Министерство науки и высшего образования

Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Рязанский государственный радиотехнический университет

имени В.Ф.Уткина»

УТВЕРЖДАЮ

Заведующий кафедрой ЭВМ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Б.В. Костров

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_ г.

**ОТЧЕТ**

**О НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ**

Исследование особенностей применения принципа инверсии управления при проектировании и разработке информационных систем

обучающегося 1 курса, 340М учебной группы

Маринина Ивана Дмитриевича

Направление подготовки: 02.04.03, Математическое обеспечение и администрирование информационных систем

Образовательная программа: Бизнес-анализ и проектирование информационных систем

Кафедра: «Электронные вычислительные машины»

Сроки прохождения НИР:

с «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_ г.

по «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_ г.

Рязань 2023 г.

**Результаты выполнения НИР:**

1. Отчет о выполнении задания

|  |  |  |
| --- | --- | --- |
| **№ п\п** | **Планируемые формы работы (примерный план)** | **Отметка о выполнении** |
| 1 | Утверждение темы НИР | Выполнено |
| 2 | Анализ литературы по теме НИР | Выполнено |
| 2.1 | Подготовка списка литературы по теме «Инверсия управления» | Выполнено |
| 2.2 | Изучение актуальных подходов к инверсии управления | Выполнено |
| 2.3 | Изучение возможности применения инверсии управления при проектировании и разработке ПО | Выполнено |
| 3 | Подготовка материалов для публикации | Выполнено |

1. Описание проделанной работы согласно заданию на НИР
2. Список использованных источников

Руководитель\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ Устюков Д.И. /

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_ г.

Обучающийся\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ Маринин И.Д, /

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_ г.

Содержание

[Введение 4](#_Toc154502850)

[1 Объектно-ориентированный подход к разработке ПО 6](#_Toc154502851)

[1.1 Интерфейсы 7](#_Toc154502852)

[1.2 Повторное использование компонентов при разработке ПО 10](#_Toc154502853)

[1.3 Тестирование 12](#_Toc154502854)

[1.4 Принципы SOLID 20](#_Toc154502855)

[1.5 Связанность кода 22](#_Toc154502856)

[1.6 Вывод по главе 1 23](#_Toc154502857)

[2 Инверсия управления 25](#_Toc154502858)

[2.1 Шаблон «Фабрика» 26](#_Toc154502859)

[2.2 Локатор служб 29](#_Toc154502860)

[2.3 Внедрение зависимостей 31](#_Toc154502861)

[2.3.1 Внедрение через конструктор 32](#_Toc154502862)

[2.3.2 Внедрение через set-метод 33](#_Toc154502863)

[2.3.3 Внедрение через интерфейс 34](#_Toc154502864)

[2.3.4 Вывод 35](#_Toc154502865)

[2.4 Вывод по главе 2 36](#_Toc154502866)

[3 Применение инверсии управления при проектировании и разработке информационных систем 37](#_Toc154502867)

[Список использованной литературы 39](#_Toc154502868)

# Введение

Подходы к разработке программного обеспечения меняются и совершенствуются вместе с увеличением мощностей аппаратного обеспечения и требований к разрабатываемому ПО. Кроме того, эти изменения продиктованы и тем, что за все время разработка ПО сместилась от решения научных к решению прикладных задач. Значительно повысилась степень автоматизации производства, экономики и сферы услуг.

Изменился и подход к построению жизненного цикла при разработке ПО. Вместе с приходом гибких методологий, фокус сдвинулся с разработки решений на их обслуживание и поддержку. Современный мир не стоит на месте и выгодополучатели разработки ПО (бизнес или производство) требуют от команд разработки соответствия этому движению. Сокращается время между выпусками новых версий и как следствие повышается скорость и темп разработки нового функционала.

Правило 60/60, введенное Дэвидом Вудом в книге «97 Things Every Project Manager Should Know», гласит, что на поддержку и обслуживание ПО приходится в среднем 60% затрат ресурсов, в то время как на разработку - всего 40%. Вводится новая метрика «поддерживаемость» (maintainability), которая отражает то, насколько дешево с точки зрения расхода ресурсов внести требуемые изменения в функционал программного обеспечения.

Как следствие, изменились тенденции и используемые технологии. На первое место вышли языки программирования высокого уровня. Они позволили перейти на необходимый уровень абстракции и дали возможность программистам сосредоточиться на решаемых задачах, а не на отлавливании ошибок, связанных, например, с утечками памяти.

Несмотря на этот переход, уровень абстракции, обеспечиваемый современными языками программирования все еще недостаточен, и это приводит к появлению дополнительных инструментов и концепций.

В данной научно-исследовательской работе будет рассмотрена одна из таких концепций – инверсия управления, а также рассмотрена возможность использования данной концепции при проектировании и разработке информационных систем.

# 1 Объектно-ориентированный подход к разработке ПО

Объектно-ориентированное программирование появилось как результат эволюции процедурного программирования. Основоположником объектно-ориентированного программирования является Алан Кей.

Идея современного объектно-ориентированного программирования заключается в первую очередь в представлении компьютерной программы как набора взаимодействующих между собой объектов. Это отличается от концепции процедурного программирования, основа которого заключается в использовании подпрограмм (процедур или функций) для модификации состояния памяти компьютера. Несомненно, объектно-ориентированный подход не вносит кардинальных изменений в процесс работы программы, но при этом вводит дополнительный слой абстракции, который усложняет программу для машины, но при этом упрощает её разработку и отладку для человека.

ООП не оперирует структурами данных напрямую – примитивные структуры данных (переменные, массивы) не могут существовать сами по себе. Они спрятаны внутри высокоуровневых структур - классов, которые описывают как свойства объектов, в виде наборов каких-то структур данных, так и поведение. В терминологии ООП отсутствует такое понятие, как «процедура» или «функция» – они больше не могут существовать сами по себе и заменены на объединяющее понятие «метод». Набор методов определяет поведение объекта.

В конечном счёте ООП позволяет программисту разрабатывать более сложные программы, используя не только алгоритмы, но и объекты, которые взаимодействуют между собой.

ООП базируется на четырех основных принципах, каждый из которых дополняет предыдущий:

1. Абстракция. Заключается в моделировании требуемых атрибутов и взаимодействий сущностей и отказе от незначительных с точки зрения функционирования, что в конечном счете позволяет построить некую абстрактную модель классов системы.
2. Инкапсуляция. Заключается в объединении свойств сущности и методов для работы с ними в составе класса. Иными словами, инкапсуляция обеспечивает сокрытие данных и предоставление некоего контракта для работы с ними. Это позволяет поведению объекта оставаться устойчивым и независимым от изменений внешней среды, в которой он используется.
3. Наследование. Заключается в способности классов наследовать поведение и свойства других классов, называемых «предками». Это позволяет выстраивать иерархию классов и обеспечивает повторное использование кода.
4. Полиморфизм. Заключается в способности объекта принимать различные формы, в зависимости от контекста, в котором он используется.

## 1.1 Интерфейсы

Объектно-ориентированное программирование вводит новую структуру – интерфейс. Интерфейс – синтаксическая структура, описывающая отношения с объектами, которые объединены общим поведением. Иными словами, интерфейс – это протокол взаимодействия с объектом.

Интерфейс не имеет аналогов в структурном и процедурном программировании. Его основная цель – отделять поведение класса от его реализации. Это повышает уровень абстракции. Достаточно выставить потребителю компонента интерфейс взаимодействия с ним и не задумываться о его реализации.

Интерфейсы способствуют повторному использованию кода и являются первым серьезным шагом на пути к инверсии контроля.

Рассмотрим следующий пример.

Имеется некий класс UserRecord, который отражает запись о каком-то пользователе.

**Запись UserRecord**

|  |
| --- |
| public record UserRecord(  Long id,  String name ) { } |

Необходимо каким-то образом хранить эти записи. За хранение записей будет отвечать репозиторий – некое хранилище данных.

Хранилища данных имеют совершенно различную природу: данные могут храниться в памяти, в базе данных, внутри файлов на физических носителях информации и т.д.

Вместо того, чтобы использовать какую-то конкретную реализацию, определим интерфейс, описывающий протокол взаимодействия с репозиторием.

**Интерфейс UserRepository**

|  |
| --- |
| public interface UserRepository {    UserRecord getUserById(Long id);    List<UserRecord> getUsers();   void saveUser(UserRecord user); } |

Интерфейс не содержит никаких деталей реализаций – лишь описывает набор методов, который гарантировано позволяет работать с репозиторием.

Потребителю сервиса не нужно знать природу или алгоритм работы репозитория – достаточно, чтобы тот лишь реализовывал нужный контракт, то есть принимал и возвращал данные строго определенных в интерфейсе типов.

Приведем возможную реализацию репозитория, хранящего данные в оперативной памяти.

**Класс InMemUserRepositoryImpl**

|  |
| --- |
| public class InMemUserRepositoryImpl implements UserRepository {   Map<Long, UserRecord> users = new HashMap<>();  @Override  public UserRecord getUserById(Long id) {  return users.get(id);  }   @Override  public List<UserRecord> getUsers() {  return users.values().stream().toList();  }   @Override  public void saveUser(UserRecord user) {  users.put(user.id(), user);  } } |

Данный класс является реализацией интерфейса UserRepository. В языке Java об этом сигнализирует ключевое слово **implements.** Реализации методов интерфейса аннотированы как **@Override**. Эта аннотация не является обязательной, но позволяет компилятору проверить, что данный метод действительно переопределяет один из методов, описанных в интерфейсе.

Таким образом, потребитель данного интерфейса может использовать его в абстракции от конкретной реализации.

Например, так может выглядеть получение имени пользователя, который был ранее сохранен в репозиторий.

|  |
| --- |
| … public String getNameOfUser(Long id, UserRepository repository) {  var user = repository.getUserById(id);  return user.name(); }  … |

Данный пример иллюстрирует возможность использования интерфейса для вызова методов объекта без какой-либо информации о конкретной реализации данных методов.

Важно отметить, что интерфейс описывает лишь *протокол* взаимодействия с объектом класса. Обычно, дополнительно оговаривается, какое поведение должна иметь реализация того или иного интерфейса. Было бы странно, если бы возможная гипотетическая реализация UserRepository, основанная, например, на базе данных, возвращала не n-го пользователя при вызове **getUserById(n)**, а, например, n+100-го.

## 1.2 Повторное использование компонентов при разработке ПО

Одной из причин, по которой объектно-ориентированное программирование стало популярным, является увеличение важности повторного использования компонентов при разработке ПО. Создание новых систем является затратным, а их поддержка ещё более дорогостоящей. Очевидно, что одним из способов повторного использования программы является её усовершенствование, поэтому поддержка программного обеспечения является особым случаем повторного использования. Оба случая требуют от программистов понимания и изменения написанного другими программного обеспечения.

Эволюционные жизненные циклы являются правилом, а не исключением. Поддержку программного обеспечения можно классифицировать как коррективную, адаптивную и совершенствующую. Коррективная поддержка — это процесс диагностики и исправления ошибок. Адаптивная поддержка включает в себя действия, необходимые для правильной интеграции программного продукта с новым оборудованием, периферийными устройствами и т.д. Совершенствование требуется, когда программный продукт успешен. По мере использования продукта возникает давление на разработчиков для усовершенствования и расширения его функциональности. Поскольку 60% деятельности по поддержке — это совершенствование, эволюционная фаза является важной частью жизненного цикла успешного программного продукта.

Языки объектно-ориентированного программирования поощряют повторное использование программного обеспечения по ряду причин. Определения классов обеспечивают модульность и сокрытие информации. Позднее связывание вызовов процедур означает, что объекты требуют меньше информации о друг друге, поэтому объектам нужно только правильное взаимодействие (протокол). Полиморфную процедуру легче повторно использовать, чем неполиморфную, потому что она будет работать с широким спектром аргументов. Наследование классов позволяет переиспользовать класс в измененной форме, создавая дочерние классы. Наследование классов также помогает формировать наборы стандартных протоколов, важны для повторного использования.

Эти возможности также полезны при поддержке. Модульность упрощает понимание последствий изменений в программе. Полиморфизм сокращает количество методов и, следовательно, размер программы, которую должен понять сопровождающий. Наследование классов позволяет построить новую версию программы, не затрагивая старую.

Многие приемы повторного использования программного обеспечения, написанного на традиционных языках, аналогичны объектно-ориентированным приемам. Например, программные оболочки (скелеты) полностью замещаются абстрактными классами. Копирование и редактирование программы замещается наследованием класса и переопределением некоторых его методов. Объектно-ориентированные приемы имеют преимущество в том, что новому классу передаются только отличия между ним и старым, что упрощает определение различий между новой и старой программой. Таким образом, набор подклассов сохраняет историю изменений, внесенных подклассами в родительский класс. Условное выполнение программы с помощью флаговых параметров или тестов вариантов практически всегда может быть заменено созданием подкласса для каждого варианта и переопределением методов, выполняющих тесты.

Повторное использование программного обеспечения не происходит случайно, даже в языках объектно-ориентированного программирования. Архитекторы систем должны планировать повторное использование старых компонентов и искать новые компоненты для повторного использования.

Подобным принципом должны руководствоваться и разработчики инструментов и библиотек для решения вспомогательных задач, например, работы с базой данных. Со стороны пользователей инструментов и библиотек имеется необходимость расширения и дополнения возможностей, так как спектр решаемых задач с разными нюансами безграничен.

## 1.3 Тестирование

Одним из важных аспектов при разработке ПО становится его тестирование. Так или иначе через этап тестирования проходит любая программа. В зависимости от различных факторов, в том числе предлагаемой функциональности, размеров программы, тестированием может заниматься как сам программист, так и команда тестирования.

Существуют различные виды тестирования ПО, кратко рассмотрим некоторые из них:

1. Модульное тестирование: это тестирование отдельных модулей или компонентов программного обеспечения. Здесь проверяется корректность работы функций, процедур или классов в изоляции от остальных частей системы. Модульные тесты обычно создаются программистами и могут быть выполнены автоматически с использованием инструментов для модульного тестирования.

2. Интеграционное тестирование: это тестирование совместной работы различных модулей или компонентов программного обеспечения. Здесь проверяется взаимодействие между модулями и обмен данными между ними. Целью интеграционного тестирования является обнаружение ошибок, которые могут возникнуть при объединении различных компонентов системы.

3. Системное тестирование: это тестирование функциональности и производительности всей системы в целом. Здесь проверяется соответствие программного обеспечения требованиям и ожиданиям пользователей. Системное тестирование может включать тестирование функций, пользовательского интерфейса, безопасности, нагрузочное тестирование и другие виды тестов, необходимые для удостоверения работоспособности системы.

4. Приемочное тестирование: это тестирование проводится для проверки программного обеспечения на соответствие требованиям заказчика или конечных пользователей. Здесь используются реальные данные и сценарии использования, чтобы удостовериться, что система соответствует ожиданиям пользователей и выполняет указанные требования.

5. Регрессионное тестирование: это тестирование выполняется после внесения изменений или доработок в программное обеспечение. Целью регрессионного тестирования является проверка, не повлияла ли модификация на уже ранее работающие функции и компоненты системы.

При тестировании ПО используются различные подходы:

1. Черный ящик (Black-box testing): этот подход к тестированию основан на внешнем поведении программы. Тестировщик не имеет знания о внутренней структуре или реализации программы и проверяет ее функциональность по спецификации требований. Входные данные и ожидаемые выходные данные используются для проведения тестов.

2. Белый ящик (White-box testing): это тестирование, при котором тестировщик имеет полное знание о внутренней структуре и коде программы. Тесты создаются на основе анализа исходного кода для обнаружения ошибок в логике программы, управлении данными, пути выполнения и т.д. В этом подходе используются такие методы, как тестирование уровней покрытия кода, тестирование граничных случаев, тестирование пути выполнения и другие.

3. Серый ящик (Gray-box testing): этот подход сочетает черный ящик и белый ящик, используя как внешнее, так и внутреннее знание программы при проведении тестирования. Тестировщик имеет частичное знание о внутренней структуре программы, что позволяет уделять больше внимания ключевым областям тестирования.

4. Автоматизированное тестирование: это использование специальных инструментов или скриптов для автоматизации выполнения тестовых сценариев. Автоматизация тестирования облегчает повторное выполнение тестов и повышает эффективность процесса тестирования.

5. Ручное тестирование: это тестирование, которое выполняется вручную людьми. Ручные тесты могут быть полезными для проверки пользовательского интерфейса, взаимодействия с системой и других аспектов, требующих внимания и интуиции человека.

Современные тенденции таковы, что некоторые виды тестирования появляются на самых ранних этапах разработки ПО - ещё до того, как основная функциональность системы готова к работе. Это позволяет найти и устранить ошибки заранее и в конечном счете снижает стоимость целевого продукта. На рисунке 1 представлен график зависимости стоимости исправления дефекта от времени его исправления.

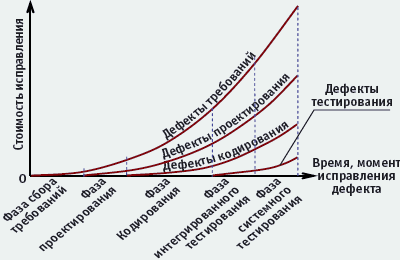


Рисунок 1 – График зависимости стоимости от времени

На практике программисты часто прибегают к автоматизированному тестированию своего же кода. В зависимости от решаемой задачи, они сами разрабатывают тесты, которые проверяют, насколько верно функционирует написанная программа.

Существует отдельный подход, называемый «разработка через тестирование» (Test driven development, TDD), который заключается в написании программы для заранее подготовленных тестов, которые покрывают различные варианты входных и выходных значений. Однако, и подход с написанием таких тестов для уже готовой программы тоже имеет место быть.

В конечном счете, разработка таких тестов позволяет получить два основных преимущества: отсутствие необходимости ручного прогона всех комбинаций входных и выходных данных и упрощение регрессионного тестирования.

Первое преимущество исходит из того, что в процессе разработки программы или отдельного её модуля, разработчику нужно постоянно проверять корректную работоспособность. Количество различных комбинаций входных и выходных данных коррелирует с количеством в структуре программы, и чаще всего проверить эти комбинации вручную попросту невозможно. Автоматизированные тесты решают эту проблему и избавляют программиста от необходимости тестировать написанный ими код вручную.

Второе преимущество является следствием первого. При внесении каких-то изменений в работу программы, необходимо также выполнять регрессионное тестирование, иными словами, проверять, что эти изменения не нарушают корректность работы других функций. Регрессионное тестирование является довольно сложным процессом и автоматизированные тесты сильно упрощают его. Программисту достаточно запустить тесты, которые были подготовлены при разработке ранее, чтобы убедиться, что его изменения не затрагивают другие функции в системе.

Обратимся к такой концепции как пирамида тестирования.

Пирамида тестирования представляет собой концепцию, которая используется для организации и приоритезации различных типов тестового покрытия в рамках проектов по разработке программного обеспечения. В пирамиде тестирования различные виды тестов располагаются по уровням, где наиболее многочисленными и основными являются модульные тесты, а менее многочисленными, но более высокого уровня, - интеграционные и системные тесты. Вот более подробное описание пирамиды тестирования:

1. Модульные тесты: это основной и самый широко распространенный вид тестов, который выполняется программистами на ранней стадии разработки. Модульные тесты проверяют правильность работы отдельных модулей, функций или классов программного обеспечения. Они обычно являются автоматизированными тестами и проверяют логику и функциональность модуля, а также позволяют выявлять ошибки на ранних этапах разработки.

2. Интеграционные тесты: после успешного прохождения модульных тестов происходит интеграция модулей и компонентов вместе для проверки их взаимодействия и работоспособности вместе. Интеграционные тесты проводятся для обнаружения ошибок, связанных с передачей данных между модулями, взаимодействием и соответствием интерфейсов. Здесь также могут использоваться автоматизированные тесты, но иногда требуется ручное тестирование.

3. Системные тесты: на этом уровне проводится тестирование всей системы в целом для проверки ее функциональности, производительности, безопасности и соответствия требованиям заказчика. Целью системного тестирования является проверка работы системы в реальных условиях использования. Тестирование проводится на основе функциональных и нефункциональных требований и может включать автоматизированные и ручные тесты.

Пирамида тестирования предлагает следующий подход:

- Большая часть тестирования (около 70-80%) должна состоять из модульных тестов, которые выполняются на уровне кода и обеспечивают быстрое обнаружение и исправление ошибок.

- Около 15-20% тестов представляют интеграционные тесты, которые проверяют правильность взаимодействия между модулями и компонентами.

- Оставшиеся 5-10% занимают системные тесты, которые проверяют систему в целом, включая ее функциональность, производительность и соответствие требованиям.

Такой подход позволяет достичь баланса между полным покрытием функциональности системы и эффективным использованием ресурсов для проведения тестирования.

Однако, при применении пирамиды тестирования, стоит внимательно обращать внимание на то, в каком контексте она используется, и от этого контекста значительно зависит то, какие инструменты и подходы требуются для тестирования. Так, например, если мы рассматриваем тестирование отдельного сервиса в рамках серверной части, то модульным тестированием для него будет являться тестирование отдельных функций, интеграционным тестированием – тестирование поведения этих функций в совокупности, а системным тестом – тестирование работы всего сервиса. Однако, если мы рассматриваем всю серверную часть приложения, то системный тест одного сервиса будет лишь модульным тестом серверной части, так как сервис является лишь отдельным модулем.

Иными словами, пирамида тестирования представляет собой более сложную, вложенную структуру, которая позволяет рассматривать различные виды тестирования в зависимости от контекста их применения.

Как было упомянуто выше, модульные тесты являются наиболее многочисленными. Обычно, в качестве модуля выделяется минимальная единица программы – функция. Однако, применение модульного тестирования требует качественной декомпозиции при разработке программы. Приведем пример.

**Класс RequestController**

|  |
| --- |
| public class RequestController {   public ResponseEntity<?> processRequest(HttpServletRequest request) throws SQLException {  if (request.getMethod().equals("POST")) {  *// Получение и данных из запроса* String data = request.getParameter("data");  Map<String, Object> mappedData = mapData(data);   *// Валидация данных* if (validateData(mappedData)) {  *// Работа с базой данных* Connection connection = establishDbConnection();  Statement statement = connection.createStatement();  statement.executeUpdate("INSERT INTO table (column) VALUES ('" + mappedData.get("column") + "')");  connection.commit();  connection.close();   *// Формирование ответа* Map<String, Object> response = new HashMap<>();  response.put("status", "success");  response.put("message", "Data successfully processed and stored in the database.");  return ResponseEntity.*ok*(response);  } else {  *// Обработка ошибок валидации* Map<String, Object> response = new HashMap<>();  response.put("status", "error");  response.put("message", "Invalid data. Please check your request payload.");  return ResponseEntity.*status*(HttpStatus.*BAD\_REQUEST*).body(response);  }  } else {  *// Обработка неподдерживаемых запросов* Map<String, Object> response = new HashMap<>();  response.put("status", "error");  response.put("message", "Unsupported request method. Only POST requests are allowed.");  return ResponseEntity.*status*(HttpStatus.*METHOD\_NOT\_ALLOWED*).body(response);  }  }   *// Вспомогательные методы для маппинга и валидации данных* private Map<String, Object> mapData(String data) {  *// Логика маппинга данных  // ...* }   private boolean validateData(Map<String, Object> data) {  *// Логика валидации данных  // ...* }   private Connection establishDbConnection() {  *// Логика установки соединения с базой данных  // ...* } } |

В данном абстрактном примере отсутствует декомпозиция. Вся обработка данных: обработка входящего запроса, выполнение некой бизнес-логики и работа с базой данной объединены в одну функцию. Протестировать такой код возможно, однако разработка и постоянное выполнение таких тестов займет слишком много времени. Нет никакой возможности отдельно проверить корректность работы какой-либо бизнес-логики – создание теста для такого метода потребует использования базы данных. Кроме того, такой тест будет проверять не только бизнес-логику, но и корректность получения параметров запроса, валидацию, маппинг. Все эти аспекты можно проверить отдельно, с использованием других видов тестирования и соответствующих инструментов. Кроме того, данный пример не использует возможности объектно-ориентированного программирования - он написан в структурной парадигме.

Данному коду можно противопоставить пример из пункта 1.1. Разделение кода по классам в рамках своей ответственности положительно скажется на тестируемости кода. Кроме того, это позволит подменять реализации зависимостей, например использовать InMemUserRepositoryImpl вместо реальной базы данных, что позволит не использовать реальную базу данных на этапе тестирования и ускорит процесс выполнения тестов. Еще один вариант – использовать Mocking-тестирование, которое позволяет подменять возвращаемые значения прямо внутри теста, без необходимости создавать и описывать новый класс.

## 1.4 Принципы SOLID

SOLID — это акроним, используемый для описания пяти принципов объектно-ориентированного программирования (ООП), которые помогают разработчикам создавать гибкие и поддерживаемые программные системы.

SOLID — это набор принципов, разработанных Робертом Мартином в 1990-х годах. Они были сформулированы с целью решения проблем разрастания и поддержки кода в объектно-ориентированных системах.

История возникновения SOLID началась с наблюдений Мартина над различными проектами и кодовыми базами. Он заметил, что многие системы страдают от проблем, таких как сложность разработки новых функций, хрупкость кода при внесении изменений и неэффективное повторное использование. С течением времени SOLID стал широко принятым набором принципов объектно-ориентированной разработки. Он стал основой для промышленных стандартов программирования и до сих пор является ценным инструментом для создания гибких и легко поддерживаемых программных систем.

SOLID состоит из следующих принципов:

1. Принцип единственной ответственности (Single Responsibility Principle - SRP) - каждый класс должен иметь только одну причину для изменения. Это помогает изолировать различные аспекты функциональности и улучшает поддерживаемость кода.

2. Принцип открытости/закрытости (Open/Closed Principle - OCP) - программные сущности должны быть открыты для расширения, но закрыты для модификации. Это достигается путем использования абстракций и полиморфизма, что позволяет добавлять новую функциональность, не изменяя существующий код.

3. Принцип подстановки Барбары Лисков (Liskov Substitution Principle - LSP) - объекты в программе должны быть заменяемыми и не нарушать работу системы во всех её точках. Это означает, что подклассы должны быть взаимозаменяемыми с базовым классом без изменения ожидаемого поведения.

4. Принцип разделения интерфейса (Interface Segregation Principle - ISP) - клиенты не должны зависеть от интерфейсов, которые они не используют. Разделение интерфейсов помогает создавать более гибкие системы, адаптированные к конкретным потребностям клиента.

5. Принцип инверсии зависимостей (Dependency Inversion Principle - DIP) - модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба должны зависеть от абстракций.

Так или иначе, все эти принципы направлены на уменьшение связанности (coupling) кода.

## 1.5 Связанность кода

Связанность (coupling) кода относится к степени зависимости между компонентами программы. Чем выше связанность, тем сильнее компоненты зависят друг от друга, что может затруднять изменение, тестирование и повторное использование кода.

Влияние связанности кода на программное обеспечение может быть значительным и иметь как положительные, так и отрицательные последствия. Некоторые аспекты влияния связанности кода:

1. Поддерживаемость: Высокая связанность может сделать программу более сложной для понимания, изменения и исправления ошибок. Если изменение одного компонента требует изменения множества других компонентов, то это увеличивает время и ресурсы, необходимые для поддержки программы.

2. Модульность: Низкая связанность способствует повышению модульности программы. Модули, которые не зависят слишком сильно друг от друга, могут быть разработаны, тестированы и поддерживаться независимо. Это улучшает повторное использование кода и облегчает разработку больших программных систем.

3. Масштабируемость: Высокая связанность может затруднить масштабирование системы. Если изменение одного компонента приводит к влиянию на другие компоненты, то процесс масштабирования становится более сложным и ошибочным. Низкая связанность позволяет легче добавлять и изменять компоненты программы без серьезного воздействия на другие компоненты.

4. Тестирование: Высокая связанность может затруднить тестирование программы. Если компоненты сильно связаны, то изменение одного компонента может потребовать повторного тестирования множества других компонентов, что делает тестирование более сложным и неэффективным. Низкая связанность позволяет проводить более изолированные и точечные тесты.

В том случае, если достигнута низкая связанность кода, возникает другая проблема – затрудняется управление зависимостями между компонентами программы. Когда компоненты слабо связаны, возникает необходимость управлять тем, как они взаимодействуют друг с другом и как они получают доступ к внешним ресурсам или зависимостям. Для решения этой проблемы используется концепция под названием «инверсия управления».

## 1.6 Вывод по главе 1

В главе 1 был рассмотрен объектно-ориентированный подход к разработке ПО. Были выделены основные концепции, а также проблемы, возникающие при использовании объектно-ориентированного программирования.

По итогам главы, можно сделать вывод о том, что для улучшения поддерживаемости, тестирования, масштабируемости кода требуется низкая связанность между классами и модулями разрабатываемой программы. С другой стороны, это порождает новую проблему – затрудняется управление зависимостями, так как появляется большое количество несвязанных между собой модулей.

# 2 Инверсия управления

Инверсия управления (Inversion of Control, IoC) – это принцип разработки программного обеспечения, когда контроль над потоком выполнения или зависимостями переносится на внешний фреймворк или контейнер. Вместо того чтобы явно создавать и управлять объектами и их зависимостями внутри приложения, IoC делегирует эту ответственность внешним средствам контроля. Это позволяет создавать более гибкие и расширяемые приложения.

Инверсия контроля является закономерным следствием применения принципов SOLID, в частности принципа единственной ответственности (SRP). Использование IoC снимает с компонентов всю ответственность по управлению своими зависимостями. Таким образом, класс, отвечающий за реализацию какой-то бизнес-логики, не управляет тем, какие реализации его зависимостей нужно подключать и тем более, своими транзитивными зависимостями.

Основные принципы IoC включают в себя:

1. Разделение ответственности: Приложение делегирует создание и управление объектами внешнему контейнеру, который берет на себя ответственность за их жизненный цикл: создание, подключение зависимостей, уничтожение.

2. Инъекция зависимостей: вместо того чтобы объекты создавали и управляли своими зависимостями, IoC-контейнер внедряет необходимые зависимости в объекты при создании.

3. Конфигурирование: IoC-контейнеры обычно предоставляют модель конфигурации, позволяющую определить, какие зависимости должны быть созданы и как они должны быть связаны.

Благодаря принципам IoC, разработчик может сосредоточиться на бизнес-логике приложения, не заботясь о деталях создания и управления объектами. IoC позволяет создавать более модульный, тестируемый и гибкий код.

Существует несколько реализаций инверсии управления. Они отличаются устройством работы, а также уровнем абстракции.

## 2.1 Шаблон «Фабрика»

Шаблон "Фабрика" является одной из форм инверсии контроля (IoC) и применяется для создания объектов без явного вызова конструктора напрямую. Он позволяет абстрагировать процесс создания объектов, делегируя эту задачу фабричному классу или методу.

Основная идея шаблона "Фабрика" заключается в том, чтобы предоставить специальное место или компонент, которому можно доверять создавать и возвращать экземпляры объектов вместо самого объекта. Таким образом, клиентский код вместо создания объекта напрямую обращается к фабрике, которая создает нужные объекты на основе переданных параметров или условий.

Примером может служить фабрика для создания различных типов транспортных средств, таких как автомобили или велосипеды. Фабрика может иметь методы, такие как "создатьАвтомобиль()" и "создатьВелосипед()", которые создают соответствующие объекты и возвращают их клиентскому коду.

Применение шаблона "Фабрика" в рамках IoC помогает упростить и структурировать процесс создания объектов, а также обеспечивает гибкость и легкость расширения при добавлении новых типов объектов. Зависимость между клиентским кодом и конкретными классами объектов снижается, так как фабрика выступает в качестве посредника между ними. Это позволяет легко изменять тип создаваемых объектов без внесения изменений в клиентский код, что соответствует принципам IoC.

|  |
| --- |
| *// Абстрактный класс или интерфейс, представляющий общий интерфейс для создаваемых продуктов* public interface Transport {  void deliver(); }  *// Конкретные классы-продукты* public class Car implements Transport {  public void deliver() {  System.*out*.println("Доставка автомобилем");  } }  public class Bicycle implements Transport {  public void deliver() {  System.*out*.println("Доставка велосипедом");  } }  *// Фабричный класс* public class TransportFactory {  *// Метод фабрики для создания объектов* public static Transport createTransport(String type) {  if (type.equals("car")) {  return new Car();  } else if (type.equals("bicycle")) {  return new Bicycle();  } else {  throw new IllegalArgumentException("Неизвестный тип транспорта");  }  } }  *// Пример использования фабрики* public class Main {  public static void main(String[] args) {  Transport car = TransportFactory.*createTransport*("car");  car.deliver(); *// Результат: Доставка автомобилем* Transport bicycle = TransportFactory.*createTransport*("bicycle");  bicycle.deliver(); *// Результат: Доставка велосипедом* } } |

В этом примере абстрактный класс Transport является общим интерфейсом для создаваемых продуктов. Конкретные классы Car и Bicycle реализуют этот интерфейс и представляют конкретные типы транспортных средств.

Фабричный класс TransportFactory содержит метод createTransport, который создает и возвращает объекты на основе переданного типа. В методе фабрики можно добавить дополнительные логику или параметры для создания объектов.

В клиентском классе Main используется фабрика для создания объектов и вызова методов. Клиентский код не зависит от конкретной реализации создания объектов и может управлять ими только через общий интерфейс Transport.

Данный метод является низкоуровневым с точки зрения инверсии контроля. У него есть следующие проблемы:

1. Расширяемость: при добавлении новых типов продуктов необходимо изменять код фабрики, что может привести к её модификации и перекомпиляции. Это может нарушить принцип открытости/закрытости (Open/Closed Principle) и осложнить расширение системы.

2. Поддержка зависимостей: Фабрика может столкнуться с проблемой создания объектов, которые зависят от других сложных компонентов или имеют множество параметров конструктора. Управление зависимостями может усложниться, и фабрика может потерять свою простоту.

3. Сложность конфигурации: В случае большого количества продуктов фабрика может стать слишком сложной для конфигурирования, особенно если требуется настройка зависимостей и параметров объектов.

4. Избыточность: Шаблон "Фабрика" добавляет дополнительную абстракцию между клиентским кодом и созданием объектов. Во многих случаях, особенно когда типы продуктов известны заранее и изменения маловероятны, такая дополнительная абстракция может показаться избыточной.

5. Связность (Coupling): Фабрика может создать связность между фабрикой и конкретными классами продуктов, что усложняет поддержку и изменение системы в целом.

Таким образом, шаблон «Фабрика» сам по себе сложно использовать как универсальный и гибкий механизм для инверсии контроля, однако он может использоваться при реализации других методов IoC.

## 2.2 Локатор служб

Локатор служб (Service Locator) — это паттерн инверсии контроля (IoC), который используется для поиска и предоставления служб или зависимостей в приложении. Он выполняет роль посредника, отвечая за поиск и возвращение нужной службы клиентскому коду.

Основная идея Локатора служб заключается в централизации механизма поиска зависимостей и предоставления объектов. Это позволяет клиентскому коду не заботиться о создании и управлении зависимостями, а просто запрашивать нужную службу у Локатора. Локатор служб может быть реализован в виде класса или компонента, который хранит карту или реестр зависимостей между интерфейсами и их конкретными реализациями.

Процесс работы Локатора служб может выглядеть следующим образом:

1. Клиентский код запрашивает нужную службу у Локатора служб, обращаясь к нему по интерфейсу.

2. Локатор служб ищет и создает объект, реализующий данный интерфейс, и возвращает его клиенту.

3. Клиентский код использует полученную службу для выполнения необходимых действий.

Преимущества использования Локатора служб включают:

1. Централизованное управление зависимостями: Локатор служб предлагает единый и централизованный механизм для поиска и предоставления зависимостей в приложении.

2. Сокрытие сложности создания объектов: Клиентский код не заботится о создании и жизненном цикле зависимостей, так как эта задача выполняется Локатором.

3. Гибкость и легкая замена зависимостей: Локатор служб позволяет просто изменять или заменять реализации служб без необходимости внесения изменений в клиентский код.

Однако, использование Локатора служб также имеет некоторые недостатки:

1. Скрытая зависимость: Клиентский код часто неявно зависит от Локатора служб, что может затруднить понимание и анализ зависимостей в системе.

2. Разделение интерфейсов и реализаций: Использование Локатора служб убирает явное указание зависимостей в виде интерфейсов и усложняет статический анализ и контроль зависимостей.

Пример:

|  |
| --- |
| // Определеление интерфейса сервиса public interface SomeService {  void doSomething(); }   // Реализация сервиса public class SomeServiceImpl implements SomeService {  public void doSomething() {  *// Реализация логики сервиса* } }  // Конфигурация IoC-контейнера @Configuration public class AppConfig {  @Bean  public SomeService someService() {  return new SomeServiceImpl();  } }   // Использование сервиса с применением локатора служб:  public class Main {  public static void main(String[] args) {  ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);  SomeService service = context.getBean(SomeService.class);  service.doSomething();  } } |

В данном примере **ApplicationContext** является контейнером, метод которого **getBean()** возвращает некую реализацию интерфейса, которая сконфигурирована в классе **AppConfig.** Внутри этого класса конфигурируются зависимости между компонентами системы.

Данный метод повышает уровень абстракции, однако требует доскональной настройки зависимостей с применением локатора служб.

## 2.3 Внедрение зависимостей

Внедрение зависимостей (Dependency Injection, DI) — это паттерн программирования, который позволяет управлять зависимостями между объектами. При использовании DI, зависимости передаются в объект извне вместо того, чтобы объект самостоятельно создавал или получал их. Это делает код более гибким, улучшает тестируемость и упрощает поддержку приложения.

DI обеспечивает инверсию контроля, то есть объекты не контролируют свои зависимости, а получают их извне. Реализация DI может быть осуществлена различными способами, такими как внедрение через конструктора, внедрение через setter или внедрение через интерфейс.

Одним из главных преимуществ внедрения зависимостей является полная пассивность объекта – цели для внедрения. DI-контейнер сам решает, какие зависимости требуются объекту на основе состава его полей. Это полностью соответствует принципу единственной ответственности (SRP).

### 2.3.1 Внедрение через конструктор

Внедрение зависимостей через конструктор (Constructor Injection) - это один из способов реализации паттерна Dependency Injection (DI). При данном подходе зависимости передаются в объект через его конструктор.

Когда используется внедрение через конструктор, объекты будут иметь свои зависимости доступными сразу после создания. Класс, принимающий зависимости через конструктор, не создает их самостоятельно, а получает их из внешнего источника (например, от контейнера внедрения зависимостей).

Преимущества внедрения через конструктор включают в себя:

1. Явное выражение зависимостей класса, что облегчает понимание и читаемость кода.

2. Упрощение тестирования, поскольку можно легко передать макеты зависимостей при создании объекта в тестовой среде.

3. Уменьшение связанности между классами, так как классы не зависят от конкретных реализаций зависимостей, а только от абстракций.

Пример:

|  |
| --- |
| public class MyClass {  private final DependencyA dependencyA;  private final DependencyB dependencyB;   public MyClass(DependencyA dependencyA, DependencyB dependencyB) {  this.dependencyA = dependencyA;  this.dependencyB = dependencyB;  }  *// ...* } |

В приведенном примере класс MyClass принимает две зависимости dependencyA и dependencyB через конструктор. Это позволяет использовать класс MyClass с различными реализациями DependencyA и DependencyB, обеспечивая гибкость и переиспользование кода.

### 2.3.2 Внедрение через set-метод

Внедрение зависимостей через set-метод, также известное как Setter Injection, является еще одним способом реализации паттерна Dependency Injection (DI). При использовании данного подхода зависимости передаются в объект через специальные методы-сеттеры.

При внедрении через set-метод, класс, принимающий зависимости, предоставляет публичные методы для установки значений этих зависимостей. Внешний источник (например, контейнер внедрения зависимостей) вызывает эти методы и передает соответствующие зависимости объекту.

Преимущества внедрения через set-метод включают в себя:

1. Гибкость изменения и замены зависимостей во время выполнения программы.

2. Возможность установки зависимостей сразу после создания объекта.

3. Удобство использования при наличии множества опциональных зависимостей.

Пример:

|  |
| --- |
| public class MyClass {  private DependencyA dependencyA;  private DependencyB dependencyB;   public void setDependencyA(DependencyA dependencyA) {  this.dependencyA = dependencyA;  }   public void setDependencyB(DependencyB dependencyB) {  this.dependencyB = dependencyB;  }  *// ...* } |

В приведенном примере класс MyClass предоставляет два публичных метода-сеттера setDependencyA и setDependencyB, которые позволяют установить зависимости dependencyA и dependencyB. Это позволяет изменять или заменять зависимости во время выполнения программы, обеспечивая гибкость и легкость поддержки кода.

### 2.3.3 Внедрение через интерфейс

Внедрение зависимостей через интерфейс (Interface Injection) - это еще один способ реализации паттерна Dependency Injection (DI). При данном подходе зависимости передаются в объект через методы, объявленные в интерфейсе, которые класс должен реализовать.

Внедрение через интерфейс позволяет объекту получать свои зависимости, вызывая определенные методы интерфейса. Объект, предоставляющий зависимости (например, экземпляр класса, реализующего интерфейс или контейнер внедрения зависимостей), связывает интерфейс с соответствующей зависимостью при передаче.

Преимущества внедрения через интерфейс включают в себя:

1. Ограничение зависимостей только по интерфейсам, что способствует легкости поддержки и уменьшает связанность.

2. Возможность использования различных реализаций интерфейса и легкое изменение зависимостей во время выполнения.

3. Гибкость при использовании нескольких зависимостей одного типа.

Пример:

|  |
| --- |
| public interface InjectFinder {  void injectFinder(MovieFinder finder); }  class MovieLister implements InjectFinder {   public void injectFinder(MovieFinder finder) {  this.finder = finder;  }  … } |

В приведенном примере MovieLister реализует интерфейс InjectFinder и использует метод injectFinder для установки зависимости типа MovieFinder. Таким образом, происходит связывание зависимостей через их интерфейсы.

### 2.3.4 Вывод

Внедрение зависимостей позволяет достичь инверсии контроля, где объекты не контролируют свои зависимости, а получают их из внешнего источника. Передача зависимостей через конструктор, set-методы или интерфейсы позволяет легко изменять или заменять эти зависимости, обеспечивая гибкость и масштабируемость приложения. Этот подход упрощает тестирование, так как можно передавать макеты зависимостей, а не полагаться на реальные ресурсы или сложную конфигурацию. Он также способствует повышению переиспользуемости кода, так как объекты не привязаны к конкретным реализациям зависимостей, а только к абстракциям.

Современные DI-фреймворки представляют собой мощный инструмент, повышающий уровень абстракции кода и снижающий связанность между компонентами разрабатываемой программы. Кроме того, идея DI полностью соответствует принципам SOLID.

Недостатком метода внедрения зависимостей является тот факт, что многие современные языки программирования (например Java), не поддерживают такой механизм, а архитектура программ на таких языках не позволяет полностью его задействовать. Как следствия, многие DI-контейнеры требуют использования механизмов прямого воздействия на память или рефлексии, что может оказывать влияние на производительность.

## 2.4 Вывод по главе 2

В главе 2 была рассмотрена концепция «инверсия управления», а также основные подходы к её реализации.

Инверсия управления является закономерным развитием объектно-ориентированного подхода при разработке ПО. Главная идея инверсии управления – делегация обязанностей по управлению зависимостями между компонентами отдельной системе.

В ходе главы были выделены три различных подхода к инверсии управления. Эти подходы отличаются уровнем абстракции, а также способом их реализации. В результате был выделен метод внедрения зависимостей, т.к. он предлагает высокий уровень абстракции и разделения ответственностей.

# 3 Применение инверсии управления при проектировании и разработке информационных систем

В главе 1 были описаны современные подходы к разработке и проектированию информационных систем. В главе 2 рассмотрена концепция «инверсия контроля» и методы, применяемые для её реализации.

Требования, выдвигаемые потребителями современного программного обеспечения, привели к изменению подходов к разработке. И хотя производительность все ещё остается одним из важнейших требований, наравне с ней от разработчиков программных продуктов требуется готовность легко изменять функциональность по запросу рынка.

Современные возможности аппаратного обеспечения также меняют требования к современному ПО. Увеличение вычислительных мощностей обходится дешевле, поэтому заказчики готовы жертвовать производительностью в пользу поддерживаемости и гибкости разрабатываемого ПО.

Поскольку поддерживаемость, гибкость, тестируемость, масштабируемость и повторное использование кода становится приоритетом, очень важно обеспечивать низкую связанность компонентов разрабатываемых программ. Однако низкая связанность компонентов приводит к некоторому усложнению при управлении зависимостями между компонентами системы. Для решения этой проблемы используется инверсия управлении.

Таким образом, можно сделать вывод, что использование этой концепции необходимо и обосновано в случае, когда необходимо обеспечить низкую связанность компонентов системы.

Важно иметь ввиду, что использование этого подхода может оказать незначительное влияние на производительность программ, однако это компенсируется снижением затрат на поддержку, разработку и тестирование программ.

Использование инструментов для инверсии контроля стало стандартом при разработке больших информационных систем. Так, например, Spring Framework, основанный на внедрении зависимостей, стал неотъемлемой частью экосистемы Java, а платформа .NET по умолчанию поддерживает внедрение зависимостей. Эти платформы являются одними из самых распространенных при разработке серверной части и невозможно игнорировать данный опыт.

Можно сделать вывод о том, что инверсия управления может активно применяться при проектировании и разработке информационных систем. Использование инверсии управления позволяет снизить связанность между компонентами программы, что в конечном счете упрощает поддержку, тестирование и масштабирование информационных систем.

# Список использованной литературы

1. Шаблоны корпоративных приложений. : Пер. с англ – М. : ООО «И.Д. Вильямс», 2016. – 544 с.
2. Мартин Р. Чистый код: создание, анализ и рефакторинг. – СПб.: Питер, 2023. – 464 с.: ил.
3. Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения. – СПб.: Питер, 2022. – 352 ст.: ил.
4. Уваров А.Н. Инверсия управления и внедрение зависимостей // Символ науки. – 2019. – №10-1 (22). – С. 28-32.
5. Зацепление (зависимость) и связанность, связность в объектно-ориентированном программировании (ООП) // Искусственный разум URL: https://intellect.icu/zatseplenie-zavisimost-i-svyazannost-svyaznost-v-obektno-orientirovannom-programmirovanii-oop-7010 (дата обращения: 23.10.2023).
6. Инверсия контроля (Inversion of control, IoC) // Alex tools Блог программиста URL: https://alextoolsblog.blogspot.com/2019/12/inversion-of-control.html (дата обращения: 23.10.2023).
7. Osherove R. The Art of Unit Testing with Examples in.NET. - Greenwich: Manning, 2009. - 324 c
8. Meszaros G. xUnit Test Patterns. Refactoring Test Patterns. - Boston: Addison-Wesley, 2007. - 948 c.
9. Симан М. Внедрение зависимостей в.NET. - Спб.: Питер, 2013. - 464 с.
10. Фаулер М. Рефакторинг. Улучшение существующего кода. - СПб.: Символ-плюс, 2010. - 432 с.
11. Бек К. Экстремальное программирование. Библиотека программиста. - СПб.: Питер, 2003. - 212 с.
12. Мартин Р.С. Принципы, паттерны и методики гибкой разработки на языке C#. - СПб.: Символ, 2012. - 768 с.
13. Приёмы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон и др. - СПб.: Питер, 2012. - 368 c.
14. Spring в действии. 6-е изд. / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2022. –544 с.: ил.
15. Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification // Oracle Help Center URL: docs.oracle.com (дата обращения: 23.10.2023).
16. Inversion of Control Containers and the Dependency Injection pattern // Martin Fowler URL: https://www.martinfowler.com/articles/injection.html (дата обращения: 23.10.2023).
17. Inversion of Control with the Managed Extensibility Framework (MEF) // Web Archive URL: http://web.archive.org/web/20220122095256/http://www.informit.com/articles/article.aspx?p=1635818&seqNum=2 (дата обращения: 23.10.2023).
18. Designing Reusable Classes // BRIAN FOOTE URL: http://www.laputan.org/drc/drc.html (дата обращения: 23.10.2023).
19. Guidance on software maintenance [microform] / Roger J. Martin and Wilma M. Osborne // The National Library of Australia URL: https://catalogue.nla.gov.au/catalog/3867028 (дата обращения: 23.10.2023).
20. Design Patterns: Dependency Injection // Web Archive URL: https://web.archive.org/web/20080229042522/http://msdn.microsoft.com/msdnmag/issues/05/09/DesignPatterns/default.aspx (дата обращения: 23.10.2023).
21. Дэвис Б. 97 Things Every Project Manager Should Know. - США, Калифорния: O'Reilly Media, Inc, 2009. - 250 с.