МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ВОЛГОГРАДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

на правах рукописи

И.В. Степанченко, С.И. Щербин, Р.О. Фокин

Программная инженерия

Учебное пособие

Волгоград
2021

УДК

Печатается по решению редакционно-издательского совета
Волгоградского государственного технического университета

Щербаков, М. В.
Программная инженерия: учеб. пособие / И.В. Степанченко, С.И. Щербин, Р.О. Фокин; ВолгГТУ. – Волгоград, 2021. – 71 с.
ISBN 978-5-9948-0000-0

В учебном пособии представлены основные положения программной инженерии для разработки сложных информационных систем, в том числе с компонентами искусственного интеллекта. Учебное пособие предназначено для магистров, обучающихся по программам магистратуры по профилю «искусственный интеллект» по направлениям 09.04.01 «Информатика и вычислительная техника», 09.04.03 «Прикладная информатика», 09.04.02 "Информационные системы и технологии".
Учебное пособие выполнено в рамках реализации гранта на разработку программ бакалавриата и программ магистратуры по профилю «Искусственный интеллект», а также на повышение квалификации педагогических работников образовательных организаций высшего образования в сфере искусственного интеллекта (конкурс 2021-ИИ-01 от 10.06.2021).

Ил. 1. Табл. -. Библиогр.: 9 назв.

ISBN 978-5-9948-0000-0

(c) Волгоградский государственный технический университет, 2021
И.В. Степанченко, С.И. Щербин, Р.О.Фокин, 2021

Оглавление

ВВЕДЕНИЕ ... 6

1. ПРОГРАММНАЯ ИНЖЕНЕРИЯ. ПРИНЦИПЫ ООП. МЕТОДЫ УПРАВЛЕНИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ... 7

1.1. Основные парадигмы ООП ... 7

1.1.1 Инкапсуляция и модификаторы доступа ... 8

1.1.2 Наследование и интерфейсы ... 8

1.1.3. Разница между интерфейсом и абстрактным классом ... 10

1.1.4. Полиморфизм ... 11

1.1.5. Перегрузки методов и операторов ... 12

1.1.6. Полиморфизм в языках со статической и динамической типизацией ... 13

1.2. Принципы SOLID как основные принципы проектирования в ООП ... 15

1.2.1 Проектирование классов на основе принципов единой ответственности и открытости/закрытости ... 17

1.2.2 Применение наследования классов и получения поведенческих свойств интерфейсов на основе принципов заменяемости и разделения интерфейсов ... 19

1.2.3 Разбор паттернов внедрения зависимостей на основе принципа инверсии зависимостей ... 21

1.3. Разработка программного обеспечения на основе объектно-ориентированной парадигмы ... 22

1.4. Управление разработкой проекта на различных этапах ... 26

1.4.1 Понятие проекта. Процессы проекта, организация команды и принятие решений, распределение ролей и ответственности, отслеживание состояния процесса, решение проблем в команде ... 26

1.4.2 Средства поддержки управления проектом. Организация документирования программных средств ... 30

1.4.3 Управление задачами проекта. Принцип работы и использование систем контроля версий 35
1.4.4 Принцип работы и использование систем контроля версий (практика) 37
2. ПРИНЦИПЫ ТЕСТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 38
2.1. Этапы тестирования ПО 38
2.1.1 Проектирование тестов 40
2.1.2 Выполнение тестового цикла, улучшение тестирования и качества финального продукта 41
2.1.3 Оптимизация тестирования ПО 42
2.2. Типы тестирования программного обеспечения 43
2.2.1 Регрессионное тестирование 44
2.2.2 Функциональное тестирование 45
2.2.3 Нагрузочное тестирование 46
2.2.3 Модульное тестирование 47
2.2.4 Оптимизационное тестирование 48
2.2.5 Тестирование интерфейса 48
2.2.6 Анализ исходного кода 49
2.2.7 Анализ документации 51
2.2.8 Финальное тестирование 52
2.3. Написание тестов 52
2.4. Тестирование разрабатываемого программного обеспечения 52
3. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ 53
3.1. Порождающие паттерны проектирования как принцип создания новых объектов и семейств объектов 53
3.2. Структурные паттерны проектирования как метод реализации иерархий классов 56
3.3. Поведенческие паттерны проектирования как метод реализации иерархий классов 57

3.4. Внедрение паттернов проектирования в разработку программного обеспечения ... 59
4. UML-ДИАГРАММЫ КАК ЯЗЫК ГРАФИЧЕСКОГО ОПИСАНИЯ В ОБЛАСТИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ... 61
4.1. Структурные диаграммы ... 61
4.2. Диаграммы поведения ... 64
4.3. Диаграммы взаимодействия ... 65
5. САМОСТОЯТЕЛЬНАЯ РАБОТА ... 67
ЗАКЛЮЧЕНИЕ ... 68
РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА ПО КУРСУ ... 69

ВВЕДЕНИЕ

Целью данного курса является формирование у студентов представления о задачах, методах и средствах программной инженерии как деятельности, нацеленной на создание программных продуктов, отвечающих потребностям заказчиков, с соблюдением плановых сроков и бюджета разработки.

Задачи дисциплины:

- изучение методов и принципов программной инженерии;
- получение навыков разработки программных систем на основе методологии программной инженерии;
- получение навыков организации работ по управлению разработкой программных систем.

1. ПРОГРАММНАЯ ИНЖЕНЕРИЯ. ПРИНЦИПЫ ООП. МЕТОДЫ УПРАВЛЕНИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1. Основные парадигмы ООП

Парадигма разработки – это набор правил и критериев, соблюдаемых разработчиками, чтобы выдержать конкретную стилистику и модель написания кода. Объектно-ориентированное программирование или ООП является одной из парадигм программирования. ООП ориентируется на данные и объекты, а не на функции и логические структуры. Обычно объекты в такой структуре представляют собой полноценные блоки данных, которые имеют определенный набор характеристик и возможностей [5].

Программный код, написанный с учётом принципов ООП, структурируется на четырех основных элементах:

1. Классы – это обобщенная форма данных в ООП. Они чаще всего описывают реальный объект и его характеристики. Например, класс человек имеет атрибуты фамилия, имя, отчество и т.д;
2. Объекты – это экземпляры существующих в программе классов. Например, для описанного выше класса Человек объектом будет конкретная личность, имеющая имя, фамилию, отчество и т.д;
3. Методы – это функции, описанные внутри объекта или класса;
4. Атрибуты – это конкретные характеристики объекта. Они имеют существующий в языке тип данных и их представляют в качестве переменных.

Помимо описанных структурных элементов, парадигма ООП основывается на четырех основных принципах:

1. Абстракция;
2. Инкапсуляция;
3. Наследование;
4. Полиморфизм

1.1.1 Инкапсуляция и модификаторы доступа

Инкапсуляция – это заключение данных и их функционала в оболочку. В случае с ООП данной оболочкой является класс. Он не только собирает все данные в одном месте, но также и защищает их от влияния извне. Такая функция именуется сокрытие. Для того, чтобы реализовать ограничения доступа к данным класса существуют модификаторы доступа – ключевые слова в объектно-ориентированных языках, которые задают параметры доступа для классов, методов и прочих элементов, также модификаторы являются специфичной частью языков программирования для облегчения инкапсуляции компонентов [9].

В зависимости от языка программирования, количество модификаторов доступа может различаться, но основными можно назвать три модификатора:

1. Public – разрешает обращаться к любым полям класса в любом месте программы;
2. Private – делает члены класса доступными только внутри самого класса. Менять извне ничего нельзя, также не получится считать данные для вывода;
3. Protected – позволяет использовать данные класса только его наследникам;

1.1.2 Наследование и интерфейсы

Наследование является механизмом повторного использования кода и способствует расширению программного обеспечения через открытые классы и интерфейсы. Установка отношений наследования между классами порождает иерархию классов. Например, мы создали класс с названием «Транспортное средство», данный класс будет иметь некоторые атрибуты: скорость, название, размер. Данные атрибуты слишком общие и для того, чтобы конкретизировать данные о транспортном средстве необходимо

создать еще один класс, например, класс «Автомобиль». При этом класс «Автомобиль» будет наследником класса «Транспортное средство», так называемым дочерним классом. В таком случае класс «Автомобиль» будет иметь все атрибуты класса «Транспортное средство» и при этом имеется возможность добавить уточняющие атрибуты: марка, пробег и т.д. Существует:

- Простое наследование. Описывает родство между двумя классами: один их них наследует второму. У одного класса может быть много наследников и даже в этом случае наследование остается простым;

- Множественное наследование. Подразумевает ситуацию, когда один потомственный класс принимает характеристики от нескольких родительских классов.

Для того, чтобы наладить множественное наследование объектов используют интерфейсы. Интерфейс – это программная структура, определяющая отношение между объектами, которые разделяют определенное поведенческое множество и не связаны никак иначе. Интерфейсы, наряду с абстрактными классами и протоколами, устанавливают взаимные обязательства между элементами программной системы. Интерфейс определяет границу взаимодействия между классами или компонентами, специфицируя определённую абстракцию, которую осуществляет реализующая сторона. Интерфейс в ООП является строго формализованным элементом объектно-ориентированного языка и широко используется в исходном коде программ.

Описание ООП-интерфейса состоит из двух частей: имени и методов интерфейса.

Имя интерфейса – строиться по тем же правилам, что и другие идентификаторы используемого языка программирования.

Методы интерфейса - в описании интерфейса определяются имена и сигнатуры входящих в него методов, то есть процедур или функций класса.

Использование интерфейсов возможно двумя способами:

- Класс может реализовывать интерфейс. Реализация интерфейса заключается в том, что в описании класса данный интерфейс указывается как реализуемый, а в коде класса обязательно определяются все методы, которые описаны в интерфейсе, в полном соответствии с сигнатурами из описания этого интерфейса. То есть, если класс реализует интерфейс, для любого экземпляра этого класса существуют и могут быть вызваны все описанные в интерфейсе методы. Один класс может реализовать несколько интерфейсов одновременно.
- Возможно объявление переменных и параметров методов как имеющих тип «интерфейс». В такую переменную или параметр может быть записан экземпляр любого класса, реализующего интерфейс. Если интерфейс объявлен как тип возвращаемого значения функции, это означает, что функция возвращает объект класса, реализующего данный интерфейс. Как правило, в объектно-ориентированных языках программирования интерфейсы, как и классы, могут наследоваться друг от друга. В этом случае интерфейс-потомок включает все методы интерфейса-предка и, возможно, добавляет к ним свои собственные.

1.1.3. Разница между интерфейсом и абстрактным классом

Можно заметить, что интерфейс, с формальной точки зрения, — это просто чистый абстрактный класс, то есть класс, в котором не определено ничего, кроме абстрактных методов. Абстрактный класс – это базовый класс, который не предполагает создания экземпляров. С помощью них реализуется ещё один принцип ООП – полиморфизм, речь о нем пойдет позже. Если язык программирования поддерживает множественное наследование и абстрактные методы, то необходимости во введении в синтаксис языка отдельного понятия «интерфейс» не возникает. Данные сущности описываются с помощью абстрактных классов и наследуются

классами для реализации абстрактных методов. Однако поддержка множественного наследования в полном объёме достаточно сложна и вызывает множество проблем, как на уровне реализации языка, так и на уровне архитектуры приложений. Введение понятия интерфейсов является компромиссом, позволяющим получить многие преимущества множественного наследования, не реализуя его в полном объёме и не сталкиваясь, таким образом, с большинством связанных с ним трудностей.

На уровне исполнения классическая схема множественного наследования вызывает дополнительный ряд неудобств:

- Если объект может параллельно наследовать n классов, существует n независимых способов к нему обращаться, а значит должно существовать n-1 дополнительных указателей на него. С точки зрения автоматического управления памятью это будет означать, что возникают ссылки, указывающие в середину объекта;
- Поддержка виртуальных вызовов подразумевает, что в объекте хранится ссылка на его виртуальную таблицу, а в случае множественного наследования n ссылок; активное использовании множественного наследования сильно увеличит объём памяти, занимаемый каждым объектом.

Использование схемы с интерфейсами (вместо множественного наследования) позволяет отбросить эти проблемы, если не считать вопроса о вызове интерфейсных методов. Классическое решение состоит в том, что интерфейсные методы вызываются менее эффективным способом, без помощи виртуальной таблицы: при каждом вызове сначала определяется конкретный класс объекта, а затем в нём ищется нужный метод.

1.1.4. Полиморфизм

Полиморфизм - свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Иными словами полиморфизм – это способность функции обрабатывать данные разных типов. Существует несколько разновидностей полиморфизма:

- параметрический;
- ad-hoc;

Параметрический полиморфизм является истинным, так как подразумевает исполнение одного и того же кода для всех допустимых типов аргументов. В данном типе полиморфная функция может работать с аргументами различного типа на основе поведения, а не значения, апеллируя лишь к необходимым ей свойствам аргументов, то есть исполнять физически один и тот же код для данных разных типов.

Ad-hoc при этом называют мнимым полиморфизмом так, как он представляет собой обеспечение косметической однородности потенциально разного исполнимого кода для каждого конкретного типа аргумента. Здесь полиморфная функция может работать с аргументами различного типа за счет переопределения. То есть одноименные функции с различными типами аргументов распознаются как разные функции.

1.1.5. Перегрузки методов и операторов

Существуют строго типизированные языки программирования, это значит, что вы не можете поместить строку в переменную типа int — сначала нужно провести преобразование. Так же и в метод нельзя передать параметр типа float, если при объявлении метода был указан тип double. Однако некоторые методы выполняются вне зависимости от переданного типа данных. К примеру, функция вывода на экран в любом известном языке программирования. Вывод производиться вне зависимости от типа данных переданного значения – это происходит потому, что у метода вывода на экран есть перегрузки. Перегрузки методов – это методы с таким же названием, но принимающие другие аргументы.

Помимо перегрузки методов, существует перегрузка операторов («+», «-» и т.д). Это работает также как и перегрузка методов, на одно и тоже название может существовать несколько вариантов в зависимости от значений.
Оба метода перегрузки являются способами реализации полиморфизма на практике.

1.1.6. Полиморфизм в языках со статической и динамической типизацией

У типизированных языков существует разделение на два вида типизации: статическая типизация и динамическая типизация. При статической типизации проверка типов данных происходит на этапе компиляции, а при динамической – на этапе выполнения.
Преимущества статической типизации:
- Проверки типов происходят только один раз — на этапе компиляции. Следовательно, не нужно озадачивать себя постоянной проверкой соответствия типов данных;
- Высокая скорость выполнения;
- При некоторых условиях существует возможность обнаружить ошибки на этапе компиляции;
Преимущества динамической типизации:
- Простота создания универсальных коллекций;
- Удобство описания обобщенных алгоритмов;
- Легкость в освоении;
Что касается полиморфизма, то Статический и динамический полиморфизм дополняют друг друга. Следует ясно представлять себе их преимущества и недостатки, чтобы использовать каждый из них там, где он дает наилучшие результаты, и сочетать их так, чтобы получить лучшее из обоих вариантов.
Динамический полиморфизм представляется в форме классов с виртуальными функциями и объектов, работа с которыми осуществляется

косвенно — через указатели или ссылки. Статический полиморфизм включает шаблоны классов и функций.

Динамический полиморфизм позволяет значению иметь несколько типов посредством открытого наследования. Благодаря своим характеристикам динамический полиморфизм наилучшим образом подходит для решения следующих задач:

- Единообразная работа, основанная на отношении надмножество/подмножество. Работа с различными классами, удовлетворяющими отношению надмножество/подмножество (базовый/производный), может выполняться единообразно;
- Статическая проверка типов;
- Динамическое связывание и раздельная компиляция. Код, который использует иерархию классов, может компилироваться отдельно от этой иерархии. Это становится возможным благодаря косвенности, обеспечиваемой указателями.
- Бинарная согласованность. Модули могут компоноваться как статически, так и динамически, до тех пор, пока схемы виртуальных таблиц подчиняются одним и тем же правилам.

Статический полиморфизм посредством шаблонов также позволяет значению иметь несколько типов. Статический полиморфизм наилучшим образом подходит для решения следующих задач:

- Единообразная работа, основанная на синтаксическом и семантическом интерфейсе. Работа с типами, которые подчиняются синтаксическому и семантическому интерфейсу, может выполняться единообразно. Интерфейсы в данном случае представляют синтаксическую сущность и не основаны на сигнатурах, так что допустима подстановка любого типа, который удовлетворяет данному синтаксису;
- Статическая проверка типов;
- Статическое связывание;

- Эффективность. Вычисления во время компиляции и статическое связывание позволяют достичь оптимизации и эффективности, недоступных при динамическом связывании;

Следует сочетать статический и динамический полиморфизм для того, чтобы получить преимущества обоих видов полиморфизма, а не для того, чтобы комбинировать их недостатки. К примеру:

- Статика помогает динамике. Используйте статический полиморфизм для реализации динамически полиморфных интерфейсов.
- Динамика помогает статике. Обобщенный, удобный, статически связываемый интерфейс может использовать внутреннюю динамическую диспетчеризацию, что позволяет обеспечить одинаковую схему размещения объектов.

1.2. Принципы SOLID как основные принципы проектирования в ООП

SOLID – это аббревиатура пяти основных принципов проектирования в объектно-ориентированном программировании: Single responsibility, Open-closed, Liskov substitution, Interface segregation и Dependency inversion.

Аббревиатура SOLID была предложена Робертом Мартином, автором нескольких книг, широко известных в сообществе разработчиков. Эти принципы позволяют строить на базе ООП масштабируемые и сопровождаемые программные продукты с понятной бизнес-логикой.

Расшифровка:

- Single responsibility — принцип единственной ответственности;
- Open-closed — принцип открытости / закрытости;
- Liskov substitution — принцип подстановки Барбары Лисков;
- Interface segregation — принцип разделения интерфейса;
- Dependency inversion — принцип инверсии зависимостей;

Принцип единственной обязанности / ответственности (single responsibility principle / SRP) обозначает, что каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс. Все его сервисы должны быть направлены исключительно на обеспечение этой обязанности

Принцип открытости / закрытости (open-closed principle / OCP) декларирует, что программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения. Это означает, что эти сущности могут менять свое поведение без изменения их исходного кода.

Принцип подстановки Барбары Лисков (Liskov substitution principle / LSP) в формулировке Роберта Мартина: «функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом».

Принцип разделения интерфейса (interface segregation principle / ISP) в формулировке Роберта Мартина: «клиенты не должны зависеть от методов, которые они не используют». Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют.

Принцип инверсии зависимостей (dependency inversion principle / DIP) — модули верхних уровней не должны зависеть от модулей нижних уровней, а оба типа модулей должны зависеть от абстракций; сами абстракции не должны зависеть от деталей, а вот детали должны зависеть от абстракций.

1.2.1 Проектирование классов на основе принципов единой ответственности и открытости/закрытости

Принцип единственной ответственности - один из пяти основных принципов объектно-ориентированного программирования и проектирования, сформулированных Робертом Мартином.

Принцип декларирует, что каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс, а все его сервисы должны быть направлены исключительно на обеспечение этой обязанности.

Следование принципу заключается обычно в декомпозиции сложных классов, которые делают сразу много вещей, на простые, ответственность которых очень специализирована. Но также и объединении в отдельный класс однотипной функциональности, которая может оказаться распределённой по многим классам, может рассматриваться как следование этому принципу.

Проектирование классов с направленностью на обеспечение единственной обязанности упрощает дальнейшие модификации и сопровождение, так как проще разобраться в одном блоке функциональности, нежели распутывать сложные взаимосвязи между различными функциональными блоками. Также при модификации логики в одном месте приложения снижаются риски возникновения проблем в других «неожиданных» его местах.

Следование SRP весьма полезная практика с точки зрения повторного использования кода. Сложные объекты с комплексными зависимостями обычно очень сложно использовать повторно, особенно если нужна только часть реализованного в них функционала. А небольшие классы с чётко очерченным функционалом, напротив, проще использовать повторно, так как они не избыточные и редко тянут за собой существенный объём зависимостей.

Принцип открытости/закрытости - один из пяти основных принципов объектно-ориентированного программирования и проектирования, сформулированных Робертом Мартином.

Принцип декларирует, что программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения. Это означает, что эти сущности могут менять свое поведение без изменения их исходного кода.

В этом контексте открытость для расширения — это возможность добавить для класса, модуля или функции новое поведение, если необходимость в этом возникнет, а закрытость для изменений — это запрет на изменение исходного кода программных сущностей. На первый взгляд, это звучит сложно и противоречиво. Но если разобраться, то принцип вполне логичен. Следование принципу OCP заключается в том, что программное обеспечение изменяется не через изменение существующего кода, а через добавление нового кода. То есть созданный изначально код остаётся «нетронутым» и стабильным, а новая функциональность внедряется либо через наследование реализации, либо через использование абстрактных интерфейсов и полиморфизм.

Принцип открытости/закрытости Мейера основывается на идее, что разработанная изначально реализация класса в дальнейшем не модифицируется (разве что исправляются ошибки), а любые изменения производятся через создание нового класса, который обычно наследуется от изначального. Согласно определению Мейера, реализация интерфейса может быть унаследована и переиспользована, но интерфейс может и измениться в новой реализации.

Позже был сформулирован полиморфный принцип открытости/закрытости. Он основывается на строгой реализации интерфейсов и на наследовании от абстрактных базовых классов или на полиморфизме. Созданный изначально интерфейс должен быть закрыт для модификаций, а новые реализации как

минимум соответствуют этому изначальному интерфейсу, но могут поддерживать и другие, более расширенные [1].

1.2.2 Применение наследования классов и получения поведенческих свойств интерфейсов на основе принципов заменяемости и разделения интерфейсов

Принцип подстановки Барбары Лисков - один из пяти основных принципов объектно-ориентированного программирования и проектирования, сформулированных Робертом Мартином.

Принцип в формулировке Роберта Мартина декларирует, что функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом. Оригинальное определение Барбары Лисков более формальное и заметно сложнее для восприятия: «В том случае, если q(x) — свойство, верное по отношению к объектам x некого типа T, то свойство q(y) тоже будет верным относительно ряда объектов y, которые относятся к типу S, при этом S — подтип некого типа T».

Следование принципу LSP заключается в том, что при построении иерархий наследования создаваемые наследники должны корректно реализовывать поведение базового типа. То есть если базовый тип реализует определённое поведение, то это поведение должно быть корректно реализовано и для всех его наследников.

LSP перекликается с контрактным программированием, определяя точные, формальные и верифицируемые описания интерфейсов. И интерфейсы, реализуемые наследниками, должны соответствовать контракту интерфейсов базового класса.

Наследник класса дополняет, но не заменяет поведение базового класса. То есть в любом месте программы замена базового класса на класс-наследник не должна вызывать проблем. Если по каким-то причинам так не

получается, то вероятнее всего имеет место либо некорректная реализация, либо неверно выбранная абстракция для наследования [2].

Соблюдение принципа подстановки Барбары Лисков позволяет гарантировать, что любой созданный нами подкласс будет без проблем использоваться ранее реализованными модулями, которые работали с надклассом. А это существенно упрощает расширение функциональных возможностей системы.

Но LSP, как и любой другой принцип, не является догмой. И иногда следование этому принципу при построении архитектуры может приводить к более ресурсоёмкой реализации, нежели работа с нарушением этого принципа.

Принцип разделения интерфейса - один из пяти основных принципов объектно-ориентированного программирования и проектирования, сформулированных Робертом Мартином.

Принцип в формулировке Роберта Мартина декларирует, что клиенты не должны зависеть от методов, которые они не используют. То есть если какой-то метод интерфейса не используется клиентом, то изменения этого метода не должны приводить к необходимости внесения изменений в клиентский код.

Следование принципу ISP заключается в создании интерфейсов, которые достаточно специфичны и требуют только необходимый минимум реализаций методов. Избыточные интерфейсы, напротив, могут требовать от реализующего класса создание большого количества методов, причём даже таких, которые не имеют смысла в контексте класса.

В чём-то принцип разделения интерфейса перекликается с принципом единственной ответственности — интерфейсы не должны быть избыточно «толстыми», если вдруг в приложении формируется слишком объёмный интерфейс, то есть высокая вероятность, что так происходит из-за того, что

в этом интерфейсе слишком много разных ответственностей, а значит логичнее всего провести декомпозицию сложного интерфейса на несколько простых.

Принцип разделения интерфейса снижает сложность поддержки и развития приложения. Чем проще и минималистичнее используемый интерфейс, тем менее ресурсоёмкой является его реализация в новых классах, тем меньше причин его модифицировать, но и в случае модификации она будет значительно проще.

1.2.3 Разбор паттернов внедрения зависимостей на основе принципа инверсии зависимостей

Принцип инверсии зависимостей - один из пяти основных принципов объектно-ориентированного программирования и проектирования, сформулированных Робертом Мартином.

Принцип декларирует, что модули верхних уровней не должны зависеть от модулей нижних уровней, а оба типа модулей должны зависеть от абстракций; сами абстракции не должны зависеть от деталей, а вот детали должны зависеть от абстракций.

Следование принципу инверсии зависимостей «заставляет» реализовывать высокоуровневые компоненты без встраивания зависимостей от конкретных низкоуровневых классов, что, например, сильно упрощает замену используемых зависимостей как по бизнес-требованиям, так и для целей тестирования. При этом зависимость формируется не от конкретной реализации, а от абстракции - реализуемого зависимостью интерфейса.

Например, мы реализуем хранение документов в веб-приложении. На первый взгляд, кажется логичным добавить зависимость от модулей работы с файловой системой непосредственно в класс, отвечающий за высокоуровневую работу с этими документами. Но в перспективе такая зависимость может создать проблемы - например, нам потребуется хранить

данные не только на диске, но и в облаке. Если зависимость внедрена от реализации, то мы столкнёмся с необходимостью её переработки. Если же зависимость выведена на уровень абстракции, то нам будет достаточно реализовать функционал работы с облаком, соответствующий ранее созданному интерфейсу работы с файлами.

Принцип инверсии зависимостей часто упрощает следованию принципу подстановки Барбары Лисков. Выделение абстракций и встраивание зависимостей от них снижает вероятность того, что в новом классе мы не полностью реализуем контракт базового класса, который мы расширяем в рамках нового.

1.3. Разработка программного обеспечения на основе объектно-ориентированной парадигмы

Реализацией технологии ООП в рамках спиральной модели является получившая в последнее время широкое распространение технология быстрой разработки приложений RAD (Rapid Application Development).

Основные принципы (концепции) технологии RAD:

– разработка приложений итерациями;

– необязательность полного завершения работ на каждом из этапов ЖЦ;

– обязательное вовлечение пользователей в процесс разработки АИС;

– необходимое применение CASE-средств, обеспечивающих целостность проекта;

– применение средств управления конфигурацией, облегчающих внесение изменений в проект и сопровождение готовой системы;

– использование прототипирования, позволяющее полнее выяснить и удовлетворить потребности конечного пользователя;

– тестирование и развитие проекта одновременно с его разработкой;

– ведение разработки немногочисленной хорошо управляемой командой профессионалов;

– грамотное руководство разработкой системы, четкое планирование и контроль выполнения работ.

На рисунке 1 изображена архитектура программы при использовании технологии ООП.

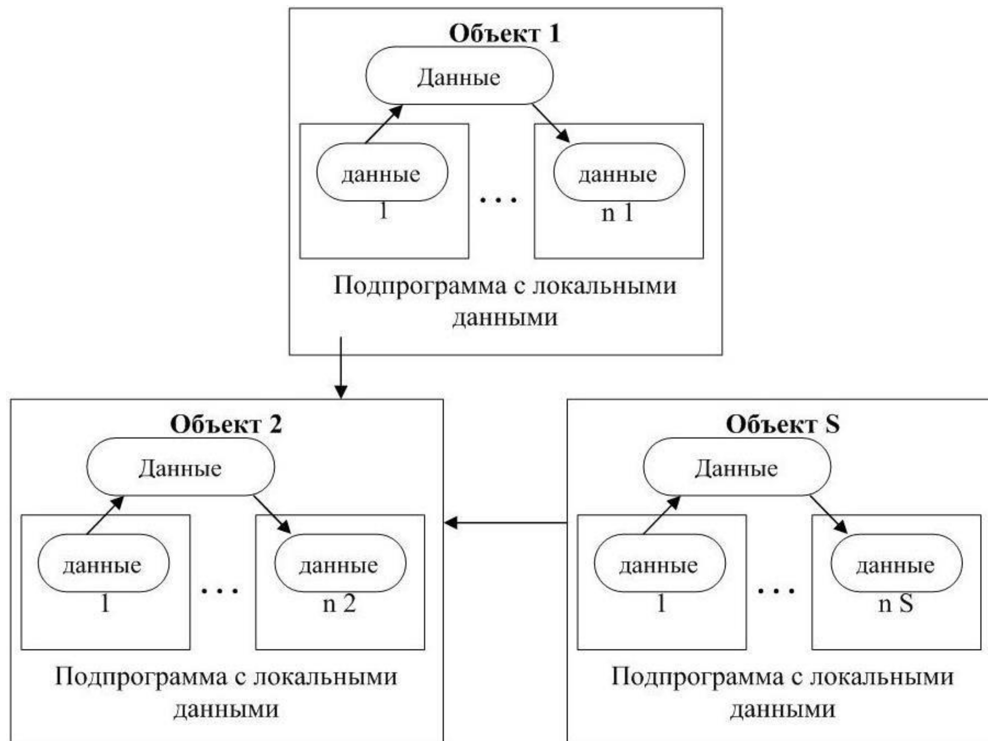

Рисунок 1 – Архитектура программы при технологии ООП

Процесс разработки программных систем по технологии RAD содержит следующие требования:

– небольшую команду программистов (от 2 до 10 человек);

– короткий производственный график (от 2 до 6 мес.);

– повторяющийся цикл, при котором разработчики по мере того, как приложение начинает обретать форму, запрашивают и реализуют в продукте требования, полученные через взаимодействие с заказчиком.

Жизненный цикл программы, выполненной в соответствии с технологией RAD, содержит следующие этапы:

- Анализ и планирование;

- Проектирование;

- Реализация;

- Внедрение;

На этапе анализа и планирования пользователи системы определяют функции и требования АИС, выделяют наиболее приоритетные функции, описывают информационные потоки. Определение требований выполняется в основном силами пользователей под руководством специалистов-разработчиков. Ограничивается масштаб проекта, определяются временные рамки для каждого из последующих этапов. Результатом данного этапа являются техническое задание на разработку АИС.

На этапе проектирования пользователи принимают участие в техническом проектировании системы под руководством специалистов-разработчиков. CASE-средства используются для быстрого получения работающих прототипов приложений. Пользователи, непосредственно взаимодействуя с ними, уточняют и дополняют требования к системе. Более подробно рассматриваются процессы системы. Анализируется и, при необходимости, корректируется функциональная схема (модель). Каждая функция рассматривается детально. При необходимости для каждого элементарного процесса создается частичный прототип: экран, диалог, отчет, устраняющий неясности или неоднозначности. Определяются требования разграничения доступа к данным. На этом этапе формируется список необходимой документации.

После детального определения состава процессов оценивается количество функциональных элементов разрабатываемой системы и принимается решение о разделении АИС на подсистемы, поддающиеся реализации одной командой разработчиков за приемлемое для RAD-проектов время (60 – 90 дней). Проект распределяется между различными командами (делится функциональная модель).

Результаты этапа:

– общая информационная модель системы;

– функциональные модели системы в целом и подсистем, реализуемых отдельными командами разработчиков;

– точно определенные интерфейсы между автономно разрабатываемыми подсистемами;

– прототипы экранов, отчетов, диалогов.

На этапе реализации выполняется непосредственно быстрая разработка приложения – разработчики производят итеративное построение реальной системы на основе полученных на предыдущем этапе моделей. Программный код частично формируется при помощи автоматических генераторов, получающих информацию непосредственно из репозитория CASE-средств. Конечные пользователи оценивают получаемые результаты и вносят коррективы, если в процессе разработки система перестает удовлетворять определенным ранее требованиям. Тестирование системы осуществляется непосредственно в процессе разработки.

После окончания работ каждой отдельной команды разработчиков производится постепенная интеграция данной части системы с остальными, формируется полный программный код, выполняется тестирование совместной работы данной части приложения с остальными, а затем тестирование системы в целом. Результатом этапа является готовая система, удовлетворяющая всем согласованным требованиям.

На этапе внедрения производится обучение пользователей, организационные изменения и опытная эксплуатация новой системы.

Технология RAD, соответствующая парадигме ООП, наряду с неоспоримыми преимуществами, обладает рядом существенных недостатков:

– отсутствие стандартов компоновки двоичных результатов компиляции объектов в единое целое даже в рамках одного языка программирования;

– взаимодействия между объектами требует разработки интерфейса, а, следовательно, дополнительных затрат времени и возникновение возможности ошибки в коде;

– изменение реализации одного объекта требует перекомпиляции всего программного продукта.

Таким образом, технология RAD эффективна для программных проектов средней сложности под конкретного заказчика. Разработка сложных программных систем (операционные системы, системы реального масштаба времени), т.е. программы с большим процентом уникального кода, требуют более высокого уровня планирования и жесткой дисциплины проектирования.

1.4. Управление разработкой проекта на различных этапах

Управление проектами - применение знаний, навыков, инструментов и техник при выполнении проектной деятельности для достижения требований проекта и запланированных результатов [3].

1.4.1 Понятие проекта. Процессы проекта, организация команды и принятие решений, распределение ролей и ответственности, отслеживание состояния процесса, решение проблем в команде

В различных источниках можно найти разнообразные определения понятия «проект»; в целом, они не противоречат, а дополняют друг друга. Проект – это:

- что-либо, что задумывается или планируется, например, большое предприятие;
- некоторая задача с определенными исходными данными и требуемыми результатами (целями), обуславливающими способ ее решения. Проект включает в себя замысел (проблему), средства его реализации (решения проблемы) и получаемые в процессе реализации результаты;

- уникальный процесс, состоящий из набора взаимоувязанных и контролируемых работ с датами начала и окончания и предпринятый, чтобы достичь цели соответствии конкретным требованиям, включая ограничения по времени, затратам и ресурсам;
- целенаправленная деятельность временного характера, направленная на создание уникального продукта или услуги;

Ключевыми особенностями проекта являются:

- строгие и обоснованные цели, которые должны быть достигнуты с одновременным выполнением ряда технических, экономических и других требований;
- наличие внутренних и внешних взаимосвязей операций, задач и ресурсов, которые требуют четкой координации при выполнении проекта, что создает возможность представления в виде комплекса взаимоувязанных работ;
- определенные сроки начала и конца проекта;
- ограниченные ресурсы;
- определенная степень уникальности целей проекта и условий его осуществления;
- неизбежность различных конфликтов.

Управление проектом осуществляется посредством надлежащего применения и интеграции 49 процессов управления проектом, логически сгруппированных по областям знаний. Процессы управления проектом объединены в пять групп:

- инициация;
- планирование;
- исполнение;
- мониторинг и контроль;
- закрытие.

Процессы управления проектом связаны между собой входами и выходами, где конечный результат одного процесса может являться входом другого, однако не обязательно находится в той же группе процессов.

Поскольку для выполнения проекта создается команда, важная роль отводится разработке организационной структуры проекта. Необходимость разработки организационной структуры объясняется тем, что для выполнения проекта создается команда проекта - новый временный рабочий коллектив, состоящий из специалистов различных структурных подразделений компаний со стороны Исполнителя и со стороны Заказчика. Как и для любого нового коллектива, для членов команды проекта необходимо определить проектные роли (временные должности), функции, обязанности, ответственность, полномочия и правила взаимодействия, а также организационную схему, отражающую отношения подчиненности. При этом несущественно, на какой период времени будет создаваться команда проекта - на несколько месяцев или на несколько лет. Структура проекта определяется сложностью, масштабностью разработки и внедрения продукты (например, информационной системы), количеством и специализацией членов команды проекта. В команду проекта могут включаться специалисты, как на полную, так и на частичную занятость. Если внедрение информационной системы осуществляется с привлечением сторонней организации - Исполнителя, то для успешного внедрения необходимо сформировать команду проекта не только от Исполнителя, но и от Заказчика, после чего определить допустимые взаимодействия между членами команд Исполнителя и Заказчика (кто, с кем, по каким вопросам взаимодействует), т. е. установить правила взаимодействия.

При работе нескольких людей вместе над одним проектом существует вероятность возникновения проблем внутри проекта. Некоторые из таких проблем типовые и шаблоны их решения разработаны. Основные проблемы и варианты их решения:

- Много идей и обсуждений в команде, но работа движется медленно. При такой проблеме нет двух ролей в проекте – специалист, профессионал, который любит и знает свою работу, но за свои границы не выходит, а также лид, лидер команды разработки, он нацелен на результат проекта и согласен для этого делать любую работу, которую нужно. Он понимает, что что-то он будет делать непрофессионально, тратить время, но на это тоже согласен.
- Отсутствие новых идей, ступор. При такой проблеме явно отсутствует в команде человек на роли генератора идей и исследователя внешних ресурсов. Генератор активно порождает оригинальные идеи и видит самореализацию в их воплощении. Исследователь много читает, активно общается и приносит новые идеи из внешнего мира: новые технологии, которые хорошо бы применить.
- Некому завершать работу. При такой проблеме в команде отсутствует роль педанта. Человек на данной роли следит за тем, чтобы были выполнены все аспекты задачи на 100%.

Для того, чтобы отслеживать состояние проекта во времени существует мониторинг. Мониторинг – это аспект управления проектом, осуществляемый на протяжении всего проекта. Мониторинг включает в себя сбор, измерение и распространение информации об исполнении, а также оценку измерений и тенденций для оказания влияния на улучшение процесса. Постоянный мониторинг дает команде управления проектом возможность понимать общее состояние проекта и определять, на какие области следует обратить особое внимание. Для того, чтобы обеспечить мониторинг процесса проекта от команды требуется каждый раз оформлять отчет об исполнении. Отчеты, составляемые командой проекта, должны содержать детальное описание работ, достижений, контрольных событий, выявленных вопросов и проблем. Отчеты об исполнении могут использоваться для сообщения ключевой информации, включающей в себя, среди прочего:

- текущий статус;
- существенные достижения за указанный период времени;
- внесенные в расписание операции;
- прогнозы;
- проблемы.

Для упрощения данного процесса были разработаны специальные программы, направленные на автоматизацию процесса управления проектами.

1.4.2 Средства поддержки управления проектом. Организация документирования программных средств

Документация является органической, составной частью программного продукта для ЭВМ и требуются значительные ресурсы для ее создания и применения. Тексты и объектный код программ для ЭВМ могут стать программным продуктом только в совокупности с комплексом документов, полностью соответствующих их содержанию и достаточных для его освоения, применения и изменения. Для этого документы должны быть корректными, строго адекватными текстам программ и содержанию баз данных – систематически, структурировано и понятно изложены, для возможности их успешного освоения и использования достаточно квалифицированными специалистами различных рангов и назначения. Качество и полнота отображения в документах процессов и продуктов в жизненном цикле программных средств должны полностью определять достоверность информации для взаимодействия заказчиков, пользователей и разработчиков, а тем самым, корректность функций и достигаемое качество программных продуктов и соответствующих систем. Посредством документов (электронных или бумажных) специалисты взаимодействуют с программными средствами и данными в реализующих их вычислительных машинах, а также между собой.

Существует большая разница между тем, чтобы просто написать и запрограммировать некоторую функцию для индивидуального использования её разработчиком, и тем, чтобы изготовить её как качественный программный продукт, отчуждаемый от разработчиков, поставляемый заказчику и пользователям. Создание программного продукта требует значительных организационных усилий, ибо её документация – это сложный живой организм, подверженный постоянным изменениям, которые могут вноситься многими специалистами. Управление документацией должно непрерывно поддерживать её полноту, корректность и согласованность с программным продуктом. Необходимо обеспечивать возможность достоверного, формально точного общения всех участников проекта ПС между собой, с создаваемым продуктом и с документами для гарантии поступательного развития, совершенствования и применения комплекса программ. Адекватность документации требованиям, состоянию текстов и объектных кодов программ должна инспектироваться и удостоверяться (подписываться) ответственными руководителями и заказчиками проекта. Ошибки и дефекты документов не менее опасны для применения ПС, чем ошибки в структуре, интерфейсах, файлах текстов программ и в содержании данных. Поэтому к разработке, полноте, корректности и качеству документации необходимо столь же тщательное отношение, как к разработке и изменениям текстов программ и данных.

Реализация документов ПС в значительной степени определяет достигаемое качество сложных программных продуктов, трудоемкость и длительность их создания. Для этого должна формироваться и использоваться регламентированная стратегия, стандарты, распределение ресурсов и планы создания, изменения и применения документов на программы и данные сложных систем. В общем случае должны быть выделены руководители и коллектив специалистов, которые будут планировать, описывать,

утверждать, выпускать, распространять и сопровождать комплекты документов. Они должны стимулировать разработчиков ПС осуществлять непрерывное, полноценное документирование процессов и результатов своей деятельности, а также контролировать полноту и качество исходных, результирующих и отчетных документов ЖЦ ПС. Официальная, описанная и утвержденная стратегия документирования должна устанавливать дисциплину, необходимую для эффективного создания высо-кокачественных документов на продукты и процессы в жизненном цикле ПС.

Методы и средства документирования каждой процедуры в стандартах обычно не раскрываются и адресуются к специальным нормативным документам различного уровня. Быстро оснащающиеся различными ме-тодами и инструментальными средствами этапы системного анализа, моделирования и проектирования ПС различных классов и назначения затрудняют стандартизацию этих процессов, достаточную для полной формализации структуры и содержания документов на программы и данные на уровне международных стандартов. Поэтому для этих этапов создаются нормативные документы – шаблоны на уровне стандартов дефакто, использующие, адаптирующие и дополняющие компоненты стандартов де-юре в разумной степени. Такие нормативные документы содержат выделенные фрагменты стандартов ЖЦ ПС и других стандартов, регламентирующих шаблоны программных документов на различных этапах проекта. В результате создаются и применяются проблемно-ориентированные совокупности методических руководств, отражающие наиболее современные методы, формы и фрагменты документов, для документальной поддержки этапов и процессов жизненного цикла ПС, определенного класса или функционального назначения.

Процессы документирования программ и данных входят в весь жизненный цикл сложных систем и ПС. Поэтому организация и реализация работ по

созданию документов должны распределяться между специалистами, ведущими непосредственное и преимущественное создание проектов комплексов программ и специалистами осуществляющими, в основном, разработку, контроль и издание документов. При создании особо сложных систем целесообразно выделение специального коллектива, обеспечивающего организацию и реализацию основных системных работ по документообороту ПС. Совокупные затраты на документирование крупных программных продуктов могут достигать 20 – 30% от общей трудоемкости проекта и необходимого числа (десятки) специалистов в жизненном цикле проекта ПС. В более простых случаях, организация работ может быть упрощена, затраты на документирование снижаются приблизительно до 10%, однако всегда целесообразно выделять специалистов, непосредственно ответственных за создание, адекватность и контроль полноценного комплекта документов на программный продукт. Состав и общий объем документов широко варьируется в зависимости от класса и характеристик объекта разработки, а также в зависимости от используемой технологии. Наиболее сложному случаю разработки критических ПС реального времени высокого качества соответствует самая широкая номенклатура документов. Такой перечень документов может быть использован как базовый для формирования на его основе состава и шаблонов документов в остальных более простых проектах.

Создание и применение ПС сложных систем сопровождается документированием этих объектов и процессов их жизненного цикла для обеспечения возможности освоения и развития функций программных средств и баз данных на любых этапах проекта ПС, а также для обеспечения интерфейса между разработчиками и с пользователями. По своему назначению и ориентации на определенные задачи и группы пользователей, документацию ПС можно разделить на:

- технологическую документацию процессов разработки и обеспечения всего жизненного цикла, включающую подробные технические описания, и подготавливаемую для специалистов, ведущих проектирование, разработку и сопровождение комплексов программ, обеспечивающую возможность отчуждения, детального освоения, развития и корректировки ими программ и данных на всем жизненном цикле ПС;
- эксплуатационную документацию программного продукта – объекта и результатов разработки, создаваемую для конечных пользователей ПС и позволяющую им осваивать и квалифицированно применять эти средства для решения конкретных функциональных задач систем.

Технологическая документация, непосредственно и в наибольшей степени должна отражать процессы жизненного цикла комплексов программ и данных, и требования к этим документам. Стандарты и нормативные документы, входящие в жизненный цикл проекта ПС, должны регламентировать структуру, состав этапов, работ и доку-ментов ЖЦ ПС. Они должны: формализовать выполнение и документирование конкретных работ при проектировании, разработке и сопровождении ПС; обеспечивать адаптацию документов к характеристикам среды разработки, внешней и операционной системы; регламентировать процессы обеспечения качества ПС и его компонентов, методы и средства их достижения, реальные значения достигнутых показателей качества. Для контроля возможных изменений целесообразно предусматривать и согласовывать с заказчиком специальный документ, регламентирующий правила применения и корректировки номенклатуры, а также состава и содержания документации, поддерживающей ЖЦ ПС.

Эксплуатационная документация должна обеспечивать отчуждаемость программного продукта от первичных поставщиков – разработчиков и возможность освоения и эффективного применения комплексов программ достаточно квалифицированными специалистами – пользователями.

Эксплуатационные документы должны исключать возможность некорректного использования ПС за пределами условий эксплуатации, при которых документами гарантируются требуемые показатели качества функционирования ПС. Основная ее задача состоит в фиксировании, полноценном использовании и обобщении результатов функционировании объектов и процессов всего жизненного цикла ПС и системы.

1.4.3 Управление задачами проекта. Принцип работы и использование систем контроля версий

Система контроля версий одна из важнейших возможностей при разработке какого-либо проекта. Система контроля версий — это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии. Она позволяет вернуть файлы к состоянию, в котором они были до изменений, вернуть проект к исходному состоянию, увидеть изменения, увидеть, кто последний менял что-то и вызвал проблему, кто поставил задачу и когда и многое другое. Использование СКВ также значит в целом, что, если вы сломали что-то или потеряли файлы, вы спокойно можете всё исправить. В дополнение ко всему вы получите всё это без каких-либо дополнительных усилий. Существуют:

- локальные системы контроля версий;
- централизованные системы контроля версий;
- распределенные системы контроля версий;

Локальные системы осуществляются путем копирования файлов проекта в отдельный каталог. Данный подход очень прост в использовании, однако сильно подвержен появлению ошибок. Очень легко забыть путь к каталогу, запутаться в каком каталоге вы находитесь и скопировать или изменить не тот файл, что может привести к неустранимой неисправности. Для решения

данной проблемы были разработаны локальные системы контроля версий с простой базой данных, которая хранит записи о всех изменениях в файлах.
С использованием локальных систем, разработчики столкнулись с ещё одной проблемой – необходимость взаимодействовать с другими разработчиками. Для решения этой проблемы были разработаны централизованные системы контроля версий. Они используют единый сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из центрального хранилища. Такой подход имеет множество преимуществ, особенно перед локальными СКВ. Например, все разработчики проекта в определённой степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать ЦСКВ, чем оперировать локальными базами данных на каждом клиенте.
Несмотря на это, данный подход тоже имеет серьёзные минусы. Самый очевидный минус — это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всё — всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков. Локальные СКВ страдают от той же самой проблемы: когда вся история проекта хранится в одном месте, вы рискуете потерять всё.
Для решения всех этих проблем были разработаны распределенные системы контроля версий. В распределенных системах контроля версий клиенты не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени) — они полностью копируют репозиторий. В этом случае,

если один из серверов, через который разработчики обменивались данными, умрёт, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных. Также многие распределенные системы контроля версий могут одновременно взаимодействовать с несколькими удалёнными репозиториями. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

1.4.4 Принцип работы и использование систем контроля версий (практика)

Для того, чтобы начать использовать систему контроля версий её необходимо установить на свой компьютер. Для этого необходимо найти официальные пакеты установки GIT и выбрать подходящий в зависимости от используемой операционной системы.

После установки необходимо произвести первичную настройку системы. Первое, что необходимо сделать после установки – указать имя и электронную почту пользователя. Далее – необходимо выбрать текстовый редактор, которых будет использоваться для набора сообщений в GIT.

После этого необходимо настроить ветвь по умолчанию. После всех операций необходимо проверить все существующие настройки.

2. ПРИНЦИПЫ ТЕСТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1. Этапы тестирования ПО

Тестирование программного обеспечения (ПО) – это процесс исследования ПО с целью выявления ошибок и определения соответствия между реальным и ожидаемым поведением ПО, осуществляемый на основе набора тестов, выбранных определённым образом. В более широком смысле, тестирование ПО – это техника контроля качества программного продукта, включающая в себя проектирование тестов, выполнение тестирования и анализ полученных результатов.

Очень часто современные программные продукты разрабатываются в сжатые сроки и при ограниченных бюджетах проектов. Программирование сегодня перешло из разряда искусства в разряд ремесел для многих миллионов специалистов. Но, к сожалению, в такой спешке разработчики зачастую игнорируют необходимость обеспечения защищённости своих продуктов, подвергая тем самым пользователей неоправданному риску. Контроль качества (тестирование) считается важным в процессе разработки ПО, потому что обеспечивает безопасность, надёжность, удобство создаваемого продукта. В настоящее время существует великое множество подходов и методик к решению задачи тестирования ПО, но эффективное тестирование сложных программных систем - процесс творческий, не сводящийся к следованию строгим и чётким правилам.

Процесс тестирования состоит из следующих этапов:

- Планирование тестирования;
- Мониторинг и контроль тестирования;
- Анализ тестирования;
- Проектирование тестов;
- Реализация тестов;

- Выполнение тестов;
- Завершение тестирования;

Это не строгая последовательность выполнения действий. Активности могут меняться местами во время всего процесса тестирования, идти параллельно или даже исключаться.

Планирование тестирования - на этом этапе определяется стратегия тестирования и пишется тест-план. Тестировщику необходимо продумать объемы тестирования, сколько потребуется людей, какие нужны девайсы, какие есть риски и так далее. А для этого надо хорошо знать продукт, чтобы декомпозировать его на составные части и оценить объем проверки каждой.

Мониторинг и контроль тестирования - предполагает непрерывное сравнение фактического хода работы с планом тестирования, используя любые метрики мониторинга тестирования. Цель мониторинга — сбор информации и обеспечение обратной связи о состоянии тестирования. То есть если что-то у нас идет не так как задумано, мы должны узнать об этом как можно раньше, чтобы минимизировать риски для разработки продукта. Этот этап характерен для любой активности в тестировании и даже во время всего жизненного цикла разработки ПО.

Анализ тестирования состоит из нескольких активностей. Одна из них — анализ базиса тестирования. В роли базиса выступают техническое задание (ТЗ), документация по проекту, которая дает примерное представление о продукте. Например, спецификации требований такие как бизнес-требования, функциональные требования, системные требования, пользовательские истории. Это такие документы, которые описывают функциональные и нефункциональные компоненты или поведение системы в целом. Выявление дефектов в ходе анализа тестирования является важным потенциальным преимуществом. Такие активности по анализу тестирования не только проверяют, являются ли требования согласованными, сформулированными должным образом и полными, но

также проверяют, правильно ли требования отражают потребности клиентов, пользователей и других заинтересованных лиц.

Проектирование тестов подразумевает создание тест-кейсов и чек-листов, которые будут описывать ход проверки. На этом этапе важно обеспечить максимально полное и эффективное покрытие тестами проект, чтобы он стал качественным в итоге.

Реализация тестов. На этом этапе определяется все ли готово к тестированию, например, настроено ли тестовое окружение (все ли устройства, на которых будет проводиться тестирование, есть), поставлены ли нужные программы. Здесь тест-кейсы выстраиваются в определенном порядке, чтобы облегчить выполнение тестов.

Выполнение тестов – конкретное выполнение всех разработанных на предшествующих этапах действий.

На этапе завершения тестирования создаются итоговые отчеты о результате тестирования и передача их всем заинтересованным лицам. Проверяется, что все необходимые отчеты о найденных ошибках исправлены, проверены и закрыты. Анализируется полученный опыт, чтобы определить изменения, необходимые для обеспечения более качественного тестирования в будущем.

2.1.1 Проектирование тестов

Данный этап тестирования ПО иначе называют тест-дизайн — это этап тестирования ПО, на котором проектируются и создаются тестовые случаи (тест-кейсы). Выделяют следующие техники тест-дизайна:

- Тестирование на основе классов эквивалентности (equivalence partitioning) — это техника, основанная на методе чёрного ящика, при которой мы разделяем функционал (часто диапазон возможных вводимых значений) на группы эквивалентных по своему влиянию на систему значений.

- Техника анализа граничных значений (boundary value testing) — это техника проверки поведения продукта на крайних (граничных) значениях входных данных.
- Попарное тестирование (pairwise testing) — это техника формирования наборов тестовых данных из полного набора входных данных в системе, которая позволяет существенно сократить количество тест-кейсов.
- Тестирование на основе состояний и переходов (State-Transition Testing) — применяется для фиксирования требований и описания дизайна приложения.
- Таблицы принятия решений (Decision Table Testing) — техника тестирования, основанная на методе чёрного ящика, которая применяется для систем со сложной логикой.
- Доменный анализ (Domain Analysis Testing) — это техника основана на разбиении диапазона возможных значений переменной на поддиапазоны, с последующим выбором одного или нескольких значений из каждого домена для тестирования.
- Сценарий использования (Use Case Testing) — Use Case описывает сценарий взаимодействия двух и более участников (как правило — пользователя и системы).

2.1.2 Выполнение тестового цикла, улучшение тестирования и качества финального продукта

Тестовый цикл – это цикл исполнения тестов, включающий этапы выполнения тестов и анализа результатов тестового процесса. Тестовый цикл заключается в прогоне разработанных тестов на некотором однозначно определяемом срезе системы (состоянии кода разрабатываемой системы). Обычно такой срез системы называют build. Тестовый цикл включает следующую последовательность действий:

1. Проверка готовности системы и тестов к проведению тестового цикла включающая:

- Проверку того, что все тесты, запланированные для исполнения на данном цикле, разработаны и помещены в систему версионного контроля.

- Проверку того, что все подсистемы, запланированные для тестирования на данном цикле, разработаны и помещены в систему версионного контроля.

- Проверку того, что разработана и задокументирована процедура определения и создания среза системы, или build.

- Проверки некоторых дополнительных критериев.

2. Подготовка тестовой машины в соответствии с требованиями, определенными на этапе планирования (например, полная очистка и переустановка системного программного обеспечения). Конфигурация тестовой машины, так же, как и срез системы, должны быть однозначно воспроизводимыми.
3. Воспроизведение среза системы.
4. Прогон тестов в соответствии с задокументированными процедурами.
5. Сохранение тестовых протоколов (test log). Test log может содержать вывод системы в STDOUT, список результатов сравнения полученных при исполнении данных с эталонными или любые другие выходные данные тестов, с помощью которых можно проверить правильность работы системы.
6. Анализ протоколов тестирования и принятие решения о том прошел или не прошел каждый из тестов (Pass/Fail).
7. Анализ и документирование результатов цикла.

Последний перед выпуском продукта тестовый цикл не должен включать изменений кода build или кода продукта тестируемой системы. Этот цикл называется "финальным". Таким образом обеспечивается ситуация, когда финальный цикл полностью повторяем, а выпускаемый продукт полностью совпадает с продуктом, который прошел тестирование. Финальный цикл необходим для гарантии достоверности результатов тестирования.

2.1.3 Оптимизация тестирования ПО

Подход к оптимизации тестов напрямую зависит от степени автоматизации процесса тестирования. Они могут быть ручными, автоматизированными и

автоматическими. Третий вариант на данный момент практически нереализуем, поэтому будут рассмотрены оставшиеся два.

Оптимизации применимые к ручным тестам. При тестировании вручную, прежде всего, необходимо выделить задачи, которые выполняются одинаково. Чаще всего они простые, но их много и приходиться выполнять их между основными тестами. Такие задачи следует переложить на автоматическую систему, к примеру написать собственные скрипты генерации тестов.

Оптимизация автоматизированных тестов. Есть ряд наиболее распространенных оптимизаций, применимых к тестам, проводимым автоматизировано:

- Параллельность тестов;
- Удаление старых тестов;
- Применение техник тест-дизайна для оптимизации наборов тест-кейсов;
- Перенос существующих тестов и проверок на другой уровень;

Для любой оптимизации нужно четко для себя определить текущие проблемы в процессе тестирования, по пунктам разложить в чем они заключаются, представить возможные варианты их решения. После этого требуется озвучить их в команде, и только лишь после всеобщего одобрения распределить усилия и решить поставленные задачи.

2.2. Типы тестирования программного обеспечения

Существует достаточно большое количество типов тестирования. Их выделяют по группам.

Уровни тестирования:

- модульное тестирование;
- интеграционное тестирование;
- системное тестирование;

Также тестирование классифицируется по определенным признакам. По объекту тестирования выделяют:

- функциональное тестирование;
- тестирование производительности;
- нагрузочное тестирование;
- стресс-тестирование;
- тестирование стабильности;
- тестирование безопасности;
- тестирование совместимости;

По знанию системы выделяют:

- тестирование черного ящика;
- тестирование белого ящика;

По времени проведения выделяют:

- альфа-тестирование;
- бета-тестирование;
- регрессионное тестирование;
- дымовое тестирование;

По степени автоматизации выделяют:

- ручное тестирование;
- автоматизированное тестирование;

2.2.1 Регрессионное тестирование

Регрессионное тестирование — это вид тестирования, направленный на проверку изменений, сделанных в приложении или окружающей среде, для подтверждения того факта, что существующая ранее функциональность работает, как и прежде. Регрессионными могут быть как функциональные, так и нефункциональные тесты. Из опыта разработки ПО известно, что повторное появление одних и тех же ошибок — случай достаточно частый. Иногда это происходит из-за слабой техники управления версиями или по

причине человеческой ошибки при работе с системой управления версиями. Поэтому считается хорошей практикой при исправлении ошибки создать тест на неё и регулярно прогонять его при последующих изменениях программы. Хотя регрессионное тестирование может быть выполнено и вручную, но чаще всего это делается с помощью специализированных программ, позволяющих выполнять все регрессионные тесты автоматически. В некоторых проектах даже используются инструменты для автоматического прогона регрессионных тестов через заданный интервал времени. Обычно это выполняется после каждой удачной компиляции (в небольших проектах) либо каждую ночь или каждую неделю.

2.2.2 Функциональное тестирование

Функциональное тестирование является одним из ключевых видов тестирования, задача которого – установить соответствие разработанного программного обеспечения (ПО) исходным функциональным требованиям компании клиента. То есть проведение функционального тестирования позволяет проверить способность информационной системы в определенных условиях решать задачи, нужные пользователям.

В зависимости от степени доступа к коду системы можно выделить два типа функциональных испытаний:

- тестирование black box (черный ящик) – проведение функционального тестирования без доступа к коду системы,
- тестирование white box (белый ящик) – функциональное тестирование с доступом к коду системы.

Тестирование black box проводится без знания внутренних механизмов работы системы и опирается на внешние проявления ее работы. При этом тестировании проверяется поведение ПО при различных входных данных и внутреннем состоянии систем. В случае тестирования white box создаются тест-кейсы, основанные преимущественно на коде системы ПО. Также

существует расширенный тип black-box тестирования, включающего в себя изучение кода, – так называемый grey box (серый ящик).

2.2.3 Нагрузочное тестирование

Нагрузочное тестирование — подвид тестирования производительности, сбор показателей и определение производительности и времени отклика программно-технической системы или устройства в ответ на внешний запрос с целью установления соответствия требованиям, предъявляемым к данной системе.

Для исследования времени отклика системы на высоких или пиковых нагрузках производится стресс-тестирование, при котором создаваемая на систему нагрузка превышает нормальные сценарии её использования. Не существует чёткой границы между нагрузочным и стресс-тестированием, однако эти понятия не стоит смешивать, так как эти виды тестирования отвечают на разные бизнес-вопросы и используют различную методологию.

В общем случае под нагрузочным тестированием понимается практика моделирования ожидаемого использования приложения с помощью эмуляции работы нескольких пользователей одновременно. Таким образом, подобное тестирование больше всего подходит для многопользовательских систем, чаще — использующих клиент-серверную архитектуру. Однако и другие типы систем ПО могут быть протестированы подобным способом. Например, текстовый или графический редактор можно заставить прочесть очень большой документ; а финансовый пакет — сгенерировать отчёт на основе данных за несколько лет. Наиболее адекватно спроектированный нагрузочный тест даёт более точные результаты.

Основная цель нагрузочного тестирования заключается в том, чтобы, создав определённую ожидаемую в системе нагрузку и, обычно, использовав идентичное программное и аппаратное обеспечение, наблюдать за показателями производительности системы.

В идеальном случае в качестве критериев успешности нагрузочного тестирования выступают требования к производительности системы, которые формулируются и документируются на стадии разработки функциональных требований к системе до начала программирования основных архитектурных решений. Однако часто бывает так, что такие требования не были четко сформулированы или не были сформулированы вовсе. В этом случае первое нагрузочное тестирование будет являться пробным и основываться на разумных предположениях об ожидаемой нагрузке и потреблении аппаратной части ресурсов.

2.2.3 Модульное тестирование

Модульное тестирование – это тип тестирования программного обеспечения, при котором тестируются отдельные модули или компоненты программного обеспечения. Его цель заключается в том, чтобы проверить, что каждая единица программного кода работает должным образом. Данный вид тестирование выполняется разработчиками на этапе кодирования приложения. Модульные тесты изолируют часть кода и проверяют его работоспособность. Единицей для измерения может служить отдельная функция, метод, процедура, модуль или объект.

В моделях разработки SDLC, STLC, V Model модульное тестирование – это первый уровень тестирования, выполняемый перед интеграционным тестированием. Модульное тестирование – это метод тестирования WhiteBox, который обычно выполняется разработчиком. Плюсы модульного тестирования:

- Модульные тесты позволяют исправить ошибки на ранних этапах разработки и снизить затраты;

- Это помогает разработчикам лучше понимать кодовую базу проекта и позволяет им быстрее и проще вносить изменения в продукт;

- Хорошие юнит-тесты служат проектной документацией;

- Модульные тесты помогают с миграцией кода. Просто переносите код и тесты в новый проект и изменяете код, пока тесты не запустятся снова.

2.2.4 Оптимизационное тестирование

Оптимизационное тестирование — устранение узких мест с помощью улучшения алгоритмов, а также использования верных технологий и решений.

2.2.5 Тестирование интерфейса

Тестирование пользовательского интерфейса — это тип тестирования, при котором мы проверяем, работает ли пользовательский интерфейс для веб-приложения нормально или имеет какой-либо дефект, который препятствует поведению пользователя и не соответствует установленным требованиям.

Функциональное тестирование пользовательского интерфейса состоит из пяти фаз:

- анализ требований к пользовательскому интерфейсу;
- разработка тест-требований и тест-планов для проверки пользовательского интерфейса;
- выполнение тестовых примеров и сбор информации о выполнении тестов;
- определение полноты покрытия пользовательского интерфейса требованиями;
- составление отчетов о проблемах в случае несовпадения поведения системы и требований либо в случае отсутствия требований на отдельные интерфейсные элементы.

Все эти фазы точно такие же, как и в случае тестирования любого другого компонента программной системы. Отличия заключаются в трактовке некоторых терминов в применении к пользовательскому интерфейсу и в особенностях автоматизированного сбора информации на каждой фазе.

Требования к пользовательскому интерфейсу могут быть разбиты на две группы:

- требования к внешнему виду пользовательского интерфейса и формам взаимодействия с пользователем;
- требования по доступу к внутренней функциональности системы при помощи пользовательского интерфейса.

Другими словами, первая группа требований описывает взаимодействие подсистемы интерфейса с пользователем, а вторая - с внутренней логикой системы.

К первой группе можно отнести следующие типы требований:

- требования к размещению элементов управления на экранных формах;
- требования к содержанию и оформлению выводимых сообщений;
- требования к форматам ввода;
- требования к реакции системы на ввод пользователя;
- требования к времени отклика на команды пользователя.

2.2.6 Анализ исходного кода

По мере продвижения проекта стоимость устранения дефектов ПО может экспоненциально возрастать. Инструменты статического и динамического анализа помогают предотвратить эти затраты благодаря обнаружению программных ошибок на ранних этапах жизненного цикла ПО. Существует две формы анализа кода:

- динамическая;
- статическая;

Динамический анализ кода — это способ анализа программы непосредственно при ее выполнении. Процесс динамического анализа можно разбить на несколько этапов - подготовка исходных данных, проведение тестового запуска программы и сбор необходимых параметров, анализ полученных данных. При тестовом запуске исполнение программы

может выполняться как на реальном, так и на виртуальном процессоре. Для этого из исходного кода в обязательном порядке должен быть получен исполняемый файл, то есть нельзя таким способом проанализировать код, содержащий ошибки компиляции или сборки. Динамический анализ выполняется с помощью набора данных, которые подаются на вход исследуемой программе. Поэтому эффективность анализа напрямую зависит от качества и количества входных данных для тестирования. Именно от них зависит полнота покрытия кода, которая будет получена по результатам тестирования. Динамическое тестирование наиболее важно в тех областях, где главным критерием является надежность программы, время отклика или потребляемые ресурсы. Это может быть, например, система реального времени, управляющая ответственным участком производства, или сервер базы данных. В таких областях любая допущенная ошибка может оказаться критической.

Динамическое тестирование позволяет убедиться, что продукт работает хорошо или выявляет ошибки, показывая, что программа не работает. Вторая цель тестирования является более продуктивной с точки зрения улучшения качества, так как не позволяет игнорировать недостатки программы. Однако если в ходе тестирования не было выявлено дефектов, то это не значит, что их нет совсем. Даже 100% покрытие кода тестами не означает, что в программе нет ошибок, поскольку динамическое тестирование не может выявить логических ошибок. Так же немаловажным аспектом является то, насколько сами утилиты для тестирования не содержат ошибок.

Статический анализ кода — это процесс выявления ошибок и недочетов в исходном коде программ. Статический анализ можно рассматривать как автоматизированный процесс обзора кода. Обзор кода (code review) – один из самых старых и надежных методов выявления дефектов. Он заключается в совместном внимательном чтении исходного кода и высказывании

рекомендаций по его улучшению. В процессе чтения кода выявляются ошибки или участки кода, которые могут стать ошибочными в будущем. Также считается, что автор кода во время обзора не должен давать объяснений, как работает та или иная часть программы. Алгоритм работы должен быть понятен непосредственно из текста программы и комментариев. Если это условие не выполняется, то код должен быть доработан. Единственный существенный недостаток методологии совместного обзора кода, это крайне высокая цена. Необходимо регулярно собирать нескольких программистов для обзора нового кода или повторного обзора кода после внесения рекомендаций. При этом программисты должны регулярно делать перерывы для отдыха. Если пытаться просматривать сразу большие фрагменты кода, то внимание быстро притупляется, и польза от обзора кода быстро сходит на нет.

2.2.7 Анализ документации

Тестовая документация – это набор документов, создаваемых перед началом процесса тестирования и непосредственно в процессе. Эти документы описывают покрытие тестами и процесс выполнения тестов, в них указываются необходимые для тестирования вещи, приводится основная терминология и т. д.

Если тестирование не документируется, это мешает увидеть полную картину проекта. Без четких целей, пошагового плана по их достижению и указания всех важных условий ожидаемый результат будет неясен. В таких условиях у всех может быть разное понимание общей цели и конечного продукта.

Отсутствие документации может серьезно повлиять на работу тестировщиков. Это особенно верно при работе со сложными продуктами или при часто меняющихся требованиях [4].

2.2.8 Финальное тестирование

Как уже было описано выше финальное тестирование – это последний цикл тестов, перед выпуском продукта на рынок или заказчику.

2.3. Написание тестов

Предположим, существует простая компьютерная игра. В ней вы можете управлять героем во все стороны, прыгать, бегать и т.д.

Задача: необходимо придумать 10 тестов, которые можно произвести над данного вида программой. Тесты оформить в виде таблицы по принципу «Номер теста»-«Суть теста»-«Ожидаемый результат»

2.4. Тестирование разрабатываемого программного обеспечения

Цель работы: найти любое программное обеспечение, и реализовать процесс его тестирования по информации, изученной ранее.

Определить:

- какое оборудование является необходимым для проведения тестов над выбранным ПО;
- каким образом будут проводиться тесты: в ручном режиме или автоматически;
- количество и виды тестов, соответственно изученным ранее.

Что необходимо представить:

- оформленный отчет о ходе тестирования, его результатах и т.д.

3. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Паттерны или шаблоны проектирования — это многократно применяемая архитектурная конструкция, предоставляющая решение общей проблемы проектирования в рамках конкретного контекста и описывающая значимость этого решения.

Паттерн не является законченным образцом проекта, который может быть прямо преобразован в код, скорее это описание или образец для того, как решить задачу, таким образом, чтобы это можно было использовать в различных ситуациях. Объектно-ориентированные шаблоны зачастую показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы или объекты приложения будут использоваться. Алгоритмы не рассматриваются как шаблоны, так как они решают задачи вычисления, а не проектирования [6].

3.1. Порождающие паттерны проектирования как принцип создания новых объектов и семейств объектов

Порождающие шаблоны — шаблоны проектирования, которые имеют дело с процессом создания объектов. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять наследуемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту.

Эти шаблоны оказываются важны, когда система больше зависит от композиции объектов, чем от наследования классов. Получается так, что основной упор делается не на жестком кодировании фиксированного набора поведений, а на определении небольшого набора фундаментальных поведений, с помощью композиции которых можно получать любое число более сложных. Таким образом, для создания объектов с конкретным поведением требуется нечто большее, чем простое инстанцирование класса.

Порождающие шаблоны инкапсулируют знания о конкретных классах, которые применяются в системе, то есть скрывают детали того, как эти классы создаются и стыкуются. Единственная информация об объектах, известная системе, — это их интерфейсы, определенные с помощью абстрактных классов. Следовательно, порождающие шаблоны обеспечивают большую гибкость при решении вопроса о том, что создается, кто это создает, как и когда. Можно собрать систему из «готовых» объектов с самой различной структурой и функциональностью статически (на этапе компиляции) или динамически (во время выполнения).

Иногда допустимо выбирать между тем или иным порождающим шаблоном. Например, есть случаи, когда с пользой для дела можно использовать как прототип, так и абстрактную фабрику. В других ситуациях порождающие шаблоны дополняют друг друга. Так, применяя строитель, можно использовать другие шаблоны для решения вопроса о том, какие компоненты нужно строить, а прототип часто реализуется вместе с одиночкой. Порождающие шаблоны тесно связаны друг с другом, их рассмотрение лучше проводить совместно, чтобы лучше были видны их сходства и различия.

Порождающие шаблоны:

- абстрактная фабрика;
- строитель;
- фабричный метод;
- ленивая инициализация;
- объектный пул;
- прототип;
- одиночка;
- пул одиночек.

Абстрактная фабрика — порождающий шаблон проектирования, предоставляет интерфейс для создания семейств взаимосвязанных или

взаимозависимых объектов, не специфицируя их конкретных классов. Шаблон реализуется созданием абстрактного класса Factory, который представляет собой интерфейс для создания компонентов системы. Затем пишутся классы, реализующие этот интерфейс.

Строитель — порождающий шаблон проектирования предоставляет способ создания составного объекта.

Фабричный метод — порождающий шаблон проектирования, предоставляющий подклассам интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Иными словами, данный шаблон делегирует создание объектов наследникам родительского класса. Это позволяет использовать в коде программы не конкретные классы, а манипулировать абстрактными объектами на более высоком уровне.

Отложенная инициализация — приём в программировании, когда некоторая ресурсоёмкая операция выполняется непосредственно перед тем, как будет использован её результат.

Объектный пул — порождающий шаблон проектирования, набор инициализированных и готовых к использованию объектов. Когда системе требуется объект, он не создаётся, а берётся из пула. Когда объект больше не нужен, он не уничтожается, а возвращается в пул.

Прототип - задаёт виды создаваемых объектов с помощью экземпляра-прототипа и создаёт новые объекты путём копирования этого прототипа. Он позволяет уйти от реализации и позволяет следовать принципу «программирование через интерфейсы». В качестве возвращающего типа указывается интерфейс/абстрактный класс на вершине иерархии, а классы-наследники могут подставить туда наследника, реализующего этот тип.

Одиночка — порождающий шаблон проектирования, гарантирующий, что в однопоточном приложении будет единственный экземпляр некоторого класса, и предоставляющий глобальную точку доступа к этому экземпляру.

Мультитон — порождающий шаблон проектирования, который обобщает шаблон "Одиночка". В то время, как "Одиночка" разрешает создание лишь одного экземпляра класса, мультитон позволяет создавать несколько экземпляров, которые управляются через ассоциативный массив. Создаётся лишь один экземпляр для каждого из ключей ассоциативного массива, что позволяет контролировать уникальность объекта по какому-либо признаку.

3.2. Структурные паттерны проектирования как метод реализации иерархий классов

Структурные шаблоны — шаблоны проектирования, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры [7]. Структурные шаблоны в основном связаны с композицией объектов, другими словами, с тем, как сущности могут использовать друг друга. Ещё одним объяснением было бы то, что они помогают ответить на вопрос «Как создать программный компонент?».

Список структурных шаблонов проектирования:

- адаптер;
- мост;
- компоновщик;
- декоратор;
- фасад;
- приспособленец;
- заместитель.

Адаптер — структурный шаблон проектирования, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс.

Мост — структурный шаблон проектирования, используемый в проектировании программного обеспечения чтобы разделять абстракцию и реализацию так, чтобы они могли изменяться независимо. Шаблон мост

использует инкапсуляцию, агрегирование и может использовать наследование для того, чтобы разделить ответственность между классами.

Компоновщик — структурный шаблон проектирования, объединяющий объекты в древовидную структуру для представления иерархии от частного к целому. Компоновщик позволяет клиентам обращаться к отдельным объектам и к группам объектов одинаково. Паттерн определяет иерархию классов, которые одновременно могут состоять из примитивных и сложных объектов, упрощает архитектуру клиента, делает процесс добавления новых видов объекта более простым.

Декоратор — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту. Шаблон декоратор предоставляет гибкую альтернативу практике создания подклассов с целью расширения функциональности.

Фасад — структурный шаблон проектирования, позволяющий скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.

Приспособленец — структурный шаблон проектирования, при котором объект, представляющий себя как уникальный экземпляр в разных местах программы, по факту не является таковым.

Заместитель — структурный шаблон проектирования, который предоставляет объект, который контролирует доступ к другому объекту, перехватывая все вызовы (выполняет функцию контейнера).

3.3. Поведенческие паттерны проектирования как метод реализации иерархий классов

Поведенческие шаблоны — шаблоны проектирования, определяющие алгоритмы и способы реализации взаимодействия различных объектов и классов. Поведенческие шаблоны связаны с распределением обязанностей между объектами. Их отличие от структурных шаблонов заключается в том,

что они не просто описывают структуру, но также описывают шаблоны для передачи сообщений/связи между ними. Или, другими словами, они помогают ответить на вопрос «Как запустить поведение в программном компоненте?»

Поведенческие шаблоны:

- цепочка обязанностей;
- команда;
- итератор;
- посредник;
- хранитель;
- наблюдатель;
- посетитель;
- стратегия;
- состояние;
- шаблонный метод.

Цепочка обязанностей — поведенческий шаблон проектирования, предназначенный для организации в системе уровней ответственности.

Команда — поведенческий шаблон проектирования, используемый при объектно-ориентированном программировании, представляющий действие. Объект команды заключает в себе само действие и его параметры.

Итератор — поведенческий шаблон проектирования. Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из агрегированных объектов.

Посредник — поведенческий шаблон проектирования, обеспечивающий взаимодействие множества объектов, формируя при этом слабую связанность, и избавляя объекты, от необходимости явно ссылаться друг на друга.

Хранитель — поведенческий шаблон проектирования, позволяющий, не нарушая инкапсуляцию, зафиксировать и сохранить внутреннее состояние объекта так, чтобы позднее восстановить его в этом состоянии.

Наблюдатель — поведенческий шаблон проектирования, также известен как «подчинённые». Создает механизм у класса, который позволяет получать экземпляру объекта этого класса оповещения от других объектов об изменении их состояния, тем самым наблюдая за ними.

Посетитель — поведенческий шаблон проектирования, описывающий операцию, которая выполняется над объектами других классов. При изменении visitor нет необходимости изменять обслуживаемые классы.

Стратегия — поведенческий шаблон проектирования, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Это позволяет выбирать алгоритм путём определения соответствующего класса. Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют.

Состояние — поведенческий шаблон проектирования. Используется в тех случаях, когда во время выполнения программы объект должен менять своё поведение в зависимости от своего состояния.

Шаблонный метод — поведенческий шаблон проектирования, определяющий основу алгоритма и позволяющий наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.

3.4. Внедрение паттернов проектирования в разработку программного обеспечения

Цель – разработать программу, использующую один из описанных выше шаблонов по вариантам.

Вариант 1: Один из порождающих паттернов.

Вариант 2: Один из структурных паттернов.

Вариант 3: Один из поведенческих паттернов.

Представить работу одного или нескольких паттернов в виде текстового отчета.

4. UML-ДИАГРАММЫ КАК ЯЗЫК ГРАФИЧЕСКОГО ОПИСАНИЯ В ОБЛАСТИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

UML — язык графического описания для объектного моделирования в области разработки программного обеспечения, для моделирования бизнес-процессов, системного проектирования и отображения организационных структур.

UML является языком широкого профиля, это — открытый стандарт, использующий графические обозначения для создания абстрактной модели системы, называемой UML-моделью. UML был создан для определения, визуализации, проектирования и документирования, в основном, программных систем. UML не является языком программирования, но на основании UML-моделей возможна генерация кода.

4.1. Структурные диаграммы

Структурные диаграммы показывают статическую структуру системы и ее частей на разных уровнях абстракции и реализации, а также их взаимосвязь. Элементы в структурной диаграмме представляют значимые понятия системы и могут включать в себя абстрактные, реальные концепции и концепции реализации. Существует семь типов структурных диаграмм:

- диаграмма составной структуры;
- диаграмма развертывания;
- диаграмма пакетов;
- диаграмма профилей;
- диаграмма классов;
- диаграмма объектов;
- диаграмма компонентов.

Диаграмма составной структуры аналогична диаграмме классов и является своего рода диаграммой компонентов, используемой в основном при моделировании системы на микроуровне, но она изображает отдельные

части вместо целых классов. Это тип статической структурной диаграммы, которая показывает внутреннюю структуру класса и взаимодействия, которые эта структура делает возможными. Эта диаграмма может включать внутренние части, порты, через которые части взаимодействуют друг с другом или через которые экземпляры класса взаимодействуют с частями и с внешним миром, и соединители между частями или портами. Составная структура — это набор взаимосвязанных элементов, которые взаимодействуют во время выполнения для достижения какой-либо цели. Каждый элемент имеет определенную роль в сотрудничестве.

Диаграмма развертывания помогает моделировать физический аспект объектно-ориентированной программной системы. Это структурная схема, которая показывает архитектуру системы, как развертывание (дистрибуции) программных артефактов. Артефакты представляют собой конкретные элементы в физическом мире, которые являются результатом процесса разработки. Диаграмма моделирует конфигурацию времени выполнения в статическом представлении и визуализирует распределение артефактов в приложении. В большинстве случаев это включает в себя моделирование конфигураций оборудования вместе с компонентами программного обеспечения, на которых они размещены.

Диаграмма пакетов — это структурная схема UML, которая показывает пакеты и зависимости между ними. Она позволяет отображать различные виды системы, например, легко смоделировать многоуровневое приложение.

Диаграмма профилей позволяет нам создавать специфичные для домена и платформы стереотипы и определять отношения между ними. Мы можем создавать стереотипы, рисуя формы стереотипов и связывая их с композицией или обобщением через интерфейс, ориентированный на ресурсы. Мы также можем определять и визуализировать значения стереотипов.

Диаграмма классов — это центральная методика моделирования, которая используется практически во всех объектно-ориентированных методах. Эта диаграмма описывает типы объектов в системе и различные виды статических отношений, которые существуют между ними. Три наиболее важных типа отношений в диаграммах классов (на самом деле их больше), это:

- ассоциация, которая представляет отношения между экземплярами типов, к примеру, человек работает на компанию, у компании есть несколько офисов.
- наследование, которое имеет непосредственное соответствие наследованию в объектно-ориентированном дизайне.
- агрегация, которая представляет из себя форму композиции объектов в объектно-ориентированном дизайне.

Статическая диаграмма объектов является экземпляром диаграммы класса; она показывает снимок подробного состояния системы в определенный момент времени. Разница в том, что диаграмма классов представляет собой абстрактную модель, состоящую из классов и их отношений. Тем не менее, диаграмма объекта представляет собой экземпляр в конкретный момент, который имеет конкретный характер. Использование диаграмм объектов довольно ограничено, а именно — чтобы показать примеры структуры данных.

На языке унифицированного моделирования диаграмма компонентов показывает, как компоненты соединяются вместе для формирования более крупных компонентов или программных систем. Она иллюстрирует архитектуры компонентов программного обеспечения и зависимости между ними. Эти программные компоненты включают в себя компоненты времени выполнения, исполняемые компоненты, а также компоненты исходного кода.

4.2. Диаграммы поведения

Диаграммы поведения показывают динамическое поведение объектов в системе, которое можно описать, как серию изменений в системе с течением времени. К диаграммам поведения относятся:

- диаграмма деятельности;
- диаграмма прецедентов;
- диаграмма состояний;
- диаграмма последовательности;
- диаграмма коммуникаций;
- диаграмма обзора взаимодействия;
- временная диаграмма.

Диаграммы деятельности представляют собой графическое представление рабочих процессов поэтапных действий и действий с поддержкой выбора, итерации и параллелизма. Они описывают поток управления целевой системой, такой как исследование сложных бизнес-правил и операций, а также описание прецедентов и бизнес-процессов. В UML диаграммы деятельности предназначены для моделирования как вычислительных, так и организационных процессов.

Диаграмма прецедентов описывает функциональные требования системы с точки зрения прецедентов. По сути дела, это модель предполагаемой функциональности системы (прецедентов) и ее среды (актеров). Прецеденты позволяют связать то, что нам нужно от системы с тем, как система удовлетворяет эти потребности.

Диаграмма состояний — это тип диаграммы, используемый в UML для описания поведения систем, который основан на концепции диаграмм состояний Дэвида Харела. Диаграммы состояний отображают разрешенные состояния и переходы, а также события, которые влияют на эти переходы. Она помогает визуализировать весь жизненный цикл объектов и, таким образом, помогает лучше понять системы, основанные на состоянии.

Диаграмма последовательности моделирует взаимодействие объектов на основе временной последовательности. Она показывает, как одни объекты взаимодействуют с другими в конкретном прецеденте.

Как и диаграмма последовательности, диаграмма коммуникации также используется для моделирования динамического поведения прецедента. Если сравнивать с Диаграммой последовательности, Диаграмма коммуникации больше сфокусирована на показе взаимодействия объектов, а не временной последовательности. На самом деле, диаграмма коммуникации и диаграмма последовательности семантически эквивалентны и могут перетекать одна в другую.

Диаграмма обзора взаимодействий фокусируется на обзоре потока управления взаимодействиями. Это вариант Диаграммы деятельности, где узлами являются взаимодействия или события взаимодействия. Диаграмма обзора взаимодействий описывает взаимодействия, в которых сообщения и линии жизни скрыты. Мы можем связать «реальные» диаграммы и добиться высокой степени навигации между диаграммами внутри диаграммы обзора взаимодействия.

Временная диаграмма показывает поведение объекта в данный период времени. По сути — это особая форма диаграммы последовательности и различия между ними состоят в том, что оси меняются местами так, что время увеличивается слева направо, а линии жизни отображаются в отдельных отсеках, расположенных вертикально [8].

4.3. Диаграммы взаимодействия

Диаграмма взаимодействия — это диаграмма, на которой представлено взаимодействие, состоящее из множества объектов и отношений между ними, включая и сообщения, которыми они обмениваются. Этот термин применяется к видам диаграмм с акцентом на взаимодействии объектов (диаграммах кооперации, последовательности и деятельности).

Целью диаграммы взаимодействия является:

- фиксирование поведения динамической системы;
- описание потока сообщений в системе;
- описание структурной организации объектов;
- описание взаимодействия между объектами;

Все остальные свойства, относящиеся к диаграмме взаимодействия, являются идентичными описанным выше диаграммам так, как диаграмма взаимодействия — это частный случай диаграммы.

5. САМОСТОЯТЕЛЬНАЯ РАБОТА

Данный предмет предполагает самостоятельную работу студента в форме курсовой работы. Курсовая работа по настоящей дисциплине представляет собой законченную работу, включающую в себя написание технического задания, разработку программного обеспечения (в соответствии с заданием), тестирование на отказоустойчивость и документа инициации проекта.

Примерные темы курсовых работ:

- Распознавание элементарных образов с помощью оператора Кэнни;
- Мессенджер с асинхронным шифрованием данных;
- Создание клиент-серверного приложения для синхронизации адресной книги телефона (на базе Android Java) с компьютером;

Данная работа позволяет оценить умения учащихся решать практические задачи проектирования архитектуры сложных систем, оценить приобретенные навыки архитектуризации.

ЗАКЛЮЧЕНИЕ

В пособии изложены базовые понятия о задачах, методах и средствах программной инженерии как деятельности, нацеленной на создание программных продуктов, отвечающих потребностям заказчиков, с соблюдением плановых сроков и бюджета разработки. Также изложены методы тестирования программного обеспечения, рассмотрены некоторые паттерны программирования. Приведены виды графической нотации, в виде UML-диаграмм, используемые для лучшего понимания архитектуры разрабатываемого программного обеспечения.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА ПО КУРСУ

1. Доррер, А. Г. Управление ИТ-проектами: учебное пособие / А. Г. Доррер, М. Г. Доррер, А. А. Попов. — Красноярск: СибГУ им. академика М. Ф. Решетнёва, 2019. — 174 с.
2. Маран, М. М. Программная инженерия: учебное пособие для вузов / М. М. Маран. — 3-е изд., стер. — Санкт-Петербург: Лань, 2022. — 196 с. — ISBN 978-5-8114-9323-4.
3. Косяков А. Системная инженерия. Принципы и практика / А. Косяков, У. Свит, С. Сеймур, С. Бимер. - М.: ДМК Пресс, 2017. - 624 с.
4. Вдовин В.М., Суркова Л.Е., Валентинов В.А. Теория систем и системный анализ. Учебник. М.: Дашков и Ко, 2012. - 639 с.
5. Гуськова, О. И. Объектно ориентированное программирование в Java: учебное пособие / О. И. Гуськова. — Москва: МПГУ, 2018. — 240 с. — ISBN 978-5-4263-0648-6.
6. Нобак, М. Принципы разработки программных пакетов: руководство / М. Нобак ; перевод с английского Д. А. Беликова. — Москва: ДМК Пресс, 2020. — 274 с. — ISBN 978-5-97060-793-0.
7. Моисеев, Д.А. МЕТОДОЛОГИЯ И ПРОЦЕСС РУЧНОГО ТЕСТИРОВАНИЯ / Д.А. Моисеев // Надежность и качество сложных систем. — 2017. — № 3. — С. 107-112. — ISSN 2307-4205.
8. Зубкова, Т. М. Технология разработки программного обеспечения: учебное пособие / Т. М. Зубкова. — Санкт-Петербург: Лань, 2019. — 324 с. — ISBN 978-5-8114-3842-6.
9. Барков, И. А. Объектно-ориентированное программирование: учебник / И. А. Барков. — Санкт-Петербург: Лань, 2019. — 700 с. — ISBN 978-5-8114-3586-9.

Учебное издание

Илья Викторович **Степанченко**
Сергей Игоревич **Щербин**
Ростислав Олегович **Фокин**

ПРОГРАММНАЯ ИНЖЕНЕРИЯ

Учебное пособие

На правах рукописи

Редактор *Л. Н. Рыжих*

Темплан 2021 г. (учебники и учебные пособия). Поз. № 19.
Подписано в печать 00.00.2021. Формат 60x84 1/16. Бумага газетная.
Гарнитура Times. Печать офсетная. Усл. печ. л. 4,0. Уч.-изд. л. 5,05.
Тираж 100 экз. Заказ

Волгоградский государственный технический университет.
400005, г. Волгоград, просп. В. И. Ленина, 28, корп. 1.

Отпечатано в типографии ИУНЛ ВолгГТУ.
400005, г. Волгоград, просп. В. И. Ленина, 28, корп. 7.