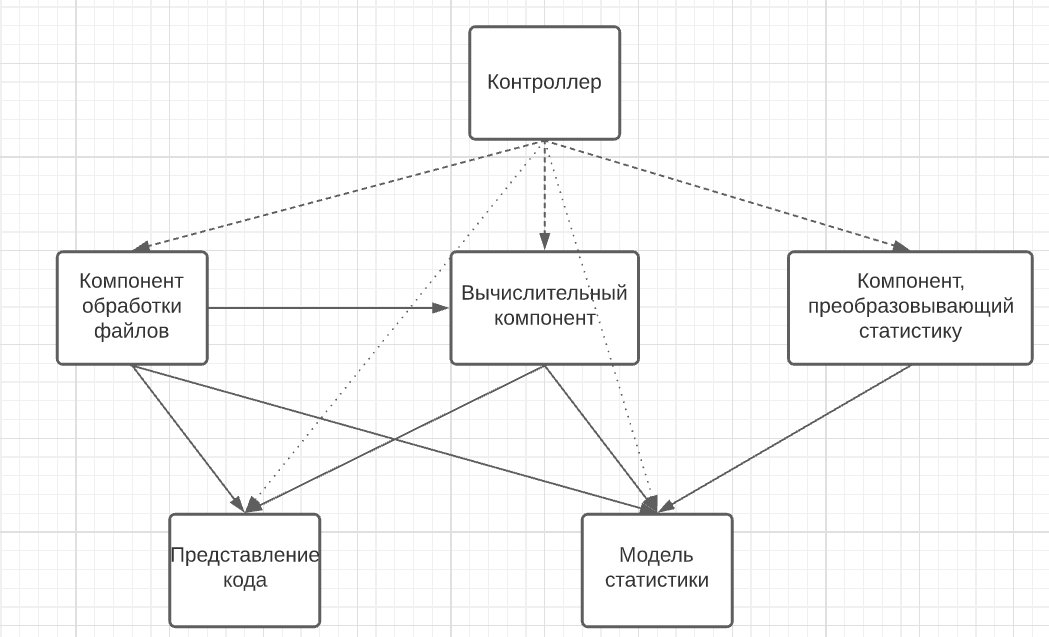


Рисунок 3.5 Диаграмма компонентов

Для успешного сбора и обработки всех необходимых данных требуется на основе предложенной библиотекой Spoon модели создать собственное представление компонентов. Диаграмма классов приведена на рисунке 3.6:

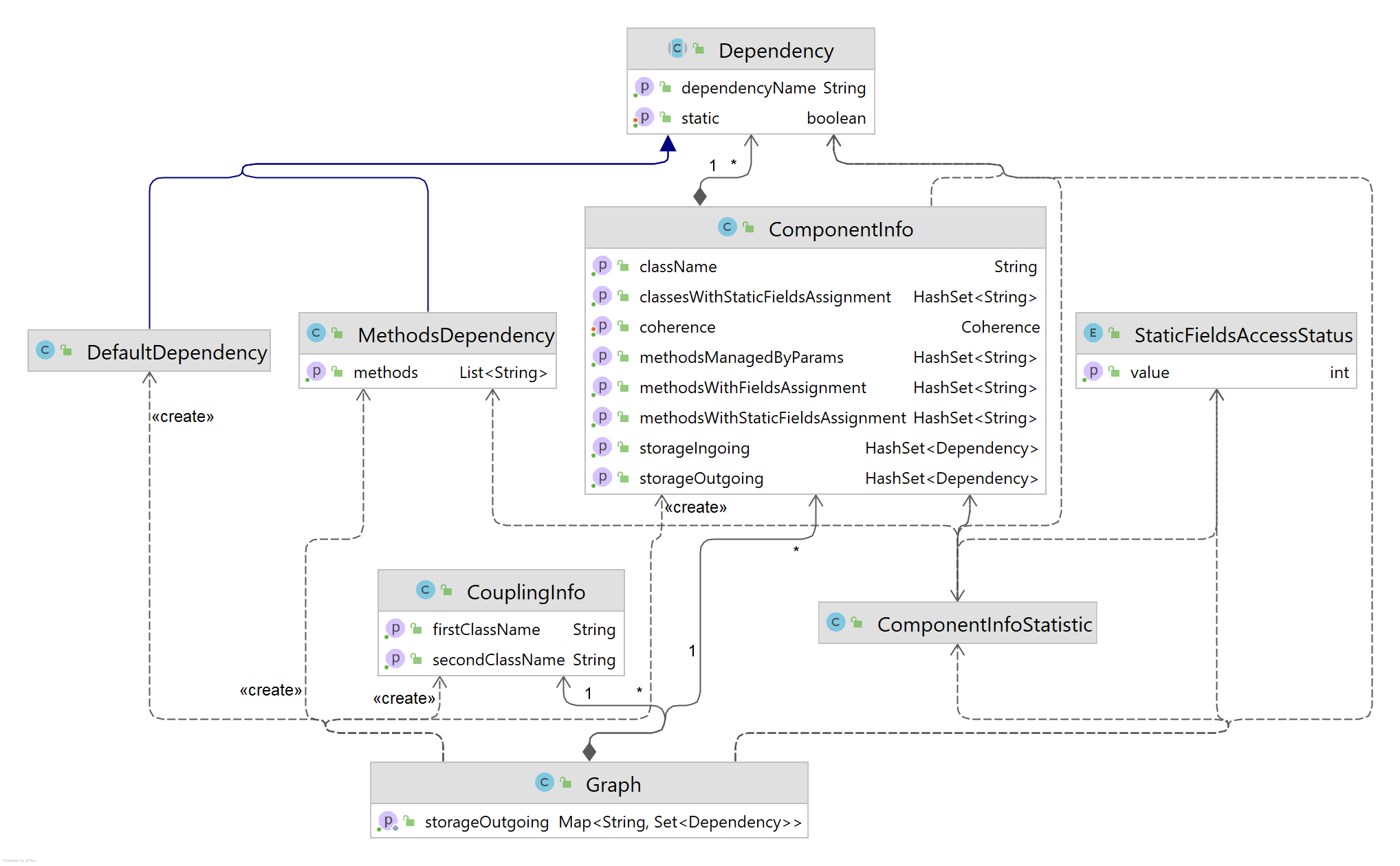


Рисунок 3.6 Диаграмма классов хранилища

**ВЫВОДЫ**

1. Рассмотрены типы архитектуры ПО, дано определение модульности.
2. Рассмотрены преимущества модульности относительно других подходов, описаны свойства и представления модулей.
3. Кратко описаны нововведения парадигмы ООП, ее преимущества.
4. Приведены описания принципов SOLID в контексте практического значения для анализа архитектуры кода.

глава 4. реализация анализатора архитектуры java-приложений

4.1 Библиотека Spoon

Для преобразования файлов с кодом в некоторую читаемую модель воспользуемся библиотекой Spoon, расширив её необходимым образом.

Spoon представляет собой библиотеку с открытым исходным кодом на языке программирования Java, предназначенную для построения метамодели программы, анализа и трансформации Java-кода. Данная библиотека производит действия, схожие со стандартными преобразованиями компилятора, производя статический анализ кода и строя абстрактное синтаксическое дерево, узлы в котором представлены как пакетами, классами и интерфейсами, так и элементами внутренней реализации классов, такими как блоки кода, ветвления и циклы.

Метамодель состоит из трех частей: структурная, содержащая объявления программных элементов, часть, посвященная коду, которая содержит в себе информацию о наполнении программных элементов, и часть, содержащая информацию о связях, например, инициализацию переменной определенного типа. Пользовательская обработка узлов дерева осуществляется за счет определения пользовательских классов-обработчиков (Processors), каждый из которых направлен на обработку узла определенного типа.

Таким образом, обработка происходит за счет применения двух паттернов проектирования: посетителя, который позволяет проверить применимость каждого класса-обработчика к определенному узлу и провести над ним некоторые манипуляции, выполнив обход по построенному синтаксическому дереву, и цепочки обязанностей — паттерна, который обеспечивает последовательное прохождение каждого узла по цепочке обработчиков, пока все обработчики не будут выполнены либо проверены на невозможность их выполнения над данным типом узлов. Для реализации данной проверки используется рефлексия над типизированным обработчиками.

Отметим ряд особенностей анализатора, вызванных использованием данной библиотеки.

Обработка узлов происходит уже после построение метамодели, что означает, что вся статистика, собранная по проекту, вычисляется без разбиения на этапы предварительных вычислений (на этапе построения дерева, что позволило бы оптимизировать количество обходов в случае, если пользователь не нуждается в метриках, которые требуют повторного обхода узлов дерева) и этап обработки уже построенной модели.

Несмотря на то, что вычисления производятся после построения модели, сохраняется зависимость между компонентом-обработчиком исходного текста программы и вычислительным компонентом, так как компонент-обработчик представлен внешней библиотекой, которой необходимо передать объекты, производящие вычисления над узлами. Вычислительный же компонент представляет собой расширения классов (которые могут также вызывать некоторую внешнюю логику) библиотеки. Однако, стоит отметить, что имеет место инверсия зависимостей за счет использования абстракции обработчика узлов дерева, так как вычислительный компонент не может измениться таким образом, чтобы вызвать изменения кода библиотеки, в то время как вычислительный компонент зависит от её реализации. Данный риск можно считать оправданным, так как, данная версия библиотеки обладает всем необходимым функционалом и мигрировать проект на новую версию библиотеки в обозримом будущем нет необходимости, класс обработчика представляет собой часть классической реализации паттерна проектирования посетитель и представить причину, которая потребует изменения его реализации довольно сложно, а также потому что выполняется принцип изменения поведения компонента не изменяя его структуры. В последнем пункте подразумевается, что любое изменение в библиотеке, которое потребует изменение кода, с ней взаимодействующее, означает отказ от данного принципа при её проектировании, что пойдет вразрез с концепцией, которой придерживались разработчики на всем этапе её поддержки. Таким образом, данные последствия можно оценивать как маловероятные. Также стоит отметить, что вычислительный компонент зависит лишь от абстракций компонента, представленного библиотекой, а абстракции мало подвержены изменениям.

Метамодель представляет собой часть библиотеки. В случае, если потребуется обработка, которая требует данных, полученных в результате работы процессоров (т. е., некоторый собственный вариант обработки), то данная операция будет производиться после построения метамодели и после выполнения обработчиков, представляющих собой расширение библиотеки. Для реализации данной логики потребуется новый вычислительный компонент, который будет зависеть уже не от абстракций, а от конкретных классов библиотеки, поскольку для запроса данных из стандартной метамодели наиболее удобно использовать методы, предоставляемой самой библиотекой. В качестве решения можно произвести построение собственной модели (например, в виде графа) на этапе обработки узлов дерева. В данную модель войдут конкретно те узлы и данные, которые потребуются на этапе последующей обработки, а значит, для их извлечения не потребуется дополнительного функционала, предоставляемого библиотекой [[8](#Spoon)].

4.2 Алгоритм обработки графа

В данном параграфе рассматривается алгоритм, который не включает в себя базовый подсчет характеристик того или иного узла дерева, которое строит библиотека Spoon.

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. Стоит отметить, что библиотека рассматривает в качестве зависимостей в том числе примитивные типы переменных, а также классы стандартной библиотеки Java. 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. После обработки циклы в зависимостях как классов, так и пакетов выводятся пользователю.

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. Основная сложность состоит в определении, какие из классов в принципе могут быть вынесены в отдельные компоненты, потому что, как описывалось выше, распределение по компонентам является одной из составляющих масштабируемой системы. Для выделения компонент воспользуемся рядом принципов, описанных в главе 2:

* Классы, которые имеют общие зависимости либо зависят друг от друга, логичнее располагать в одном компоненте.
* Классы, реализующие управление потоком выполнения программы, должны зависеть от абстрактных классов и интерфейсов, а не от конкретных реализаций.
* 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.
* Зависимости должны быть направлены в сторону большего значения стабильности.

Приведенные выше принципы позволяют разработать алгоритм, который будет использовать описанное выше представление программы и давать приблизительное представление того, как обрабатываемые классы могут быть разбиты по компонентам. Алгоритм состоит из следующих шагов:

1. Построить пары компонентов (классов либо пакетов), которые имеют зависимости друг от друга в виде наследования, типа полей, переменных либо вызовов статического метода другого класса.
2. Удалить из построенных пар те, где внедренная зависимость является абстракцией.
3. Удалить из зависимостей те, где параметр неустойчивости превышает некоторое пороговое значение.
4. Построить систему непересекающихся множеств, элементами которой является некоторое представление компонент, а принадлежность классов одному множеству определяется наличием зависимостей между ними.
5. 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 представляет собой следствие из принципа направленности зависимостей. То есть, есть некоторые высокоуровневые классы, которые являются определяющими для графа выполнения программы. Классы, которые инициализируются с помощью данных (за счет вызова фабричных методов или присваивания), имеют некоторые внешние зависимости, которые должны быть проинициализированы совместно с экземплярами данных классов. В таком случае передача данных значений в конструктор либо соответствующие методы данных классов генерирует зависимость вызывающего класса от конкретных реализаций, избежать которую невозможно.

В случае, если любая подобная зависимость подразумевает предпочтительное помещение классов в один компонент, то все классы программы будет предложено поместить в один компонент, что не совсем верно, так как классы, которые инициализируются из высокоуровневых, не зависят от конкретных реализаций друг друга. Данные высокоуровневые классы, которые определяют бизнес-логику, отличает низкий параметр устойчивости, так как любое изменение логики программы на них отражается в виде измененных зависимостей. Такие классы, которые «связывают» компоненты друг с другом, имеют единственную зависимость от конкретных реализаций и существуют в коде программы в случае неиспользования некоторых фреймворков, определяющих понятие контекста программы, поэтому их внешними зависимостями можно пренебречь, так как они не связывают свои зависимости друг с другом.

4.3 Модификация анализатора в случае интеграции технологий внедрения зависимостей

С учетом того, что рассмотренные выше технологии используют в современных приложениях, разработанных с использованием Java, анализатор может использоваться для оценки кода приложений как использующих только данные технологии внедрения, так и их комбинации со стандартными средствами языка Java. В таком случае анализатор должен быть подвергнут ряду модификаций, изложенных ниже.

Само понятие зависимости должно быть изменено в случае, если используется внедрение с помощью Bootique. Методы, которые предоставляют бины, являются исключительно конфигурационными и вряд ли могут рассматриваться как зависимость от внутреннего устройства классов. Аналогичная ситуация с реализациями BQModule, в которых используются внедрение напрямую в объект класса Binder. В случае использования Spring при использовании конфигурации с помощью Java-кода (конкретно аннотации Bean над методом, предоставляющим объект для внедрения) подобные зависимости класса, содержащего данный метод, учитываться не должны. При этом заметим, что несмотря на внедрение с помощью аннотаций вместо передачи объектов в конструктор либо сеттеры, анализатор не требует изменений, так как зависимости будут отслежены из-за полей либо параметров, совместно с которыми применяются данные аннотации.

Также отсутствует необходимость в механизме выявления циклических зависимостей среди бинов, так как контекст с бинами не позволяет этого сделать. Однако, данный механизм все равно необходим для подобных зависимостей среди Java-кода, создающего объекты напрямую. В случае, если анализатор будет расширен возможностью предложить возможное решение либо самостоятельно устранить циклические зависимости следует учесть возможность разрыва зависимости с помощью внедрения провайдера данного объекта в случае Bootique. Также стоит учесть возможность внедрения объектов одних и тех же классов разными способами, что не всегда может быть выявлено контекстом. В таком случае наиболее разумно разорвать зависимость, не зависящую от контекста.

В случае зависимостей от конкретных неустойчивых классов вместо их абстракций для увеличения параметра абстрактности кода необходимо не просто создать требуемый интерфейс (абстрактный класс), но и произвести необходимые манипуляции для внедрения его в контекст. Для этого в случае использования Spring Boot достаточно добавления аннотации для интерфейса (в случае, если он расположен в обрабатываемых директориях при использовании контекста требуемого типа, в противном случае требуется расширение xml-файла, добавления напрямую бина в контекст либо расширение набора сканируемых директорий), в случае же Bootique необходимо обнаружить реализацию интерфейса BQModule, которая производит конфигурацию бинов для заданного класса, и изменить передаваемый параметр метода bind. При этом стоит учесть, что объект может внедряться в том числе в составе коллекций.

Таким образом, современные технологии внедрения зависимостей в языке Java позволили сократить их количество, обеспечить механизм внедрения зависимостей в класс без использования некоторого класса на вершине иерархии с наибольшей устойчивостью, зависящего от всех необходимых реализаций (по крайней мере, в случае Spring), исключить циклические зависимости при условии конфигурации бинов исключительно с помощью контекста.

Однако, даже они не обеспечивают гарантии отсутствия архитектурных ошибок — внедрения конкретных реализаций, использования циклических зависимостей в случае комбинации внедрения со стандартными средствами языка (при этом стоит отметить, что циклическая зависимость в бинах теперь является не просто архитектурной ошибкой, а фатальной, неизбежно влекущей за собой ошибку в момент выполнения приложения), серьезных отличий структуры графа зависимостей от древовидной. Разработанный анализатор архитектуры с некоторыми модификациями может использоваться для оценки архитектуры приложений, разработанных с использованием данных технологий, и, в случае расширения его возможностей, предлагать улучшения без использования дополнительных средств.

4.4 Анализ результата разработки на данном этапе

В качестве демонстрации работы приложения, а также для рассмотрения на примере принципов построения архитектуры и результатов их применение проанализируем с помощью созданного приложения исходный код самого приложения.

В ходе работы подсчитываются следующие характеристики:

* Входящие зависимости. Подсчитываются для классов по построенному представлению графа зависимостей, описанному в предыдущем пункте.
* Исходящие зависимости.
* Неустойчивость — согласно формуле, приведенной в главе 2.
* Абстрактность — подсчитывается с помощью обработчиков, выявляющих среди узлов дерева, построенного с помощью библиотеки Spoon, абстрактные классы и интерфейсы.
* Цикломатическая сложность — в качестве формулы подсчета с учетом того, что данное приложение имеет единственные точки входа и выхода, а также его однопоточность, для простоты взято количество разветвлений управляющего графа программы, увеличенное на 1.
* Выявляются циклы среди зависимостей на уровне классов и пакетов. Детальный разбор подсчет приводится в предыдущем параграфе.
* Выявляется возможное разбиение на компоненты. Аналогично предыдущему пункту.

Наибольший показатель неустойчивости, равный 1.0, имеет класс Main, содержащий в себе точки входа и выхода программы, что ожидаемо, так как данный класс реализует бизнес-логику наиболее высокого уровня. Наименьший показатель свойственен реализация некоторых абстракций, которые не имеют внешних зависимостей и предоставляют, фактически, бизнес-логику низшего уровня. К таковым, например, относятся класс DefaultDSU, представляющий собой стандартную реализацию системы непересекающихся множеств, а также класс AbstractionFilter, который является реализацией интерфейса Filter библиотеки Spoon и предназначен для подсчета количества абстракций в исходном коде.

Для примера не совсем удачного архитектурного решения рассмотрим класс Graph, который сочетает в себе некоторую бизнес-логику, связанную с обработкой представлений графов зависимостей программы, и, собственно, хранение данных, которые проходят обработку. Данный класс имеет шесть входящих и две исходящие зависимости, что означает, что граф зависимостей программы на уровне классов по структуре далек от дерева.

В качестве возможного решения можно отметить вынесение данных, которые содержит класс Graph, в отдельный компонент, либо использование иного компонента для организации бизнес-логики над данными. Данные компоненты будут иметь только входящие (в первом случае) либо только исходящие (во втором случае, если не считать зависимости, исходящие из наиболее устойчивых классов) зависимости, что снизит число путей, ведущих в узел графа зависимостей, соответствующий данному классу.

В ходе анализа выявлен циклический путь между пакетами storage и processors, причиной которой является класс AbstractionFilter. Классы-обработчики, хранящиеся в пакете processors, имеют зависимость от модели, представленной в классе Graph, так как наполняют её данными, класс же Graph использует функционал библиотеки Spoon для выявления абстрактных классов с помощью фильтра. Данное обстоятельство имеет разные трактовки. С одной стороны, можно считать, что класс AbstractionFilter помещен не в тот пакет, так как его использование не связано с использованием других классов данного пакета. С другой стороны, если разработчик умышленно группирует классы по пакетам с семантической точки зрения (например, в данном случае в пакете processors расположены классы, которые являются расширением абстракций библиотеки Spoon), то это демонстрирует несостоятельность оценивания пакетов как компонентов, что используют некоторые стандартные решения анализаторов (например, JDepend).

Алгоритм, описанный в предыдущем параграфе, выявляет возможное разбиение на 3 компонента, к которым относятся классы-обработчики, основной компонент, содержащий реализацию бизнес-логики программы и компонент, включающий в себя класс Main. Последний компонент обусловлен тем, что бизнес-логика высокого уровня сосредоточена в одном классе. То, что алгоритм не выявил более детального разбиения, показывает его недостаточность по ряду аспектов для проведения подобного анализа, а также результат спорных архитектурных решений, некоторые из которых рассмотрены выше. Итоговую диаграмму зависимостей компонентов можно представить следующим образом:

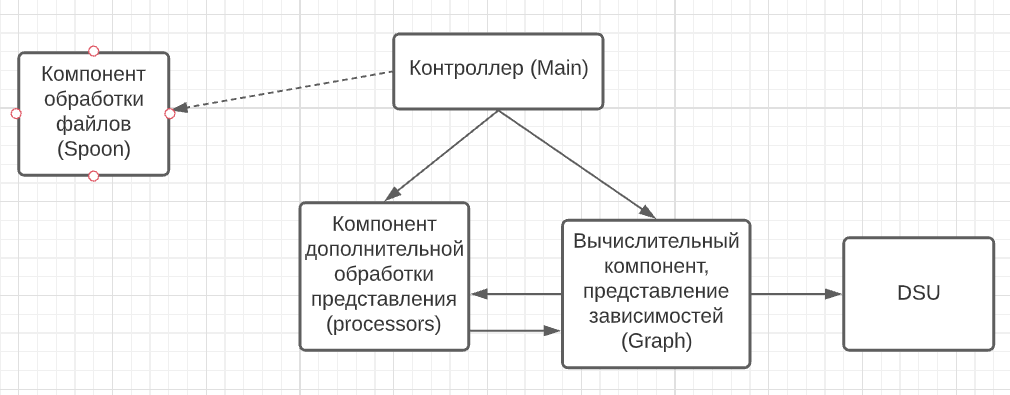


Рисунок 4.1 Итоговая диаграмма компонентов

4.5 Методы изменения кода для улучшения его характеристик

Продолжим рассматривать стандартные проблемы, связанные с направленностью зависимостей, на примере кода разрабатываемого анализатора. Выделим три основных проблемы:

1. Граф имеет строение, далекое от древовидного.
2. Наличие циклической зависимости между компонентами.
3. Низкая абстрактность кода.

**4.5.1 Инверсия зависимостей**

Как уже было сказано ранее, древовидное строение обеспечивает наиболее гибкую структуру для работы с кодом программы. Нарушается структура по причине наличия циклических зависимостей, повторного использования кода класса, а также при использовании инверсии (в случае, когда зависимость между двумя классами разбивается за счет добавления абстракции, оба класса будут иметь зависимость, направленную в сторону абстракции. Безусловно, это не является недостатком архитектуры кода, так как является частью паттерна построения многоуровневой архитектуры, несмотря на то что граф зависимостей не является деревом. Стоит отметить, что в рамках проектируемого анализатора данная проблема избегается за счет того, что он не рассматривает зависимости от абстракций). При этом заметим, что поддержание точной древовидной структуры не представляется возможным даже в рамках компонента: предположим, у нас есть некоторые два класса, которые выполняют независимую друг от друга логику, однако используют в вычислениях модель А. Один из данных классов также имеет зависимость от некоторой другой модели Б, при этом в зависимости от ветви графа исполнения программы логика данных классов вызывается в некотором классе контроллере. Наглядно данная ситуация представлена на рисунке ниже:

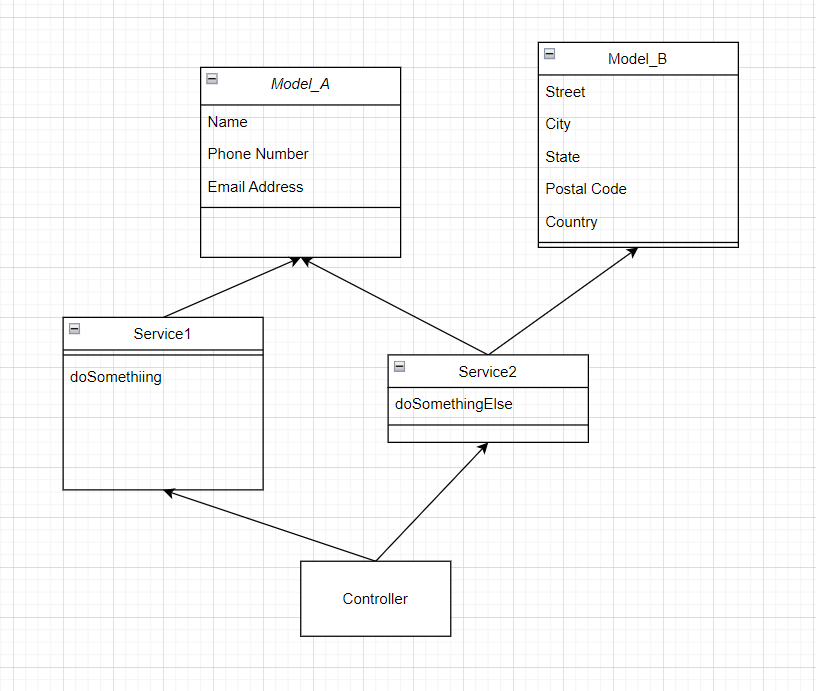


Рисунок 4.2 Ромбовидная зависимость

В данном случае возможны два варианта. Если классы Service1 и Service2 используют разные методы модели A, то в таком случае проблема решается введением двух новых абстракций, обладающих соответствующими методами. Более общее решение сводится к задаче построения непересекающегося множества абстракций, задающих требуемое для отдельных зависимостей поведение. Стоит отметить, что при этом возрастает значение абстрактности кода, что увеличивает расстояние характеристик от оптимальной прямой.

Если же вышеуказанные классы используют одинаковые методы (поля) модели в своих вычислениях, то, вообще говоря, обеспечить древовидность структуры графа возможно только отказавшись от повторного использования кода. Использование инверсии зависимостей для разрыва ромбовидного фрагмента графа лишь завуалирует проблему для анализатора, поскольку создаваемые абстракции все равно будут иметь зависимости, нарушающие древовидную структуру. Однако, данные фрагменты нарушают принцип SRP, так как в случае изменения модели потребуется вносить изменения в необязательно зависящие от одного актора компоненты. В случае, если устойчивость модели высока, дополнительные изменения не требуются, иначе расстояние до оптимальной прямой увеличится. В противном случае разумно переместить модель на более низкий уровень, инвертировав зависимость от нее, чтобы защитить сервисы от изменений.

Для теоретической обоснованности последующих шагов введем точное обозначение уровня компонентов кода. Граф выполнения любой программы содержит по крайней мере по одной точке входа и выхода. Так как данные точки обеспечивают взаимосвязь программы и пользователя, логично расположить их на одном уровне компонентов, который и будем считать наивысшим.

Дальнейшее разделение компонентов по уровням будем производить в соответствии с принципами, описанными в главе 1: компонент наиболее высокого уровня должен задавать поведение программы и не зависеть от более низкоуровневых реализаций запрашиваемого поведения. Как можно видеть из кода самого реализатора, по умолчанию код более высокого уровня вызывает более низкий, приобретая от него зависимостей, соответственно, каждое использование инверсии зависимостей от более низкого уровня четко обозначает этот самый уровень (при условии отсутствия иных зависимостей от данных компонентов на более высоком; в таком случае потребуется инвертировать все зависимости данного уровня). Рассмотрим пример из книги [2]:

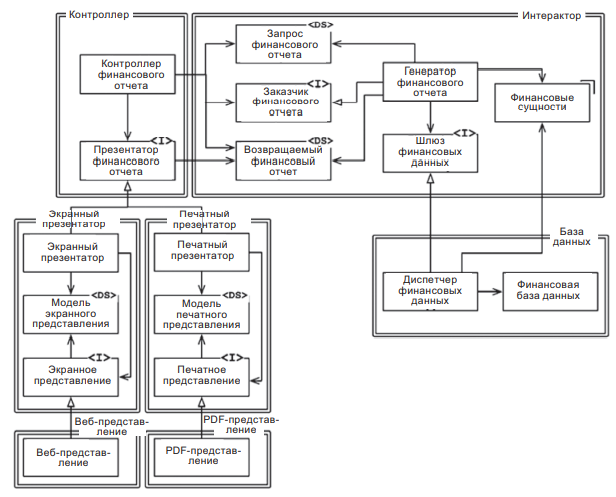


Рисунок 4.3 Пример распределения классов по уровням

Символом <I> на данной схеме обозначены интерфейсы, <DS> — структуры данных. Уровень представлений отделен от уровня презентаторов таким же способом, которым предлагается производить это при оценке проекта анализатором [2].

Методы разрыва циклических зависимостей уже описаны в главе 1, однако возникает вопрос, какой из участков цикла оптимален для разрыва. Стоит отметить, что при грамотном проектировании подобная ситуация в принципе может возникнуть исключительно среди классов, принадлежащих одному уровню компонентов. В таком случае место разрыва зависимости должно совпадать с местом, наиболее сильно связанным с уровнем, расположенным на один выше, чтобы как можно надежнее исключить нарушение SRP.

Проблема низкой абстрактности кода решится в ходе решения первых двух за счет добавления новых интерфейсов и дробления существующих.

**4.5.2 Использование остовных деревьев**

В рамках одного уровня допускаются произвольные структуры графа зависимостей компонентов. Для их оптимизации, при условии, что они не попадают под типичный случай из предыдущего пункта, можно попытаться перестроить код в соответствии с древовидной структурой за счет перемещения методов между классами либо просто предоставить возможную реализацию пользователю для внесения дальнейших изменений в программу.

Для построения остовного дерева воспользуемся классическим алгоритмом Прима, после чего получим список зависимостей, которые необходимы при данном наборе классов, а также зависимости, которые, вообще говоря, избыточны. Данные избыточные зависимости нельзя в общем случае назвать изъяном архитектуры, так как они могут быть вызваны стремлением сделать код понятнее за счет распределения по компонентам либо повторно использовать существующее поведение. По этой же причине вносить изменения в существующую структуру графа зависимостей в рамках одного компонента по каким-либо причинам может быть опасно в связи с нарушением логики работы программы. Рассмотрим возможный вариант работы алгоритма:

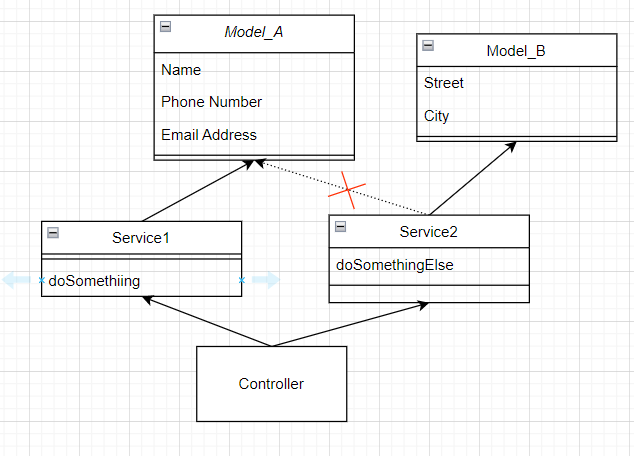


Рисунок 4.4 Возможный результат работы алгоритма Прима

Несложно заметить, что любые нарушения древовидной структуры графа зависимостей (при условии отсутствия циклических) можно представить в виде одного из рассмотренных в предыдущем пункте случаев ромбовидной структуры: в связи с использованием одинаковых либо различных методов одного класса из нескольких других. Таким образом, результатом работы алгоритма станет перечень классов, которые нуждаются в преобразованиях, описанных в предыдущем пункте.

Сложность заключается в том, что для начала поиска необходимо понимать, в рамках какого участка подграфа осуществляется построение, а также необходимо учитывать, что в данном графе могут быть несколько входных точек, которые обусловлены реализациями запрашиваемой логики различными компонентами более высокого уровня. Таким образом, следует определить входные точки уровня, после чего запустить алгоритм прима из каждой из них, чтобы гарантировать минимализм построенного дерева на всем объеме подграфа. При этом возникнут ситуации, в которых различные входные точки используют одни и те же классы, однако даже расстояние до них в графе может отличаться (за единицу расстояния возьмем ребро зависимости).

Учитывая, что сложность прямого решения задачи явно выходит за полиномиальную, данная задача представляет собой классический случай, требующий динамического программирования. При этом нет необходимости сохранять полную структуру анализируемого для вершины графа, достаточно пометить вершину как посещенную, добавив в список входных ту, которая инициировала посещение. Модель же дерева будет строиться параллельно в рамках алгоритма Прима, при этом не вошедшие в него зависимости будут более важны, так как именно они будут предоставлены пользователю.

4.5.3 Разбиение компонентов по уровням

В рамках пунктов 4.5.1 и 4.5.2 было выявлено, что для каждого уровня программы для успешной работы вышеописанных алгоритмов необходимо определить его границы и входные точки.

Рассмотрим управляющий граф программы. В зависимости от типа приложения количество входных точек данного графа меняется (в случае с консольным приложением это одна точка — запуск программы, в случае с веб-приложением — каждая точка входа контроллеров API).

В рамках следования по управляющему графу от точки входа программа вызывает различные методы других классов, при этом, как уже было сказано выше, бизнес-логика не должна зависеть от низкоуровневых реализаций. В идеальном случае (либо после первой итерации алгоритма увеличения абстрактности приложения) зависимости бизнес-логики представляют собой зависимости от абстракций либо классов с высокой степенью устойчивости, при этом в обоих случаях классы-зависимости (интерфейсы) принадлежат уровню бизнес-логики.

При этом стоит отметить, что в случае наличия от конкретных реализаций даже с высокой степенью устойчивости зависимостей с более низкого уровня, они также должны быть отделены абстракцией во избежание некорректной работы алгоритмов. В таком случае границей уровня является зависимость, представляющая собой наследование абстракции либо реализацию конкретного интерфейса, при этом стоит отметить, что в случае абстрактных классов зависимость от используемых в стандартных реализациях методов конкретных классов сохраняется.

Так как данные классы могут располагаться в разных пакетах и обрабатываться в разной последовательности, алгоритм построения очередной системы непересекающихся множеств имеет место выполнять уже после построения модели графа либо на его этапе за счет формирования системы по двум параметрам — распределения классов по компонентам и реализации абстракций. Данный метод построения позволит оптимизировать работу анализатора и определить наивысший уровень кода в рамках входной точки, так как можно отметить, что каждый уровень программы можно представить в виде отдельного компонента (либо их множества, в случае если зависимости кода, соответствующие разным входным точкам программы, не пересекаются), таким образом, алгоритм построения компонентов, уже существующий в анализаторе, может быть дополнен до алгоритма определения уровня конкретного класса в рамках программы.

За счет использования уровней кода возникает возможность точно определить возможное расположение логики программы при ее доработке, при этом данная логика будет гарантировать масштабируемость программы и сохранение архитектурных метрик ее кода.

Основной проблемой при реализации алгоритма является выявление точек входа/выхода программы. Рассмотрим два различных подхода:

* Использование пользовательских параметров;
* Использование средств анализатора.

В первом случае пользователь анализатора напрямую указывает точки входа своей программы. Для это понадобится доработать анализатор по ряду направлений, а именно внести возможность указать главный метод программы в случае, если она представляет собой консольное приложение, либо контроллер в случае веб, при этом потребуется дополнительно проанализировать код контроллера для выявления точек доступа к API.

При использовании средств анализатора в качестве маркеров входных точек программы в случае консольного приложения может выступать публичный статический метод с определенной семантикой (например, main), в случае использования API — методы, аннотируемые соответствующим фреймворку маркером входной точки (например, Path над методом в случае Bootique либо аннотация @Controller над классом в случае Spring Boot). Минус данного подхода заключается в том, что в обоих случаях наличие маркера не дает гарантий — публичные статические методы могут быть использованы в любой части программы, семантические характеристики в принципе не могут служить доказательством так как зависят от программиста, контроллеры же MVC-архитектуры могут не использоваться либо в целом быть оторванными от основной логики программы.

Стоит отметить, что сам анализатор может представлять собой также приложения разного типа — веб-приложение, консольное приложение либо обычный проект, построенный с целью протестировать разработанные алгоритмы на примере. В различных реализациях анализатора также отличается подход, наиболее удобный для использования при выявлении точек входа/выхода анализируемого приложения — в случае консольного приложения это дополнительный параметр при старте, в случае веб-приложения — параметр запроса либо алгоритм, в случае же тестового проекта данный момент не играет роли так как можно рассмотреть конкретный используемый в коде пример проекта.

4.5.4 Анализ вероятности изменения зависимого компонента

В главе 2 рассматривался метод оценки вероятности того, что в модуле потребуются изменения при изменении некоторого другого модуля. Данная характеристика 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 во избежание непреднамеренных обязательных релизов), дополнительно оценить независимость участков кода друг от друга, что позволит характеризовать в том числе масштабируемость программы.

Для использования в расчетах рассмотренной в параграфе 2.1 формулы 2.8 требуется построить отображение мер сил связности и сцепления, определенных для модулей, на меры, которые будут использоваться при оценке классов. В первую очередь, в качестве части компонента будет рассматриваться метод вместо класса либо их группы. Построим отображение определения силы связности модулей на силу связности классов:

1. СС = 3, временная связность. Данное значение будет использоваться по умолчанию, так как Java не позволяет загружать в память некоторые методы определенного класса, что означает, что все методы класса гарантированно будут использоваться в один момент времени.
2. СС = 5, процедурная связность. Наличие сценария в поведении класса подразумевает наличие некоторого метода более высокого уровня, который последовательно вызывает все внешние методы класса. Стоит отметить, что анализировать во избежание увеличения сложности алгоритма стоит только на единичную глубину, так как если метод, транзитивно последовательно вызывающий все методы класса, расположен на более высоком уровне, вероятность необходимости внесения изменений будет учтена.
3. СС = 7, коммуникативная связность. В данном контексте будем считать связность коммуникативной, если методы в качестве параметра получают параметры одинакового сложного типа данных.
4. СС = 9, информационная связность. Пожалуй, самый сложный в оценивании показатель. Наиболее логично рассматривать его как наличие некоторой переменной у вызывающего методы данного класса метода более высокого уровня, которое может быть выявлено косвенно по наличию среди вызовов методов, которые возвращают и принимают в качестве параметра одинаковые типы данных соответственно (рассматривать ситуацию некорректного использования ссылочного типа данных, при которой он изменяется при передаче его в метод в качестве параметра, избыточно и трудно осуществимо).
5. СС = 10, функциональная связность. При вызове внешнего метода класса в ходе его выполнения вызываются все остальные.

В ходе отображения числовых характеристик не был учтен момент, согласно которому лишь часть методов класса могут удовлетворять данным характеристикам, аналогично тому, как в оценках модулей не учитывается соотношение подходящих частей модуля. Подразумевается, что наличие хотя бы одного метода, удовлетворяющего требованиям, повышает значение силы связности. В ходе реализации можно добиться более тонкого анализа за счет балансировки влияния отношения соответствующих силе связности методов к общему числу методов класса за счет использования весов, хотя это и противоречит базовой формуле 2.8.

Построим отображение для характеристик сцепления:

1. СЦ = 0, полная независимость. Мера по умолчанию.
2. СЦ = 1, сцепление по данным. Классы вызывают методы друг друга, при этом в качестве параметров и возвращаемых значений используются примитивные типы данных. В качестве примитивных типов данных в данном контексте разумно учитывать любые стандартные средства языка программирования.
3. СЦ = 3, сцепление по образцу. Классы совместно используют некоторые статические методы либо переменные других классов.
4. СЦ = 5, сцепление по общей области. Классы также разделяют статические переменные, однако в процессе работы с ними могут изменять значение.
5. СЦ = 7, сцепление по управлению. Один класс передает другому некоторый флаг, который используется в определении перехода графа управления программы.
6. СЦ = 9, сцепление по внешним ссылкам. Класс вызывает метод другого класса, возвращающий некоторую сложную структуру данных, которая хранится в нем в качестве поля.
7. СЦ = 10, сцепление по содержанию. Класс может в том числе изменять значение полей другого класса.

Возникает закономерный вопрос, как трактовать вызов одним классом метода другого класса, который возвращает некоторую сложную структуру данных, в нем не хранящуюся. Повлечь изменение в зависимом классе в таком случае может изменение возвращаемого типа данных, либо логики его обработки. Аналогичная ситуация происходит с любыми хранящимися данными в классе, поэтому значение СЦ = 9 становится наиболее часто используемым при прямой зависимости.

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.

Предположим, в рамках проектирования рассматривается возможность вынесения всех общих глобальных (статических) структур данных в отдельный компонент. За счет величины СЦ = 3 против девяти в случае вызова метода с передачей сложного типа данных вероятность необходимости совместного изменения компонентов будет значительно ниже, что позволит отсеять ряд слабых по данному показателю зависимостей (предположим, вероятность в районе 30%. Тогда при работе алгоритма достаточно задать пороговое значение вероятности в районе 40%, после чего обрабатывать с помощью системы непересекающихся множеств только связи между компонентами, превышающими данный порог).

По формуле 2.8 для каждой пары компонентов требуется рассчитать значения связности для каждого класса и сцепления для каждой рассматриваемой пары. Заметим, что при оценке связности удобно использовать уже имеющийся механизм определения зависимости, выражающейся в вызове метода класса. Так, связность уровня СС = 5 означает, что некоторый класс в рамках своих методов вызывает все публичные метода класса-зависимости. В свою очередь, СС = 7 означает, что вызываемый метод имеет те же параметры, что и некоторый другой метод, который уже вызывался у данного класса (отметим, что полученный подобным анализом характеристики не являются в полной мере связностью данного класса по ее определению; так, например, класс может иметь некоторый метод, который не используется классами в рамках анализируемого кода, либо не используется вовсе. Они никак не влияют на другие компоненты, поэтому могут быть исключены. Также несколько сужено описанное выше отображение за счет рассмотрения исключительно публичных методов по тем же причинам). Значение СС = 9 зависит от возвращаемого при вызове метода значение, СС = 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 также могут быть выявлены на этапе добавления зависимостей по методу, однако для других значений разумно использовать дополнительную обработку на этапе построенной модели. Так, 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.

Сравнение алгоритма с описанным выше приведено в следующей таблице:

Таблица 4.1 Сравнение алгоритмов разбиения на компоненты

|  |  |  |
| --- | --- | --- |
| Критерий | Алгоритм, основанный на зависимостях | Алгоритм, основанный на сцеплении |
| Момент выполнения | Построение графа на этапе разбора, обработка дополнительным этапом | Часть характеристик считается на этапе разбора, часть дополнительным этапом. Граф также строится отдельно |
| Сложность | O(n2\*logn) без учета этапа разбора, где n — число компонентов | O(n2\*m), где n — число компонентов, m — число методов класса |
| Хранимые данные | Зависимости в формате имя класса — имя класса-зависимости | Перечень статических методов для каждого класса, значения связности для каждого класса, значения сцепления для каждой зависимости, имена методов, задающих зависимость (хранятся для алгоритма разбиения на интерфейсы) |
| Управление | Использует показатель стабильности для отделения элементов наивысшего уровня | Использует порог вероятности изменения в зависимом компоненте при изменениях в данном |
| Применение | Построение точной модели возможных компонент, которые имеют между собой исключительно зависимости от абстракций и могут быть без проблем разбиты на совместно использующиеся группы классов | Построение вариативного числа компонент, которое основано на связности класса и его сцеплении с другими классами. Между компонентами могут быть зависимости, с высокой долей вероятности это будут вызовы методов без использования сложных типов либо совместные обращения к статическим методам и полям некоторого класса |

Таким образом, анализатор, разработанный на данном этапе, имеет ряд архитектурных недостатков и может быть улучшен за счет использования некоторых алгоритмов на графах. В первую очередь это касается выявления недостатков определенного типа — зависимости высокоуровневой логики от низкоуровневой, бессмысленного нарушения древовидной структуры, низкой абстрактность кода. Кроме того, в рамках используемой библиотеки, возможно внести изменения в обрабатываемый проект, однако для этого требуется установить места, вмешательство в которых принесет наибольшую пользу с точки зрения архитектурных метрик приложения, что повлечет за собой облечение поддержки и доработки кода.

4.6 Реализация алгоритмов анализа строения компонентов

Целью представленных ниже алгоритмов является дополнение функционала анализатора за счет предоставления пользователю списка неточностей, расширенного возможными разбиениями используемых классов по интерфейсам, а также распределением компонент приложения по уровням, базирующимся на использовании инверсии зависимостей для абстрагирования более высокого уровня. Первый позволит пользователю избежать ромбовидных зависимостей за счет замены зависимостей на новые абстракции, второй предназначен для облегчения внедрения новых компонентов.

Разбиение класса по интерфейсам базируется на системе непересекающихся множеств, хотя и представляет собой нестандартную для данной структуры данных задачу — вместо объединения множеств в случае удовлетворения некоторому условию требуется разбивать уже существующее, однако при этом можно выделить ряд свойств, характерных только для нее — наличие системы множеств, которые не могут иметь общих элементов, необходимость поиска множества для некоторого элемента, быстрого создания нового множества на основе элементов. Итоговый алгоритм выглядит следующим образом:

* В рамках формирования модели графа зависимостей сформировать входные, представляющие собой вызов некоторого метода класса-зависимости. Данные методы аккумулировать как одну зависимость (так как зависимый класс у них совпадает). Обрабатывать данные зависимости группами, представляющими собой все вызовы методов определенного класса из зависящих от него, использовать некоторый квалификатор для предоставления пользователю информации, какой класс должен использовать предлагаемый интерфейс.
* Сформировать из первой подобной зависимости множество в системе, поместить в него все вызываемые методы.
* Поместить все не найденные в системе элементы нового набора методов в новое множество.
* Дополнить номера требуемых данному классу множеств методов номерами, соответствующим уже существующем в системе множествам, хранящим требуемые ему элементы.
* В случае, если класс использует лишь некоторые методы из множества, поместить все неиспользованные методы в отдельное множество, добавив его индекс в требуемые ко всем классам, использующим множество методов до преобразований. Модифицировать множества в системе, если требуется (изменить размеры и элементы-корни).

Рассмотрим пример. Представленная ниже схема представляет собой стандартную ромбовидную зависимость (около стрелок указаны методы, вызываемые зависимым классом):

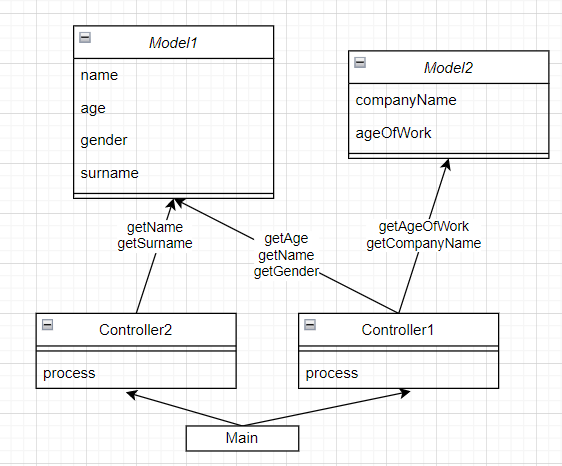


Рисунок 4.5 Модели до дробления

Результат работы анализатора на данном участке кода:

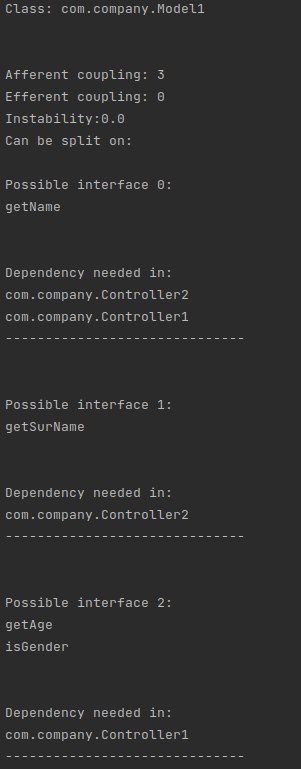


Рисунок 4.6 Разбиение по интерфейсам

Как видно из рисунка 4.6, анализатор предлагает наилучшее независимое (т. е., без расширения существующих) разбиение модели по интерфейсам, а также указывает, какой класс может заменить зависимость от модели.

Разбиение по уровням осуществляется по описанному выше алгоритму. Для сбора информации о зависимостях-абстракциях расширим модель за счет набора абстрактных классов и интерфейсов, поставленных в соответствие их реализациям. Для имитации технологий внедрения зависимостей добавим в модели поля без их инициализации, потому что иначе основной класс будет иметь зависимость от всех используемых моделей и нарушать структуру уровней в анализаторе (при использовании Spring либо Bootique такой проблемы не возникает). Результат работы анализатора:

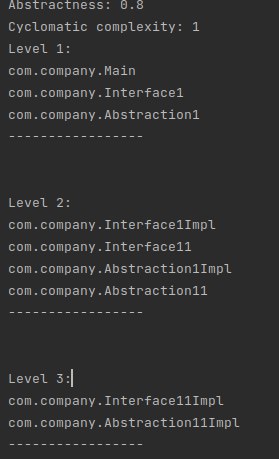


Рисунок 4.7 Разбиение компонентов по уровням

Подобная информация позволяет определить уровень абстракции, на котором должен работать код, выполняющий некоторые функции, а также найти непреднамеренные пересечения уровней, которые будут выражены в отношении большого количества классов к некоторому уровню. Это означает, что детали либо зависят от ненужных конкретных реализации высокоуровневой бизнес-логики, либо высокоуровневая бизнес-логика зависит от низкоуровневого кода. Также возможно, что несколько уровней зависят от некоторых сквозных участков логики что также, вообще говоря, некорректно.

Таким образом, разработанный анализатор позволяет пользователю получить объемный набор данных о коде, который представляет собой не только численную информацию, позволяющую оценить код по некоторым общим критериям, но и конкретные недостатки, предоставляя пользователю выбор относительно дальнейшего изменения кода. Тем не менее, можно выделить ряд граничных ситуаций, в которых анализатор может работать неэффективно — наличие сквозных участков логики, использование наследования от реализации (вообще говоря, нежелательная практика), игнорирование существующей сети интерфейсов и так далее. Каждый из описанных случаев накладывает ограничения на работу анализатора, однако даже при этом он способен предоставить основной функционал, свойственный всем аналогам — расчет характеристик кода и нахождение циклических зависимостей.

В будущих исследованиях работу анализатора можно улучшить, усовершенствовав алгоритм распределения по компонентам за счет использования различных метрик связности и анализа уровней логики более сложных приложений, а также возможно расширить его функционал за счет предоставления рекомендаций по улучшению архитектуры и внесения изменений в проект самим анализатором.

**ВЫВОДЫ**

1. Описана методика построения модели кода, используемая для этого библиотека, специфика анализатора, связанная с ее применением.
2. Реализован подсчет основных метрик компонентов кода.
3. Спроектирован и реализован алгоритм распределения классов по компонентам.
4. Спроектирован и реализован алгоритм распределения классов по уровням, механизм поиска входных и выходных точек программы.
5. Проанализирована эффективность анализатора в случае использования современных средств внедрения зависимости.
6. Разработан и внедрен алгоритм выявления циклических зависимостей на уровнях пакетов и классов.
7. Рассмотрены типичные проблемы проанализированного кода, спроектирован алгоритм разбиения ромбовидных зависимостей с использованием остовных деревьев.
8. Спроектирован и реализован алгоритм поиска вариантов разбиения ромбовидных зависимостей за счет разбиения моделей на интерфейсы.
9. Построено отображение характеристик связности и сцепления модулей на связность и сцепление классов.
10. Спроектирован и реализован алгоритм подсчета вероятности изменения в компоненте при изменениях в другом компоненте, разработан алгоритм разбиения классов по компонентам на основе этой метрики с заданием порога, проведен сравнительный анализ рассмотренных алгоритмов разбиения по компонентам.

ЗАКЛЮЧЕНИЕ

Основной задачей разработки программного обеспечения является построение системы, которое удовлетворяет всем предъявленным к ней требованиям по функциональности и используемым ресурсам. Не существует универсальных рекомендаций для построения оптимальной архитектуры. Задачей архитектора является лавирование между различными принципами построения архитектуры, которое позволяет добиться требуемых характеристик. Данные характеристики могут относиться как к затраченным на систему ресурсам, так и к предоставляемой системой функциональности, простоте разработки, свойствам системы и т. д.

Для оценивания архитектуры программного обеспечения разработано множество наборов метрик, которые позволяют придать собственным оценкам архитектора по отношению к системе математический характер. Данные характеристики позволяют оценить соответствие системы некоторым обобщенным принципам её построения, найти дефекты в логике и предотвратить рост затрат на её сопровождение. Наиболее часто из них используются связность, позволяющая оценить грамотность наполнения класса, сцепление, абстрактность, цикломатическая сложность и устойчивость. Основная проблема использования стандартных анализаторов — возможное несоответствие подхода архитектора к созданию системы со значениями, которые имеют показатели рассмотренных выше метрик в данных анализаторах.

Анализатор, разработанный в ходе данной работы на данном этапе, включает в себя стандартный функционал для компонентов: подсчет 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 различными методами.

Также анализатор включает в себя функционал, которым не обладают рассмотренные в сравнительном анализе решения: поиск возможного разбиения модели по интерфейсам, представление многоуровневой схемы приложения. Данные алгоритмы позволяют избежать бесполезных отклонений графа зависимостей приложения от древовидного, исключить ромбовидные зависимости, повысить абстрактность кода при необходимости и определить уровень дальнейшего расширения функционала приложения.

За счет способностей библиотеки Spoon к изменению анализируемого кода, анализатор можно расширить алгоритмами изменения кода на основе предоставляемых данных. Разработанная в ходе данной работы версия собирает исчерпывающие сведения без вмешательства в структуру кода, предоставляя пользователю возможность самостоятельно сделать выводы на основании полученных значений и предложенных возможностей. За счет этого достигается гибкость, а также соответствие разрабатываемого кода с ожиданиями архитектора.

В качестве дальнейших направлений научной деятельности по данному направлению с использованием данного анализатора можно представить алгоритм поиска наиболее связанного с более высоким уровнем компонента, распределение сквозных элементов логики по уровням за счет добавления мостов между ними, разработку модуля, занимающегося изменением кода и базирующемся на семантических его метриках (так, например, можно генерировать имена интерфейсам на основании названий их методов), внедрение систем сборки таких как maven, gradle и анализ зависимостей между компонентами уровня модулей, внедрение наследования абстракций в механизм разбиения моделей, оптимизацию анализатора для граничных условий, описанных выше. Также предоставляемые метрики могут использоваться для обучения нейросетей, например, с целью задать наиболее близкое разбиение моделей по абстракциям к оптимальной прямой, основываясь на характеристиках устойчивости модели и абстрактности кода.

Результаты работы были представлены в научной статье «Разработка анализатора архитектуры java-приложений» в журнале «НАУЧНЫЕ ИССЛЕДОВАНИЯ XXI ВЕКА» в выпуске №1 от 2022 года, а также на 79-й конференции БГУ.

список литературы

1. 4 типа архитектуры программного обеспечения [Электронный ресурс]. — Режим доступа: https://medium.com/nuances-of-programming/4-типа-архитектуры-программного-обеспечения. Дата доступа: 15.10.2021.
2. Мартин, Р. Чистая архитектура. Искусство разработки программного обеспечения / Р. Мартин — СПб.: Питер, 2018. — С. 80-340.
3. Назаров, С. В. Архитектура и проектирование программных систем / С.В. Назаров — М.: ИНФРА-М, 2013. — Гл. 2,4,6. — С. 115-320.
4. Технология разработки программного обеспечения: конспект лекции / сост. И. И. Савенко; Томский политехнический университет. — Томск: Изд-во Томского политехнического университета, 2014
5. A Comparative Analysis of Software Architecture Evaluation Methods / Izadkhah, D., Hooshyar M // Journal of Software — 2016 — vol. 11, №9
6. Class Cohesion Metrics for Software Engineering: A Critical Review / Athar A., Liaquat R., Farooque A. // Computer Science Journal of Moldova — 2017 — vol. 25, №1
7. Software Architecture Analysis [Электронный ресурс]. — Режим доступа: https://medium.com/quintoandar-tech-blog/software-architecture-analysis-25bd0274d837. Дата доступа: 05.11.2021.
8. Spoon — Source Code Analysis and Transformation for Java [Электронный ресурс]. — Режим доступа: https://spoon.gforge.inria.fr/. Дата доступа: 14.11.2021.
9. 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.
10. C. Kaner, S. Member, IEEE, W. P. Bond Software Engineering Metrics: What do They Measure and How Do We Know? / C. Kaner, S. Member // 10TH INTERNATIONAL SOFTWARE METRICS SYMPOSIUM, METRICS 2004
11. IEEE Standard for a Software Quality Metrics Methodology: IEEE Std 1061-1998. — Введ. 30.11.1994 — IEEE Standards Dept., 1998
12. Мартин Р. Чистый код. Создание, анализ и рефакторинг. — СПб.: Питер, 2019. — Гл. 6,8, 12-17
13. Bootique: Minimally Opinionated Framework for Runnable Java [Электронный ресурс]. — Режим доступа: https://bootique.io/docs/. Дата доступа: 13.04.2022.
14. Inversion of Control and Dependency Injection with Spring | Baeldung [Электронный ресурс]. — Режим доступа: https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring. Дата доступа: 13.04.2022.

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ А

Результат работы анализатора

Class: com.company.Interface11

Afferent coupling: 3

Efferent coupling: 0

Coherence: 3

Instability: 0.0

Can be split on:

Possible interface 0:

process

Dependency needed in:

com.company.Abstraction1Impl

------------------------------

------------------------------------------------------------------------

Class: com.company.Interface1Impl

Afferent coupling: 0

Efferent coupling: 3

Coherence: 10

Instability: 1.0

Can be split on:

------------------------------------------------------------------------

Class: com.company.Interface11Impl

Afferent coupling: 0

Efferent coupling: 1

Coherence: 3

Instability: 1.0

Can be split on:

------------------------------------------------------------------------

Class: com.company.Abstraction1Impl

Afferent coupling: 0

Efferent coupling: 4

Coherence: 5

Instability: 1.0

Can be split on:

------------------------------------------------------------------------

Class: com.company.Abstraction11Impl

Afferent coupling: 0

Efferent coupling: 1

Coherence: 3

Instability: 1.0

Can be split on:

------------------------------------------------------------------------

Class: com.company.Main

Afferent coupling: 0

Efferent coupling: 2

Coherence: 3

Instability: 1.0

Can be split on:

------------------------------------------------------------------------

Class: com.company.Interface1

Afferent coupling: 2

Efferent coupling: 0

Coherence: 3

Instability: 0.0

Can be split on:

------------------------------------------------------------------------

Class: com.company.Abstraction11

Afferent coupling: 3

Efferent coupling: 0

Coherence: 3

Instability: 0.0

Can be split on:

Possible interface 0:

compute

Dependency needed in:

com.company.Interface1Impl

------------------------------

------------------------------------------------------------------------

Class: com.company.Abstraction1

Afferent coupling: 3

Efferent coupling: 0

Coherence: 3

Instability: 0.0

Can be split on:

Possible interface 0:

getName

Dependency needed in:

com.company.Abstraction1Impl

------------------------------

------------------------------------------------------------------------

found Component:

com.company.Abstraction1Impl

com.company.Abstraction1

------------------------------

found Component:

com.company.Abstraction11Impl

com.company.Abstraction11

------------------------------

found Component:

com.company.Interface11Impl

com.company.Interface11

------------------------------

found Component:

com.company.Interface1Impl

com.company.Interface1

------------------------------

found Component:

com.company.Main

------------------------------

com.company.Interface1Impl com.company.Interface1 0.0

com.company.Interface11Impl com.company.Abstraction1 0.0

com.company.Interface1Impl com.company.Abstraction1 0.0

com.company.Interface11Impl com.company.Interface1 0.0

com.company.Abstraction1Impl com.company.Abstraction11Impl 0.0

com.company.Interface11 com.company.Abstraction11Impl 0.0

com.company.Abstraction11Impl com.company.Main 0.0

com.company.Abstraction11Impl com.company.Abstraction11 0.0

com.company.Interface1Impl com.company.Interface11Impl 0.0

com.company.Interface11 com.company.Abstraction1Impl 0.19

com.company.Interface1 com.company.Abstraction1 0.0

com.company.Interface11 com.company.Abstraction1 0.0

com.company.Abstraction11 com.company.Abstraction1 0.0

com.company.Abstraction1Impl com.company.Interface1 0.0

com.company.Main com.company.Abstraction11 0.0

com.company.Interface11 com.company.Interface1Impl 0.0

com.company.Abstraction1Impl com.company.Abstraction11 0.0

com.company.Abstraction11Impl com.company.Abstraction1 0.0

com.company.Abstraction11Impl com.company.Interface1 0.0

com.company.Interface1Impl com.company.Main 0.0

com.company.Abstraction1Impl com.company.Main 0.0

com.company.Interface11Impl com.company.Main 0.0

com.company.Interface11Impl com.company.Abstraction11 0.0

com.company.Interface1Impl com.company.Abstraction1Impl 0.0

com.company.Main com.company.Interface1 0.0

com.company.Interface1 com.company.Abstraction11 0.0

com.company.Interface11 com.company.Main 0.0

com.company.Interface1Impl com.company.Abstraction11Impl 0.0

com.company.Abstraction1Impl com.company.Abstraction1 0.19

com.company.Interface11 com.company.Interface11Impl 0.0

com.company.Interface11 com.company.Interface1 0.0

com.company.Interface11 com.company.Abstraction11 0.0

com.company.Interface11Impl com.company.Abstraction11Impl 0.0

com.company.Main com.company.Abstraction1 0.0

com.company.Interface1Impl com.company.Abstraction11 0.265

com.company.Interface11Impl com.company.Abstraction1Impl 0.0

Abstractness: 0.4444444444444444

Cyclomatic complexity: 1

Level 1:

com.company.Main

com.company.Interface1

com.company.Abstraction1

-----------------

Level 2:

com.company.Interface1Impl

com.company.Interface11

com.company.Abstraction1Impl

com.company.Abstraction11

-----------------

Level 3:

com.company.Interface11Impl

com.company.Abstraction11Impl

-----------------