«Московский государственный технический университет имени Н.Э. Баумана»
(МГТУ им. Н.Э. Баумана)

Факультет: Информатика и системы управления

Кафедра: ИУ7

Проектирование программного обеспечения

Студент группы ИУ7-63,

Степанов Александр Олегович

Преподаватель:

Бекасов Денис Евгеньевич

2020 г.

Оглавление

1 | Введение

1.1 Цель и задачи

1.1.1 Цель

Ответить для себя на вопрос «Что такое хорошая Архитектура?»

1.1.2 Задачи

— Критерии хорошей Архитектуры

— Проектирование программных компонентов (SOLID)

— Проектирование программных систем

— Жизненный цикл разработки ПО

— Организация процесса разработки

— Организация процесса развертывания и сопровождения

1.2 Архитектура

— Сфера применения

— Модификация

— Поддержка

1.2.1 Архитектура программного обеспечения

Архитектура программного обеспечения – совокупность важнейших решений об организации программной системы.

— выбор структурных элементов и их интерфейсов, с помощью которых составлена система, а также их поведения в рамках сотрудничества структурных элементов

— соединение выбранных элементов структуры и поведения во всё более крупные системы

— общий архитектурный стиль

1.2.2 Гради Буч (создатель UML)

Архитектура отражает важные проектные решения по формированию системы, где важность определяется стоимостью изменений

1.2.3 Джойзеф Йодер

Если вы думаете, что хорошая архитектура стоит дорого, попробуйте плохую архитектуру

1.2.4 Ральф Джонсон

Архитектура – это набор верных решений, которые хотелось бы принять на ранних этапах работы над проектом, но которые не более вероятны, чем другие

1.2.5 Том Гилб

Архитектура – это гипотеза, которую требуется доказать реализацией и оценкой

1.2.6 Роберт Мартин

Поспешай не торопясь

Поспешность – самонадеянность, управляющая перепроектированием, приведет к тому же беспорядку что и прежде

1.2.7 Программные архитектуры

— Неизменны на всем протяжении развития IT, начиная с 50-60х гг XX века

— Низкоуровневые детали и высокоуровневая структура являются частями одного целого

— Цель – уменьшение трудозатрат на создание и сопровождение системы

Идея ПО – простая возможность менять поведение компьютера

1.2.8 Битва за архитектуру

— Разработчики – архитектура

— Менеджмент – функциональность

1.3 Парадигмы программирования

— Структурное программирование

— Объектно-ориентированное программирование

— Функциональное программирование

1.3.1 Структурное программирование

1968, принципы Дейкстры

Структурное программирование накладывает ограничение на прямую передачу управления

1. Декомпозиция на подпрограммы
2. Одна точка входа и выхода во всех конструкциях
3. Проектирование сверху вниз
4. Запрет безусловного перехода
5. Любой алгоритм кодируется тремя структурами: последовательность, ветвление и цикл
6. Деление на блоки
7. Базовые конструкции могут быть вложены друг в друга

1.3.2 Объектно-ориентированное программирование

Объектно-ориентированное программирование накладывает ограничение на косвенную передачу управления

— Инкапсуляция

— Наследование

— Полиморфизм

1.3.3 Функциональное программирование

Функциональное программирование накладывает ограничение на присваивание. Если мы можем сделать элемент неизменяемым, то должны сделать его таковым

2 | Проектирование программных компонентов

2.1 Программные структуры среднего уровня. Принципы SOLID

2.1.1 Уровни проектирования

— Уровень функций и методов

— Уровень классов

— Уровень организации компонентов

— Архитектурный уровень

2.1.2 Программный компонент

Программный компонент – единица развертывания (`.jar, .gem, .dll`)

Программный компонент – программная часть системы компонент программного обеспечения

— Независимое развертывание

— Независимая разработка

2.1.3 Программный компонент

Из чего состоит?

Состоит из хорошо спроектированных программных структур среднего уровня

Как создать хорошую программную структуру среднего уровня?

2.1.4 SOLID

— **SRP**: Single Responsibility Principle (Принцип единственной ответственности)

— **OCP**: Open-Closed Principle (Принцип открытости/закрытости)

— **LSP**: Liskov Substitution Principle (Принцип подстановки Барбары Лисков)

— **ISP**: Interface Sgregation Principle (Принцип разделения интерфейсов)

— **DIP**: Dependency Inversion Principle (Принцип инверсии зависимостей)

SRP. Принцип единственной ответственности

— Модуль должен отвечать за одного и только одного актора (внешняя роль)

— Модуль должен иметь одну и только одну причину для изменения

Несоблюдение:

— Проблема модификации общих частей

— Проблема слияния изменений

OCP. Принцип открытости/закрытости

— Программные сущности должны быть открыты для расширения и закрыты для изменения

— Цель: легкая расширяемость и безопасность от влияния изменений

Упорядочивание в иерархию, защищающую компоненты уровнем выше от изменения в компонентах уровня ниже

Чем выше политики – тем выше защита

LSP. Принцип подстановки Барбары Лисков

— Если для каждого объекта o1 типа S существует такой объект o2 типа T, что для всех программ P, определенных в терминах T, поведение P не изменяется при подстановке o1 вместо o2, то S является подтипом T.

— Простое нарушение совместимости может вызвать **загрязнение** архитектуры системы **значительным количеством дополнительных механизмов**

Проблема квадрат-прямоугольник (квадрат не может поддерживать логику прямоугольника)

ISP. Принцип разделения интерфейсов

— Зависимости, несущие лишний груз ненужных и неиспользуемых особенностей, могут стать причиной неожиданных проблем.

DIP. Принцип инверсии зависимости

Для максимальной гибкости:

Зависимости должны быть направлены на абстракции, а не конкретные реализации

2.1.5 Инверсия управления

IoC – архитектурное решение интеграции, упрощающее расширение возможностей системы, при котором поток управления программы контролируется фреймворком.

Логика:

— логика взаимодействия программы разбросана

— поток управления задан неявно

2.1.6 Dependency Injection

— Внедрение зависимости – процесс предоставления внешней зависимости программному компоненту

— В соответствии с SRP объект отдает заботу о построении требуемых ему зависимостей внешнему общему механизму

2.1.7 Лабораторная работа 1

1. Use-Case – диаграмма курсового
2. ER-диаграмма сущностей (не путать с БД)
3. Технологический стек
4. UML диаграммы классов для двух отдельных компонентов из курсового – компонента доступа к данным и компонента с бизнес-логикой
5. Программная реализация компонента доступа к данным

2.2 Принципы организации компонентов

2.2.1 История

— Неперемещаемые библиотеки

— Перемещаемые библиотеки (связывающий загрузчик)

— Компоновщик (редактор связи) + загрузчик

Любая программа растет, пока не заполнит все доступное время на компиляцию и компоновку

2.2.2 Компоненты

Программные компоненты – динамически связываемые файлы, которые можно подключать во время выполнения (связывающий загрузчик).

2.2.3 Связность компонентов

Принцип эквивалентности повторного использования и выпусков (REP)

«Выпуск»:

- Номер версии
- Описание новой версии
- Change Log (Лог изменений)

Единица повторного использования – Единица выпуска

Классы и модули, объединяемые в компонент, должны выпускаться вместе

Объединение в один выпуск должно иметь **смысл** для автора и пользователей.

Принцип согласованного изменения (CPP)

Развитие принципов «единственной ответственности» (SRP) и «открытости/-закрытости» из SOLID

- В один компонент должны включаться классы, изменяющиеся по одним причинам и в одно время
- В разные компоненты должны включаться классы, изменяющиеся по разным причинам и в разное время

Принцип совместного и повторного использования (CRP)

Для **большинства** приложений простота сопровождения **важнее** возможности повторного использования.

Идея: Объединение в компонент классов, закрытых для одного и того же вида изменений.

Изменение требований ⇒ изменение **МИНИМАЛЬНОГО** количества компонентов

— Не вынуждаете пользователей компонента зависеть от того, что им не требуется

— Классы, не имеющие тесной связи, не должны включаться в компонент

2.2.4 Баланс

— Много ненужных выпусков

— Изменения затрагивают много компонентов

— Проблемы с повторным использованием

2.2.5 Сочетаемость компонентов

— Принцип ацикличности зависимостей

— Принцип устойчивых зависимостей

— Принцип устойчивости абстракций

Принцип ацикличности зависимостей (ADP)

Циклы в графе зависимостей недопустимы

Отдельные компоненты – отдельные разработчики/команды

Появление цикла – появление БОЛЬШОГО копмпонента

Разрыв цикла:

— Применить принцип DIP

— Создать новый компонент, от которого зависят проблемные

2.2.6 Проектирование сверху вниз?

— Граф зависимостей формируется для защиты стабильных и ценных компонентов от влияния изменчивых компонентов

— Структура компонентов отражает удобство сборки сопровождения

Поэтому она не проектируется полностью в начале разработки

2.2.7 Принцип устойчивых зависимостей

Зависимости должны быть направлены в сторону устойчивости

$$\text{Метрика неустойчивости} = \frac{\text{выходы}}{\text{входы} + \text{выходы}}$$

Метрика неустойчивости компонента должна быть выше метрик неустойчивости компонентов, от которых он зависит.

2.2.8 Принцип устойчивости абстракций

Устойчивости компонента пропорциональна его абстракции

Пример: Компоненты, содержащие только интерфейсы в C# и Java.