Министерство образования Республики Беларусь
Учреждение образования «Витебский государственный университет имени П.М. Машерова»
Кафедра прикладного и системного программирования

М.Г. Семенов, С.А. Ермоченко, Е.А. Корчевская

УПРАВЛЕНИЕ ПРОЕКТАМИ В СФЕРЕ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Методические рекомендации

Витебск
ВГУ имени П.М. Машерова
2020

УДК 004.42(075.8)
ББК 32.973я73
С30

Печатается по решению научно-методического совета учреждения образования «Витебский государственный университет имени П.М. Машерова». Протокол № 3 от 30.12.2019.

Авторы: доцент кафедры прикладного и системного программирования ВГУ имени П.М. Машерова, кандидат физико-математических наук **М.Г. Семенов;** заведующий кафедрой прикладного и системного программирования ВГУ имени П.М. Машерова, кандидат физико-математических наук **С.А. Ермоченко;** доцент кафедры прикладного и системного программирования ВГУ имени П.М. Машерова, кандидат физико-математических наук, доцент **Е.А. Корчевская**

Р е ц е н з е н т :
заведующий кафедрой «Информационные системы и автоматизация производства» УО «ВГТУ», кандидат технических наук, доцент *В.Е. Казаков*

Семенов, М.Г.

С30 Управление проектами в сфере информационных технологий : методические рекомендации / М.Г. Семенов, С.А. Ермоченко, Е.А. Корчевская. – Витебск : ВГУ имени П.М. Машерова, 2020. – 35 с.

В методических рекомендациях описаны общие понятия и модели управления проектами в сфере информационных технологий. На конкретных примерах показаны принципы и практики проектирования гибкой архитектуры программного обеспечения.

Предназначается для студентов второй ступени высшего образования специальности «Информатика и технологии программирования» (дисциплина «Управление проектами в сфере информационных технологий»).

УДК 004.42(075.8)
ББК 32.973я73

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Данные методические рекомендации посвящены рассмотрению основ управления проектами в сфере информационных технологий. Сегодня проектная организация деятельности в разработке программного обеспечения является всемирно признанной методологией осуществления инвестиционной деятельности. В условиях постоянно изменяющегося и развивающегося мира информационных технологий, при планировании и реализации проектов нужно быть гибкими и готовыми к постоянным изменениям. В связи с этим отдельное место в настоящем издании отводиться для рассмотрения так называемого «гибкого» (Agile) подхода к разработке.

В рамках рекомендаций рассматриваются такие темы, как элементы и термины общей теории управления проектами, роли на проекте, этапы жизненного цикла проекта, каскадная и гибкая модель жизненного цикла, философия гибкой методологии, Скрам, шаблоны, принципы и практики гибкой разработки программного обеспечения. Все главы содержат список контрольных вопросов для самопроверки. В третьей главе наряду с теоретическим материалом присутствует множество примеров исходного кода.

Материал соответствует рабочей программе курса «Управление проектами в сфере информационных технологий» (специальности второй ступени высшего образования «Информатика и технологии программирования»).

1 ОБЩИЕ ПОНЯТИЯ УПРАВЛЕНИЯ ПРОЕКТОМ

Проект [1] – это временное предприятие, направленное на создание уникального продукта, услуги или результата. Временный характер проектов подразумевает то, что проект имеет определенное начало и конец. Конец наступает в одном из двух случаев: когда цели проекта были достигнуты; когда заключается, что цели проекта не будут или не могут быть достигнуты; когда необходимость в проекте отпадает.

Каждый проект создает *уникальный* продукт, услугу или результат. Стоит понимать, что в различных проектах могут присутствовать повторяющиеся элементы, однако в целом результат проекта будет уникальным. Так например, в двух разных проектах по разработке интернет-магазина можно использовать одинаковый стек технологий ASP.NET Core / EntityFramework / Indentity / ReactJS / PostgreSQL. Однако, в данном примере могут различаться дизайн, структура базы данных, список возможностей и другие детали реализации.

Управление проектом – это применение знаний, навыков, инструментов и методов к проектной деятельности для достижения цели проекта. Управление проектом осуществляется посредством соответствующего применения и интеграции логически сгруппированных процессов управления проектами, которые подразделяются на пять групп процессов. Эти пять групп процессов:

- инициирование;
- планирование;
- выполнение;
- мониторинг и контроль;
- закрытие.

Управление проектом обычно включает следующие функции, но не ограничивается ими:

- определение требований;
- учет различных потребностей, проблем и ожиданий заинтересованных сторон при планировании и реализации проекта;
- установление, поддержание и проведение коммуникаций между заинтересованными сторонами;
- управление заинтересованными сторонами для удовлетворения требований проекта и достижения цели;
- управление ограничениями проекта, которые включают:
  - объем задач
  - качество
  - расписание
  - бюджет
  - ресурсы
  - риски.

Соотношение между ограничениями, указанными выше, таково, что если какое-либо из них изменится, то вероятно, будет затронуто как минимум одно из оставшихся. Например, если график сокращается, часто необходимо увеличить бюджет, чтобы добавить дополнительные ресурсы для выполнения того же объема работы за меньшее время. Если увеличение бюджета невозможно, объем задач или целевое качество могут быть уменьшены для достижения конечного результата проекта за меньшее время в пределах той же суммы бюджета. Заинтересованные стороны проекта могут иметь разные идеи относительно того, какие факторы являются наиболее важными, создавая еще большую проблему. Изменение требований или целей проекта может создать дополнительные риски. Команда проекта должна быть в состоянии оценить ситуацию, сбалансировать требования и поддерживать активную связь с заинтересованными сторонами для реализации успешного проекта.

В связи с возможными изменениями, разработка плана управления проектом является итеративной деятельностью, которая постепенно развивается на протяжении всего жизненного цикла проекта. Итеративная разработка включает в себя постоянное совершенствование и детализацию плана по мере появления более подробной и конкретной информации и более точных оценок. Итеративная проработка позволяет команде управления проектом определять работу и управлять ею с большей детализацией по мере развития проекта.

Менеджер проекта – это лицо, назначенное исполняющей организацией для руководства командой, отвечающей за достижение целей проекта. В целом, менеджеры проектов несут ответственность за выполнение задач, удовлетворение потребностей команды и индивидуальных потребностей. Менеджер проекта становится связующим звеном между стратегией и командой. Эффективные менеджеры проектов требуют баланса этических, межличностных и концептуальных навыков, которые помогают им анализировать ситуации и взаимодействовать соответствующим образом. Важными навыками в данном направлении являются следующие:

- лидерство
- мотивация
- общение
- влияние
- принятие решений
- политическая и культурная осведомленность
- переговоры
- доверие
- управление конфликтами.

Поскольку проекты носят временный характер, успех проекта должен измеряться с точки зрения завершения проекта в рамках ограничений по объему задач, времени, стоимости, качеству, ресурсам и риску, как утвер-

ждено между менеджерами проекта и высшим руководством. Чтобы обеспечить реализацию преимуществ проекта, тестовый период (например, плавный запуск в службах) может быть частью общего времени проекта, прежде чем передать его на постоянную работу. Успех проекта следует отнести к последним исходным условиям, утвержденным уполномоченными заинтересованными сторонами. Менеджер проекта несет ответственность за установление реалистичных и достижимых сроков для проекта и за выполнение проекта в рамках утвержденных исходных условий.

1.1 Роли на проекте

Рассмотрим, какие роли присутствуют на проекте кроме менеджера проекта.

Заинтересованная сторона – это отдельное лицо, группа или организация, которые могут повлиять, быть затронутыми или чувствовать себя затронутыми решением, действием или результатом проекта. Заинтересованные стороны могут быть активно вовлечены в проект или иметь интересы, на которые может положительно или отрицательно повлиять выполнение или завершение проекта. Различные заинтересованные стороны могут иметь конкурирующие ожидания, которые могут создать конфликты в рамках проекта. Заинтересованные стороны могут также оказывать влияние на проект, его результаты и проектную команду для достижения набора результатов, которые удовлетворяют стратегическим бизнес-целям или другим потребностям. Управление проектом – приведение проекта в соответствие с потребностями или целями заинтересованных сторон – имеет решающее значение для успешного управления взаимодействием с заинтересованными сторонами и достижения целей организации. Управление проектами позволяет организациям последовательно управлять проектами и максимизировать ценность результатов проекта и привести проекты в соответствие с бизнес-стратегией. Оно обеспечивает структуру, в которой менеджер проекта и спонсоры могут принимать решения, которые удовлетворяют как потребности и ожидания заинтересованных сторон, так и стратегические цели организации или учитывают обстоятельства, когда они могут не совпадать.

Некоторые примеры заинтересованных лиц:

• Спонсор – это человек или группа лиц, которые предоставляют ресурсы и поддержку для проекта и несут ответственность за успех. Спонсор может быть внешним или внутренним по отношению к организации менеджера проекта. От первоначальной концепции до закрытия проекта спонсор продвигает проект. Это включает в себя выступление в качестве представителя более высоких уровней управления для сбора поддержки во всей организации и продвижения выгод, которые приносит проект. Спонсор ведет проект через инициирующие процессы до получения официального

разрешения и играет важную роль в разработке первоначального объема задач. Для вопросов, которые находятся вне контроля менеджера проекта, спонсор служит путем эскалации. Спонсор также может быть вовлечен в другие важные вопросы, такие как санкционирование изменений в объеме задач, пересмотр этапов и принятие решений, когда риски особенно высоки. Спонсор также обеспечивает плавную передачу результатов проекта в бизнес запрашивающей организации после закрытия проекта.

• Клиенты (заказчики) и пользователи. Клиентами (заказчиками) являются лица или организации, которые будут утверждать и управлять продуктом, услугой или результатом проекта. Пользователи – это лица или организации, которые будут использовать продукт, услугу или результат проекта. Клиенты и пользователи могут быть внутренними или внешними по отношению к исполняющей организации, а также могут существовать на нескольких уровнях. В некоторых областях применения клиенты и пользователи являются синонимами, в то время как в других случаях клиенты относятся к субъекту, приобретающему продукт проекта, а пользователи относятся к тем, кто будет непосредственно использовать продукт проекта. Так например, при разработке нового музыкального мобильного приложения клиентами (заказчиками) могут являться владельцы бизнеса, а пользователями – люди, которые скачают это приложение.

• Продавцы (отдел продаж). Продавцы, также называемые поставщиками или подрядчиками, являются сторонними компаниями, которые заключают договорное соглашение на предоставление компонентов или услуг, необходимых для проекта. В некоторых ситуациях, у организации спонсора может присутствовать свой собственный отдел продаж.

• Деловые партнеры. Деловые партнеры – это внешние организации, имеющие особые отношения с предприятием, иногда достигаемые в процессе сертификации. Деловые партнеры предоставляют специализированные знания или выполняют определенную роль, такую как установка, настройка, обучение или поддержка.

Команда проекта включает в себя менеджера проекта и группу лиц, которые действуют вместе, выполняя работу над проектом для достижения его целей. Например, команда проекта может включать в себя дизайнеров, бизнес-аналитиков, тестировщиков, разработчиков интерфейса, разработчиков серверной части, архитекторов базы данных, сотрудники службы поддержки. Структура и характеристики проектной команды могут варьироваться в широких пределах, но одной из постоянных является роль руководителя проекта как руководителя группы.

1.2 Жизненный цикл проекта

Жизненный цикл проекта – это последовательность этапов, через которые проект проходит от его инициации до его закрытия. Этапы, как правило, являются последовательными, и их имена и номера определяются потребностями организации или организаций, участвующих в проекте, в сфере управления и контроля, характером самого проекта и областью его применения. Этапы могут быть разбиты по функциональным или частичным целям, промежуточным результатам или конечным результатам, конкретным этапам в общем объеме работ или финансовой доступности. Фазы обычно ограничены по времени, с начальной и конечной точкой или контрольной точкой. Жизненный цикл может быть задокументирован в методологии. Жизненный цикл проекта может определяться или формироваться уникальными аспектами организации, отрасли или используемой технологии. Хотя у каждого проекта есть определенное начало и определенный конец, конкретные результаты и виды деятельности, которые происходят между ними, будут сильно различаться в зависимости от проекта. Жизненный цикл обеспечивает базовую основу для управления проектом, независимо от конкретной работы.

В сфере информационных технологий обычно выделяют следующие этапы жизненного цикла:

- Инициация проекта
- Анализ
    - определение проблемы
    - составление объема задач
- Планирование
    - выработка требований
    - создание плана разработки
    - разработка архитектуры ПО, или высокоуровневое проектирование
    - детальное проектирование
- Выполнение проекта
    - программирование и отладка
    - модульное тестирование
    - интеграционное тестирование
    - интеграция
    - системное тестирование
    - внедрение
    - корректирующее сопровождение
- Завершение проекта

1.3 Заключение

В первой главе были основные понятия общей теории управления проектами, роли на проекте, элементы жизненного цикла проекта. Для более подробного изучения данной теории рекомендуем изучить источники [1–4].

1.4 Список контрольных вопросов

1. Что такое проект?
2. Назовите функции менеджера проекта.
3. Назовите элементы жизненного цикла проекта.
4. Какие роли, кроме менеджера проекта, можно выделить?
5. Выберите какой-нибудь веб-сервис, который вы активно используете. Подумайте, какие еще функции Вы бы хотели в нем увидеть. Как Вы думаете, почему их там еще нет?

2 КАСКАДНАЯ И ГИБКАЯ МОДЕЛИ УПРАВЛЕНИЯ ИТ-ПРОЕКТОМ

Существует множество различных методов и подходов к разработке программного обеспечения. Анализируя многолетний опыт исследований разработки, К. Джонс, руководитель исследовательских работ в компании Software Productivity Research, установил, что он и его коллеги сталкивались с 40 разными методами сбора требований, 50 вариантами проектирования программного обеспечения и 30 видами тестирования, применявшимися в проектах, реализуемых более чем на 700 языках программирования. Разные типы проектов призывают к разным сочетаниям подготовки и конструирования. Каждый проект уникален, однако обычно проекты подпадают под общие стили разработки. Можно выделить две наиболее популярных модели жизненного цикла проекта:

Каскадная модель жизненного цикла (также известная как полностью управляемая планом) – это такая модель, в которой объем задач проекта, а также время и затраты, необходимые для его реализации, определяются на самом раннем этапе жизненного цикла проекта, насколько это практически возможно. Как показано на Рисунке 2.1, эти проекты проходят серию последовательных или пересекающихся фаз, причем каждая фаза, как правило, фокусируется на подмножестве действий проекта и процессах управления проектом. Работа, выполняемая на каждом этапе, по своей природе обычно отличается от работы на предыдущем и последующих этапах, по-

этому состав и навыки, требуемые от проектной команды, могут варьироваться от этапа к этапу.

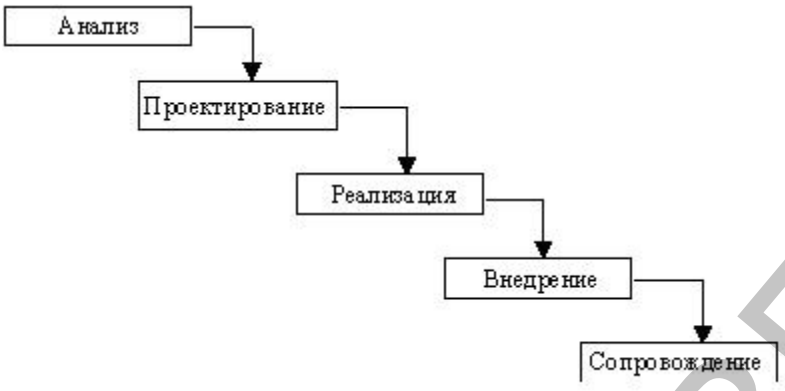

Рисунок 2.1 – **Каскадная модель жизненного цикла**

Каскадные жизненные циклы, как правило, предпочтительны, когда продукт хорошо понятен, существует значительная основа отраслевой практики или когда продукт должен поставляться в полном объеме, чтобы иметь ценность для групп заинтересованных сторон. Даже проекты с каскадными жизненными циклами могут использовать концепцию планирования с плавающей волной, когда доступен более общий план высокого уровня и выполняется более подробное планирование для соответствующих временных окон, поскольку приближаются новые рабочие операции и назначаются ресурсы.

Итеративные и инкрементные модели жизненного цикла (также известна как гибкая разработка) – это те, в которых фазы проекта (также называемые итерациями) преднамеренно повторяют одно или несколько действий проекта по мере того, как понимание проектной группой продукта увеличивается. Итерации разрабатывают продукт через серию повторяющихся циклов, в то время как приращения последовательно увеличивают функциональность продукта. Эти жизненные циклы развивают продукт и итеративно, и постепенно. Итеративные и инкрементные проекты могут выполняться поэтапно, а сами итерации будут выполняться последовательно или частично. Во время итерации будут выполняться действия всех групп процессов управления проектами. В конце каждой итерации, результат или набор результатов будут завершены. Будущие итерации могут улучшить эти результаты или создать новые. Каждая итерация постепенно создает конечные результаты до тех пор, пока не будут выполнены критерии выхода для фазы, что позволяет команде проекта включить обратную связь. В большинстве итеративных жизненных циклов будет разработано общее видение для всего предприятия, но подробный охват разрабатывается по одной итерации за раз.

Зачастую планирование следующей итерации выполняется по мере продвижения работ по области действия и результатам текущей итерации. Работа, требуемая для данного набора результатов, может различаться по продолжительности и усилиям, а команда проекта может меняться между или во время итераций. Те результаты, которые не рассматриваются в рамках текущей итерации, как правило, ограничиваются только на высоком уровне и могут быть предварительно назначены для конкретной будущей итерации.

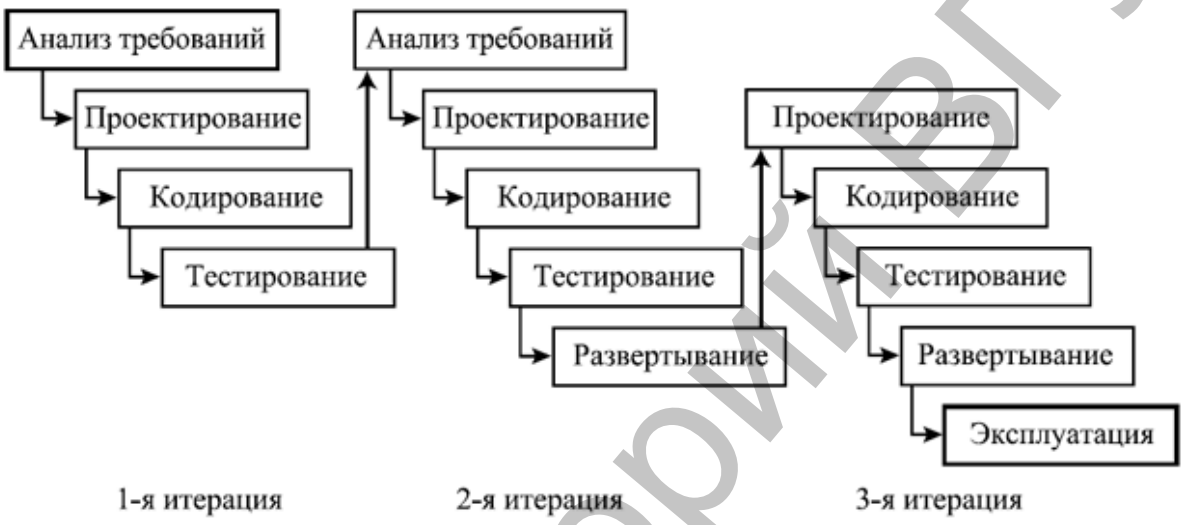

Рисунок 2.2 – **Итеративная и инкрементная модель жизненного цикла**

С. Макконелл выделяет [5] три самых популярных типа проектов, а также оптимальные в большинстве случаев методы работы над ними (Таблица 2.1):

Табл. 2.1 – Оптимальные методы работы над программными проектами трех популярных типов

	Бизнес-системы	Системы целевого назначения	Встроенные системы, от которых зависит жизнь людей
Типичные приложения	Интернет-сайты, сайты в интрасетях, системы управления материально-техническим снабжением, игры, системы управления информацией	Встроенное ПО, пакетное ПО, программные инструменты, веб-сервисы	Авиационное ПО, ПО для медицинских устройств, операционные системы
Модели жизненного цикла	Гибкая	Гибкая, каскадная	Каскадная

Исторически первые проекты в сфере информационных технологий заказывались военными и придерживались строгой и последовательной

каскадной модели. Как следствие каскадная модель стала выбором по умолчанию и для ИТ-проектов, когда они таковые стали распространяться в бизнесе. Однако с течением времени минусы такого подхода для бизнес-проектов становились все более очевидными. А именно:

1. Каждое нововведение нужно дополнительно согласовывать. Команда проекта не очень радостно воспримет новость о новом функционале. Когда в результате согласие будет достигнуто, придется вносить обновления в существующий документы или создавать дополнительные соглашения, рассчитывать затраты, объемы и сроки заново.

2. Каскадная модель как стратегия устранения финансовых рисков клиента приводит к еще большим рискам. Клиент ставит себя в определенные рамки и лишает гибкости, рискуя в итоге получить не тот товар, который интересен его аудитории, а тот который описан в документации.

3. Как правило, исполнитель вносит в бюджет риски, что увеличивает общую стоимость проекта.

Таким образом, появилось множество различных более гибких моделей управления проектами. Их объединение с формулировкой общих ценностей и принципов произошло в феврале 2001 года на встрече 17 независимых известных практиков различных методик программирования, именующих себя «Agile Alliance».

2.1 Философия гибкой разработки программного обеспечения

Ценности гибкой разработки (Agile Manifesto [6]):

Люди и взаимодействие важнее процессов и инструментов.

Работающий продукт важнее исчерпывающей документации.

Сотрудничество с клиентом важнее согласования условий контракта.

Готовность к изменениям важнее следования первоначальному плану.

Таким образом, не отрицая важности того, что справа, больше ценится то, что слева.

Основные принципы гибкой разработки:

- Наивысшим приоритетом является удовлетворение потребностей клиента, благодаря регулярной и ранней поставке ценного программного обеспечения.
- Изменение требований приветствуется, даже на поздних стадиях разработки.
- Работающий продукт следует выпускать как можно чаще, с периодичностью от пары недель до пары месяцев.
- На протяжении всего проекта разработчики и представители бизнеса должны ежедневно работать вместе.
- Над проектом должны работать мотивированные профессионалы. Чтобы работа была сделана, создайте условия, обеспечьте поддержку и полностью доверьтесь им.

- Непосредственное общение является наиболее практичным и эффективным способом обмена информацией, как с самой командой, так и внутри команды.
- Работающий продукт – основной показатель прогресса.
- Инвесторы, разработчики и пользователи должны иметь возможность поддерживать постоянный ритм бесконечно.
- Постоянное внимание к техническому совершенству и качеству проектирования повышает гибкость проекта.
- Простота – искусство минимизации лишней работы — крайне необходима.
- Самые лучшие требования, архитектурные и технические решения рождаются у самоорганизующихся команд.
- Команда должна систематически анализировать возможные способы улучшения эффективности и соответственно корректировать стиль своей работы.

Процесс гибкой разработки в общем виде представлен на рисунке 2.3.

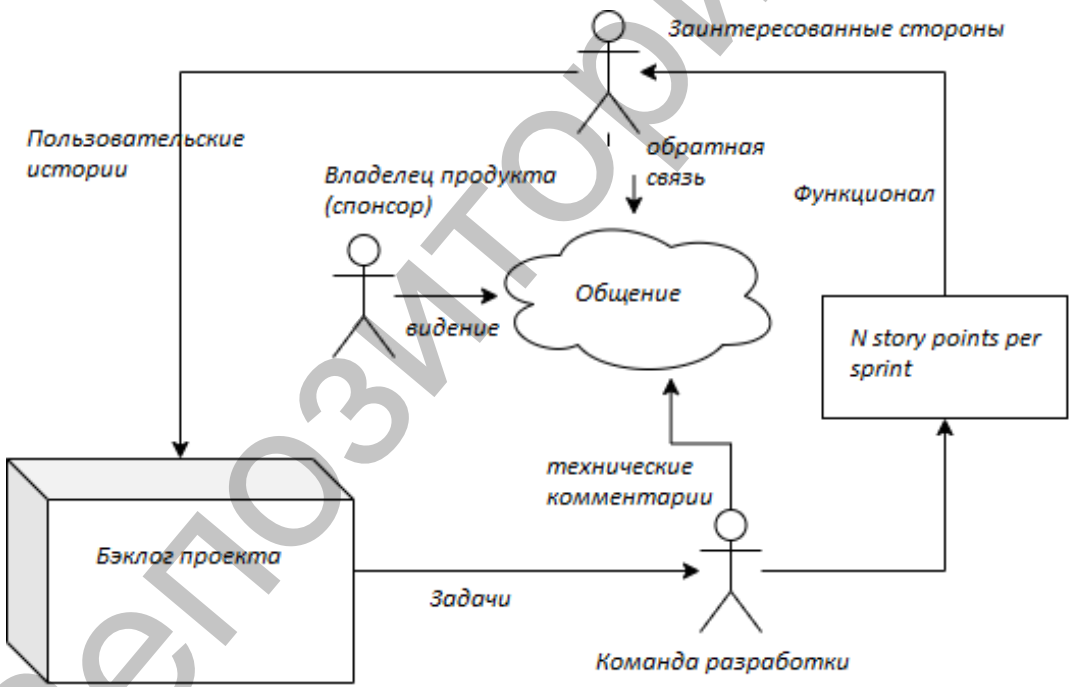

Рисунок 2.3 – **Схема процесса гибкой разработки**

Процесс гибкой разработки программного обеспечения является итеративным и инкрементным, целью каждой итерации в котором является конкретный инкремент (как правило, новый функционал) продукта. Итерацию в терминологии гибкой разработки называют *спринтом (sprint)* (фиксированный период времени, как правило, 1–4 недели). Основную

роль в данном процессе выполняет *владелец продукта* (*product owner*, спонсор). Как правило, он является изначальным инициатором проекта. У него есть четкое видение продукта. *Заинтересованные лица (stackeholders)* выражают свои пожелания о продукте, которые в терминологии гибкой модели называют *пользовательскими историями (user stories).*

На основе поступающих пользовательских историй формируется *бэклог продукта* (*backlog*) – упорядоченный на основе соотношения ценности для продукта к стоимости разработки список пользовательских историй. Как правило, бэклог пересматривается на каждой итерации. В оценке ценности для продукта и стоимости разработки участвуют как заинтересованные лица, так и команда разработки. По стоимости разработки каждую пользовательскую историю оценивают в абстрактных величинах, которые называют *стори поинты (story points)*.

На каждый спринт из верхушки бэклога выбирается список историй, которые должны составить инкремент продукта по итогу спринта. Как правило, количество историй выбирается таким образом, чтобы общая их стоимость в сторипонтах соответствовала количеству сторипонитов закрых в рамках прошлого спринта. По итогам каждого спринта проводиться добавление новых пользовательских историй, перестроение бэклога, а также поставка инкремента продукта. Все это проводится в тесном общении владельца продукта, заинтересованных лиц и команды разработки.

Данный процесс продолжается до завершения проекта.

2.2 Скрам (Scrum)

Наиболее популярным на данный момент фреймворком гибкой разработки является Скрам (Scrum, [7]).

Фреймворк – это набор базовых элементов и правил, своего рода каркас, на котором строится процесс разработки.

Скрам – это фреймворк, предназначенный для разработки, поставки и поддержки сложных продуктов. Создателями Скрама являются Кен Швабер и Джефф Сазерленд.

Скрам – это процессный фреймворк, который начали использовать для управления работой над сложными продуктами в начале девяностых годов. Скрам не является процессом, техникой или исчерпывающим методом. Напротив, Скрам – это фреймворк, в котором можно использовать разнообразные процессы и методы. Скрам проявляет несовершенства в управлении продуктом и методах работы, чтобы вы могли постоянно улучшать продукт, команду и рабочее окружение. Основными элементами фреймворка являются Скрам-команды и связанные с ними роли, события, артефакты и правила. Каждый элемент фреймворка служит определенной цели и является обязательным для успешного использования Скрама.

Правила Скрама связывают вместе *события*, *роли* и *артефакты (бэклог продукта, инкремент спринта)*, регулируют отношения и взаимодействия между ними. Они описаны далее. Существуют различные тактики использования фреймворка.

Скрам-команда состоит из Владельца Продукта, Команды Разработки и Скрам-мастера. Скрам-команды являются самоорганизующимися и кросс-функциональными. Самоорганизующиеся команды самостоятельно решают, как выполнять свою работу, а не следуют внешним указаниям. Кросс-функциональные команды обладают всеми необходимыми компетенциями для выполнения работы и не зависят от людей, которые не входят в команду. Модель команды в Скраме направлена на улучшение гибкости, творчества и продуктивности. Скрам-команды доказали свою эффективность для решения любых сложных задач. Скрам-мастер несет ответственность за продвижение и поддержку Скрама в соответствии с Руководством по Скраму. Он достигает этих целей, помогая всем понять теорию, практики, правила и ценности Скрама.

Оптимальный размер Команды Разработки достаточно мал, чтобы команда оставалась гибкой, и достаточно велик, чтобы выполнять значимую работу за время Спринта. Когда в Команде менее трех человек, взаимодействие и производительность снижается. Небольшие Команды Разработки могут столкнуться с проблемой нехватки навыков для создания готового к выпуску Инкремента Продукта. Команды размером более девяти человек испытывают трудности с координацией работы. Сложность работы в больших Командах Разработки возрастает настолько, что эмпирический процесс становится неприменим.

События Скрама. Обязательные события Скрама предусмотрены, чтобы процесс был регулярным, и другие собрания были бы не нужны. Каждое событие ограничено по времени и не может превышать максимальную установленную длительность. Продолжительность Спринта не может быть изменена после его старта. Остальные события могут быть завершены досрочно, если цели мероприятий достигнуты. Это позволяет избежать потерь времени. Каждое событие в Скраме (кроме Спринта) – это формальная возможность для инспекции и адаптации.

Инспекция. Участники процесса должны регулярно инспектировать артефакты Скрама и свой прогресс в движении к Цели Спринта, чтобы вовремя обнаружить нежелательные отклонения.

Адаптация. Если в результате инспекции выясняется, что одна или несколько характеристик процесса выходят за допустимые пределы, и это приводит продукт в неприемлемое состояние, то процесс или обрабатываемый материал необходимо изменить. Чем раньше будут внесены изменения, тем меньше риск дальнейших отклонений.

Скрам предполагает четыре формальных события для инспекции и адаптации:

• Планирование Спринта;

• Ежедневный Скрам;

• Обзор Спринта;

• Ретроспектива Спринта.

Планирование Спринта. Задачи, над которыми будет трудиться Команда Разработки во время Спринта, определяются на Планировании Спринта. План создается совместными усилиями всей Скрам-Команды. Планирование Спринта ограничено по времени. Для Спринта длительностью один месяц Планирование не должно занимать более 8 часов. Если Спринт короче, то и Планирование проводится быстрее. По результатам Планирования Спринта Скрам-команда решает:

• каким будет Инкремент в конце Спринта;

• как организовать работу, чтобы получить готовый Инкремент Продукта.

Ежедневный Скрам – это встреча Команды Разработки, которая проводится каждый день во время Спринта. Встреча не должна занимать более 15 минут, за которые Команда разработки планирует свою работу на ближайшие 24 часа. Команда оптимизирует взаимодействие между её членами и повышает свою производительность, анализируя сделанное за последние сутки и прогнозируя оставшуюся на этот Спринт работу. Команда сама определяет формат встречи, но акцент всегда остается на достижении Цели Спринта. Какие-то команды проведут дискуссию, какие-то будут использовать вопросы, например:

• Что я сделал вчера, что помогло Команде Разработки приблизиться к Цели Спринта?

• Что я сделаю сегодня, чтобы помочь Команде Разработки достичь Цели Спринта?

• Вижу ли я какие-либо препятствия, которые могут помешать мне или Команде Разработки достичь Цели Спринта?

Обзор Спринта проводится в конце Спринта с целью инспекции Инкремента и, по необходимости, адаптации Бэклога Продукта. Скрам-команда и заинтересованные лица во время Обзора Спринта совместно обсуждают, что было сделано за Спринт. Эти данные, как и любые изменения Бэклога Продукта в течение Спринта, служат основанием для обсуждения следующих шагов к оптимизации ценности Продукта. Для Спринтов длительностью один месяц продолжительность встречи не превышает 4 часов. Чем короче Спринт, тем короче его Обзор.

Ключевые элементы Обзора Спринта:

• в число участников встречи входят Скрам-команда и ключевые заинтересованные лица, которых приглашает Владелец Продукта;

• Владелец продукта объясняет, какие Элементы Бэклога готовы, а какие нет;

• Команда Разработки рассказывает о том, что получилось во время Спринта, какие возникли проблемы и как они были решены;

• Команда Разработки демонстрирует готовую работу и отвечает на вопросы об Инкременте;

• Владелец Продукта описывает текущее состояние Бэклога Продукта. При необходимости он прогнозирует возможные даты завершения разработки Продукта, основываясь на текущих показателях прогресса;

• все присутствующие обсуждают, над чем стоит работать дальше. Таким образом Обзор Спринта предоставляет ценные данные для следующего Планирования Спринта;

• проводится обзор, как изменения рынка или потенциальное использование продукта могли изменить то, что нужно сделать в первую очередь;

• выполняется обзор сроков, бюджета, возможностей и позиций на рынке для будущих релизов или возможностей продукта.

Ретроспектива Спринта – это возможность для Скрам-команды провести инспекцию, направленную на себя, и создать план улучшений командной работы в следующем Спринте. Ретроспектива Спринта проводится после Обзора Спринта и перед Планированием следующего Спринта. Максимальная продолжительность Ретроспективы – 3 часа для Спринта длительностью один месяц.

Цели проведения Ретроспективы Спринта:

• инспекция прошедшего Спринта применительно к людям, отношениям, процессам и инструментам. Обнаружение и упорядочение того, что прошло хорошо и того, что нуждается в улучшении;

• создание плана внедрения улучшений в процесс работы Скрам-команды.

С более детальным описанием фреймворка Скрам можно ознакомиться в Руководстве [7].

2.3 Заключение

В данной главе были рассмотрены две наиболее распространенные модели управления ИТ-проектами. Рассматривались виды и примеры проектов наиболее подходящие для каждой из них. Кроме того, подробно был рассмотрен наиболее популярный фреймворк управления проектами в сфере информационных технологий – Скрам.

2.4 Список контрольных вопросов

1. Охарактеризуйте каскадную модель управления проектами.
2. Для каких типов проектов лучше подходит каскадная модель?
3. Опишите процесс гибкой разработки программного обеспечения.
4. Для каких типов проектов лучше подходит гибкая модель?
5. Что такое Скрам?
6. Назовите роли Скрам-процесса?
7. Назовите и охарактеризуйте основные события Скрама.

3 УПРАВЛЕНИЕ ИСХОДНЫМ КОДОМ ИТ-ПРОЕКТА

Программное обеспечение, написанное по-разному, может решать одну и ту же проблему. При этом важно понимать, что различная структура исходного кода может быть более удобна или менее удобна для внесения дальнейших изменений в последующих итерациях проекта. Мартин Фаулер [8] определяет *рефакторинг* исходного кода (или просто рефакторинг) как «*процесс изменения программной системы таким образом, что он не изменяет внешнее поведение кода, но улучшает его внутреннюю структуру*». Таким образом, систематически проводя рефакторинг и поддерживая исходный код в виде наиболее удобном для внесения дальнейших изменений, можно значительно увеличить эффективность работы по внесению изменений и дополнений в дальнейшем. Актуальным является вопрос: какая структура исходного кода является удобной, а какая нет?

Роберт Мартин [9] предлагает следующие признаки «плохого» кода:

- жесткость
- хрупкость
- неподвижность
- вязкость
- избыточная сложность
- избыточное повторение
- неясность

Остановимся подробнее на каждом из этих признаков.

Жесткость исходного кода – это тенденция к существенному изменению программного обеспечения, даже в простых ситуациях. Конструкция является жесткой, если одно изменение вызывает каскад последующих изменений в зависимых модулях. Чем больше модулей нужно изменить, тем жестче дизайн. Большинство разработчиков так или иначе сталкивались с этой ситуацией. Их просят внести то, что кажется простым изменением. Они просматривают изменения и дают разумную оценку требуемой работы. Но позже, когда они прорабатывают изменения, они обнаруживают, что есть непредвиденные последствия изменения. Разработчики обнаруживают, что гоняются за изменениями через огромные части кода, модифицируя гораздо больше модулей, чем они первоначально оценили, и обнаруживают один за другим дополнительные изменения, которые они должны помнить, чтобы внести. В конце концов, изменения занимают намного больше времени, чем первоначальная оценка.

Хрупкость исходного кода – это тенденция к появлению ошибок во многих местах, при внесении одного изменения. Часто новые проблемы возникают в областях, которые не имеют концептуальной связи с областью, которая была изменена. Решение этих проблем приводит к еще большему количеству сложностей, и команда разработчиков начинает

напоминать собаку, преследующую хвост. Когда хрупкость модуля увеличивается, вероятность того, что изменение приведет к неожиданным затруднениям, становится очень высокой. Это кажется абсурдным, но такие модули вовсе не редкость. Это модули, которые постоянно нуждаются в ремонте, те, которые никогда не выходят из списка ошибок.

Исходный код является *неподвижным*, когда он содержит части, которые могут быть полезны в других системах, но усилия и риск, связанные с отделением этих частей от исходной системы, слишком велики. Это неудачное, но очень распространенное явление.

Вязкость имеет две формы: *вязкость исходного кода* и *вязкость среды*. Столкнувшись с изменением, разработчики обычно находят более одного способа сделать это изменение. Некоторые из способов соответствуют общему дизайн исходного кода; другие нет. Когда методы, соответствующие дизайну, использовать труднее, чем те, которые ему не соответствуют, тогда *вязкость исходного кода* высока. Легко сделать неправильную вещь, но трудно сделать правильную вещь. Мы хотим спроектировать наше программное обеспечение так, чтобы изменения, которые сохраняют дизайн, было легко сделать.

Вязкость среды возникает, когда среда разработки медленная и неэффективная. Например, если время компиляции очень велико, разработчики будут склонны делать изменения, которые не приводят к большой перекомпиляции, даже если эти изменения не сохраняют дизайн. Если системе контроля исходного кода требуется несколько часов, чтобы зарегистрировать всего несколько файлов, разработчики будут испытывать желание внести изменения, требующие как можно меньшего количества повторных проверок, независимо от того, сохранен ли дизайн.

В обоих случаях вязкий проект – это проект, в котором дизайн программного обеспечения трудно сохранить. Мы хотим создавать системы и проектные среды, которые позволяют легко сохранять и улучшать дизайн.

Дизайн обладает свойством *избыточной сложности*, когда он содержит элементы, которые в настоящее время не нужны. Это часто случается, когда разработчики прогнозируют изменения в требованиях и помещают заготовки в программное обеспечение, чтобы справиться с этими потенциальными изменениями. Поначалу это может показаться хорошим решением. В конце концов, подготовка к будущим изменениям должна сделать наш код гибким и предотвратить кошмарные изменения позже. К сожалению, эффект часто бывает противоположным. Готовясь ко многим непредвиденным обстоятельствам, дизайн перенасыщается конструкциями, которые никогда не используются. Некоторые из них могут пригодиться в дальнейшем, но многие другие – нет. Между тем, дизайн несет вес этих неиспользованных элементов. Это делает программное обеспечение сложным и трудным для понимания.

Избыточное повторение. Копирование и вставка могут быть полезными операциями редактирования текста, но в тоже время они могут быть катастрофическими операциями редактирования исходного кода. Слишком часто программные системы построены на десятках или сотнях повторяющихся элементов кода. Это происходит так: Андрей должен написать некоторый код. Он может найти полезные элементы для решения поставленной задачи в других частях исходного кода. После этого, он копирует и вставляет этот код в свой модуль и вносит соответствующие изменения. До этого код, который был искусно скопирован Андреем из другого модуля, был помещен туда Еленой, которая скопировала его из модуля, написанного Сергеем. Сергей написал этот код с нуля самостоятельно, однако впоследствии обнаружил критическую ошибку. После исправления данной критической ошибки Сергеем, модули Андрея и Елены все еще продолжают содержать её.

Когда один и тот же код появляется снова и снова в несколько разных формах, разработчики пропускают абстракцию. Обнаружение всего повторения и устранение его с помощью соответствующей абстракции, возможно, не занимает приоритетного места в их списке задач, но это может иметь большое значение для облегчения понимания и обслуживания системы. Когда в системе имеется избыточный код, работа по изменению системы может стать трудной. Ошибки, обнаруженные в таких повторяющихся единицах, должны быть исправлены при каждом повторении. Однако, поскольку каждое повторение немного отличается от другого, исправление не всегда одинаково.

Неясность – это склонность модуля к пониманию. Код может быть написан в ясной и выразительной манере, или он может быть написан непрозрачным и запутанным способом. Код, который развивается со временем, становится все более непрозрачным с возрастом. Требуется постоянное усилие, чтобы код был ясным и выразительным, чтобы непрозрачность была минимальной Когда разработчики впервые пишут модуль, код может показаться им понятным. В конце концов, они погрузились в это и поняли это на интимном уровне. Позже, после того, как близость прошла, они могут вернуться к этому модулю и задаться вопросом, как они могли написать что-нибудь столь ужасное. Чтобы предотвратить это, разработчики должны поставить себя на место своих читателей и предпринять согласованные усилия по рефакторингу своего кода, чтобы их читатели могли его понять. Им также нужно, чтобы их код был рассмотрен другими.

Рассмотрим теперь некоторые принципы, соблюдение которых позволяет увеличить качество исходного кода.

3.1 Принцип единственной ответственности

Этот принцип был описан в работе Т. Демарко и М. Пейдж-Джонса. Они назвали его сплоченностью, которую они определили как функциональную взаимосвязь элементов модуля. В последующем Р. Мартином было предложено изменить это значение и связать сплоченность с причинами, которые вызывают изменение модуля или класса: «*Модуль должен иметь единственную причину для изменений*».

Предположим, что мы разрабатываем модуль системы компьютерной геометрии. В данной системе необходимо реализовать возможности отображения различных геометрических фигур, а также вычисления их числовых характеристик (например, площади). Рассмотрим пример класса, реализующего работу с прямоугольниками в данной системе (Рисунок 3-1).

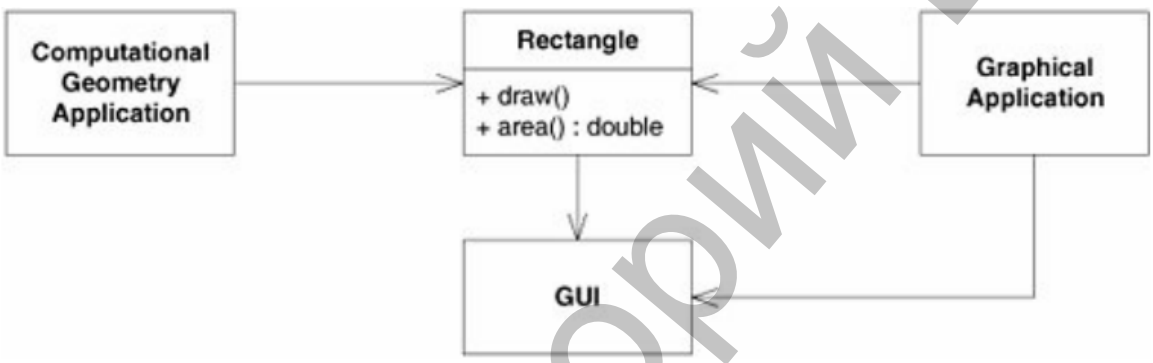

Рисунок 3.1 – **Диаграмма системы компьютерной геометрии**

Два разных приложения используют класс Rectangle. Одно приложение производит вычисления, а другое занимается отображением. Этот дизайн нарушает принцип единственной ответственности. Класс Rectangle имеет две ответственности и как следствие – две причины для изменения. Первая заключается в предоставлении математической модели геометрии прямоугольника. Вторая – в визуализации прямоугольника в графическом интерфейсе. Нарушение данного принципа вызывает несколько неприятных проблем. Во-первых, мы должны включить GUI в приложение вычислительной геометрии. В большинстве современных языках программирования сборка GUI должна была бы быть построена и развернута с приложением вычислительной геометрии. Во-вторых, если изменение приложения GraphicalApplication по какой-либо причине вызывает изменение Rectangle, это изменение может привести к пересборке, повторному тестированию и повторному развертыванию приложение Computational Geometry Application. Если мы забудем это сделать, это приложение может сломаться непредсказуемым образом.

В данном примере, лучшим вариантом является разделение двух обязанностей на два совершенно разных класса, как показано на рисунке 3.2.

Этот дизайн перемещает вычислительные части Rectangle в класс GeometricRectangle. Теперь изменения, внесенные в способ визуализации прямоугольников, не могут повлиять на приложение Computational Geometry Application.

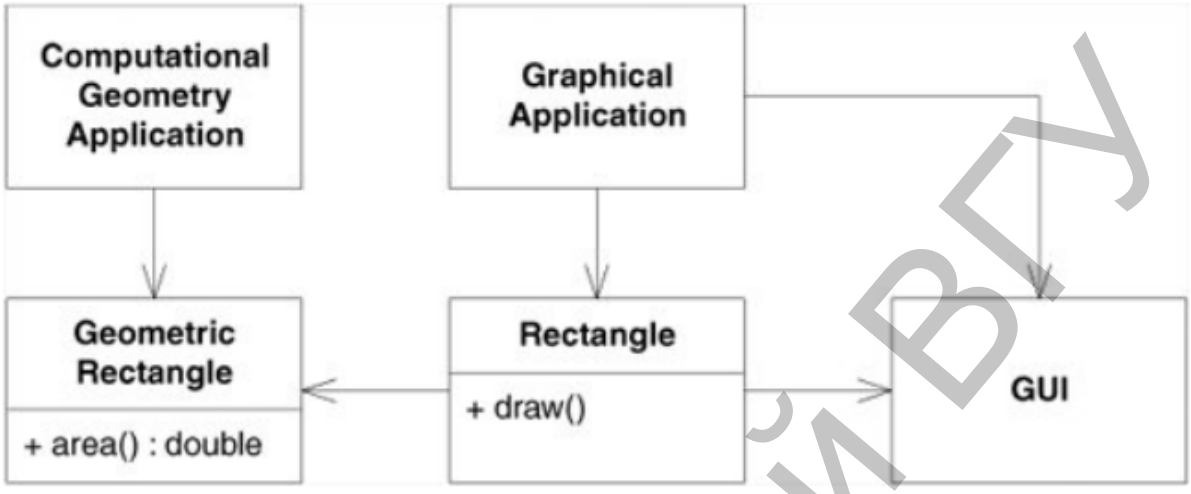

Рисунок 3.2 – **Диаграмма системы компьютерной геометрии, соответствующая принципу единственной ответственности**

Принцип единственной ответственности – один из самых простых принципов для понимания, но и один из самых трудных для применения. Объединение обязанностей – это то, что мы делаем постоянно. Поиск и разделение этих обязанностей – это то, чему на самом деле необходимо уделять особое внимание во время дизайна программного обеспечения.

3.2 Принцип открытости/закрытости

«Программные объекты (классы, модули, функции и т.д.) должны быть открыты для расширения, но закрыты для модификации.»

Когда одно изменение в программе приводит к каскаду изменений в зависимых модулях, дизайн обладает свойством жесткости. Принцип открытости/закрытости рекомендует проводить рефакторинг системы, чтобы дальнейшие изменения такого рода не вызывали больше модификаций. Если принцип открытости/закрытости применяется хорошо, дальнейшие изменения достигаются путем добавления нового кода, а не путем изменения старого кода, который уже протестирован и хорошо работает.

Модули, соответствующие принципу открытости/закрытости, имеют два основных свойства:

1. Они *открыты* для расширения. Это означает, что поведение модуля может быть расширено. По мере изменения требований приложения существует возможность расширения модуля новым поведением, которые

удовлетворяют этим изменениям. Другими словами, мы можем изменить то, что делает модуль.

2. Они *закрыты* для модификации. Расширение поведения модуля не приводит к изменению исходного или двоичного кода модуля. Бинарная исполняемая версия модуля в связываемой библиотеке, DLL или файле .EXE остается неизменной.

Казалось бы, эти два свойства противоречат друг другу. Обычный способ расширить поведение модуля – внести изменения в исходный код этого модуля. Обычно считается, что модуль, который нельзя изменить, имеет фиксированное поведение. Как это возможно, что поведение модуля может быть изменено без изменения его исходного кода? Не меняя модуль, как мы можем изменить то, что делает модуль? Ответ абстракция. В C # или любом другом объектно-ориентированном языке программирования можно создавать абстракции, которые являются фиксированными и представляют неограниченную группу возможных вариантов поведения. Абстракции являются абстрактными базовыми классами (или интерфейсами), а неограниченная группа возможных поведений представлена всеми возможными производными классами (или реализациями интерфейса). Модуль может манипулировать абстракцией. Такой модуль может быть закрыт для модификации, поскольку он зависит от фиксированной абстракции. Тем не менее, поведение этого модуля может быть расширено путем создания новых производных абстракции.

Рисунок 3-3 показывает простой дизайн, который не соответствует принципу открытости/закрытости. Классы Client и Server являются конкретными. Класс Client использует класс Server. Если мы хотим, чтобы объект Client использовал другой объект сервера, класс Client должен быть изменен, чтобы назвать новый класс сервера.

Рисунок 3.3 – **Пример простого нарушения принципа открытости/закрытости**

На рисунке 3.4 показан дизайн, который решает ту же самую задачу, что и предыдущий, но при этом соответствует принципу открытости/закрытости с использованием шаблона Стратегия [10]. В этом случае класс ClientInterface является абстрактным с абстрактными функциями-членами. Класс Client использует эту абстракцию. Однако объекты класса Client будут использовать объекты производного класса Server. Если мы хотим, чтобы в некоторых ситуациях объекты Client использовали другой класс сервера с измененным по-

ведением, можно создать новый производный от ClientInterface класс. При этом класс Client останется без изменений.

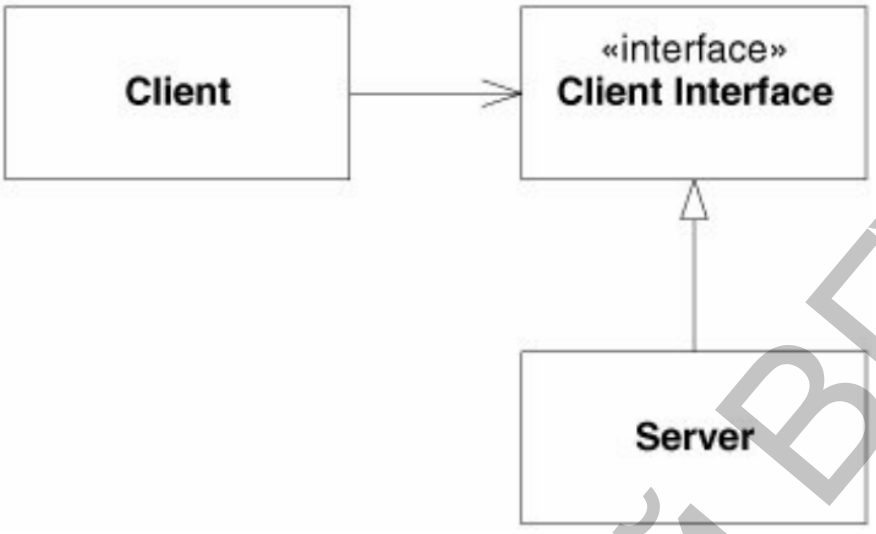

Рисунок 3.4 – **Пример дизайна соответствующего принципу открытости/закрытости**

3.3 Принцип подстановки Лисков

Основными механизмами соблюдения принципа открытости/закрытости являются абстракция и полиморфизм. В статически типизированных языках, таких как C#, одним из ключевых механизмов, поддерживающих абстракцию и полиморфизм, является наследование. Именно с помощью наследования мы можем создавать производные классы, которые реализуют абстрактные методы в базовых классах. Однако слепое использование механизма наследование может привести к еще большим проблемам. Для предотвращения большинства наиболее часто встречающихся ошибок наследования необходимо следовать принципу подстановки Лисков: «Пусть q(x) является свойством верным относительно объектов x некоторого типа T. Тогда q(y) также должно быть верным для объектов y типа S, где S является подтипом типа T». Также существует более упрощенная формулировка для механизма наследования: «Объекты производных классов должны корректно замещать объекты базового класса».

Для лучшего понимания данного принципа рассмотрим пример его нарушения. Рассмотрим класс Rectangle, который хранит размеры прямоугольника и содержит логику вычисления его площади:

```
public class Rectangle
{
     private Point topLeft;
     private double width;
     private double height;
```

```
        public double Width
        {
            get { return width; }
            set { width = value; }
        }

        public double Height
        {
            get { return height; }
            set { height = value; }
        }

        public double Area(){
            return width * height;
        }
    }
```

Предположим, что появилась необходимость в создании класса Square, который определяет квадраты схожим образом. Очевидно, что все квадраты являются прямоугольниками, и площадь может быть вычислена аналогичным образом. Эта ситуация выглядит идеальной для применения механизма наследования. Единственное, что необходимо изменить – это поведение изменения размеров. Действительно, ведь у квадрата мы не можем поменять ширину, не изменив высоты, иначе он не будет являться квадратом. Итоговый код класса Square может выглядеть следующим образом:

```
public class Square : Rectangle
{
        public override double Width
        {
            set
            {
                base.Width = value;
                base.Height = value;
            }
        }

        public override double Height
        {
            set
            {
```

```
            base.Height = value;
            base.Width = value;
        }
    }
}
```

Предположим теперь, что для класса Rectangle был написан следующий метод:

```
void testRectangle(Rectangle r)

{

    r.Width = 5;

    r.Height = 4;

    if(r.Area() != 20)

        throw new Exception("Bad area!");

}
```

Действительно, данный метод будет прекрасно отрабатывать с объектом класса Rectangle. Однако, при передаче в данный метод объект класса-наследника Square будет выброшено исключение. Таким образом происходит нарушение принципа подстановки Лисков.

3.4 Принцип инверсии зависимостей

Данный принцип состоит из двух утверждений:

1. Модули высокого уровня не должны зависеть от модулей низкого уровня. И те и другие должны зависеть от абстракций.

2. Абстракции не должны зависеть от деталей реализации. Детали должны зависеть от абстракций.

Применение термина «инверсия» в названии этого принципа обусловлена тем, что традиционные методы разработки программного обеспечения, такие как структурированный анализ и проектирование, имеют тенденцию создавать программные структуры, в которых модули высокого уровня напрямую зависят от модулей низкого уровня и в которых политика (абстракция) зависит от деталей (Рисунок 3-5). Действительно, одна из целей этих методов состоит в том, чтобы определить иерархию подпрограмм, которая описывает, как модули высокого уровня выполняют вызовы модулей низкого уровня. Структура зависимостей хорошо спроектированной объектно-ориентированной программы «инвертирована» по отно-

шению к структуре зависимостей, которая обычно является результатом традиционных процедурных методов.

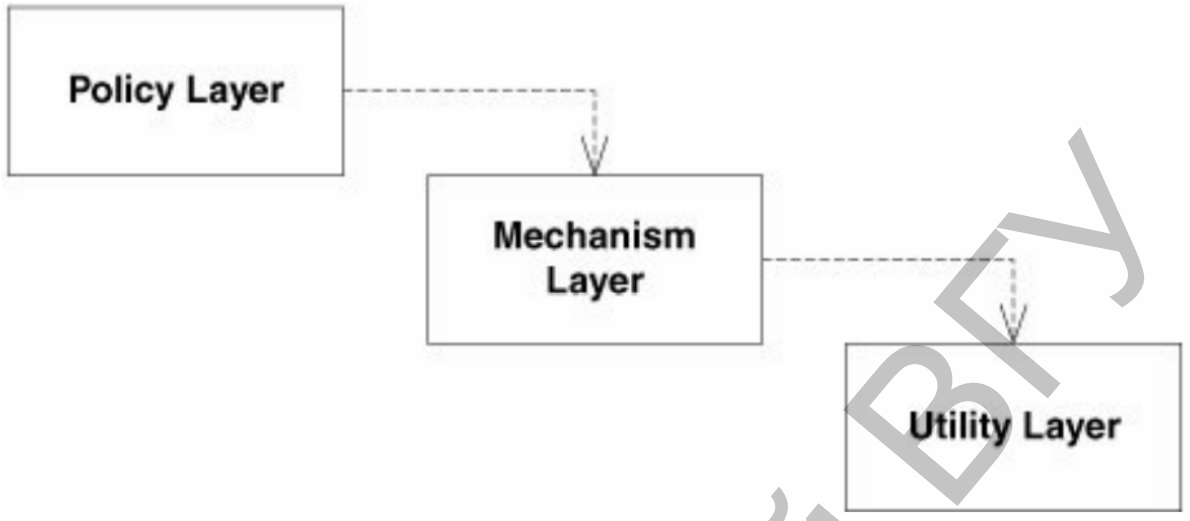

Рисунок 3.5 – **Пример прямой зависимости модулей высокого уровня от низкоуровневых модулей (нарушение принципа инверсии зависимостей)**

Рассмотрим значение модулей высокого уровня, которые зависят от модулей низкого уровня. Это модули высокого уровня, которые содержат важные архитектурные решения и бизнес-модели приложения. Эти модули содержат идентификационные данные приложения. Тем не менее, когда эти модули зависят от модулей нижнего уровня, изменения в модулях нижнего уровня могут оказывать непосредственное влияние на модули более высокого уровня и могут приводить к их каскадному изменению. Очевидно, что такой подход не является хорошим решением. Это высокоуровневые модули формирования политики, которые должны влиять на подробные модули низкого уровня. Модули, которые содержат бизнес-правила высокого уровня, должны иметь приоритет над модулями, содержащими подробности реализации, и быть независимыми от них. Высокоуровневые модули просто не должны никоим образом зависеть от низкоуровневых модулей. Более того, это модули высокого уровня, определяющие политику, которые как правило наиболее часто переиспользуются. Когда высокоуровневые модули зависят от низкоуровневых модулей, становится очень трудно повторно использовать эти высокоуровневые модули в разных контекстах. Однако, когда высокоуровневые модули независимы от низкоуровневых модулей, высокоуровневые модули можно использовать довольно просто. Этот принцип лежит в основе дизайна большинства современных фреймоврков.

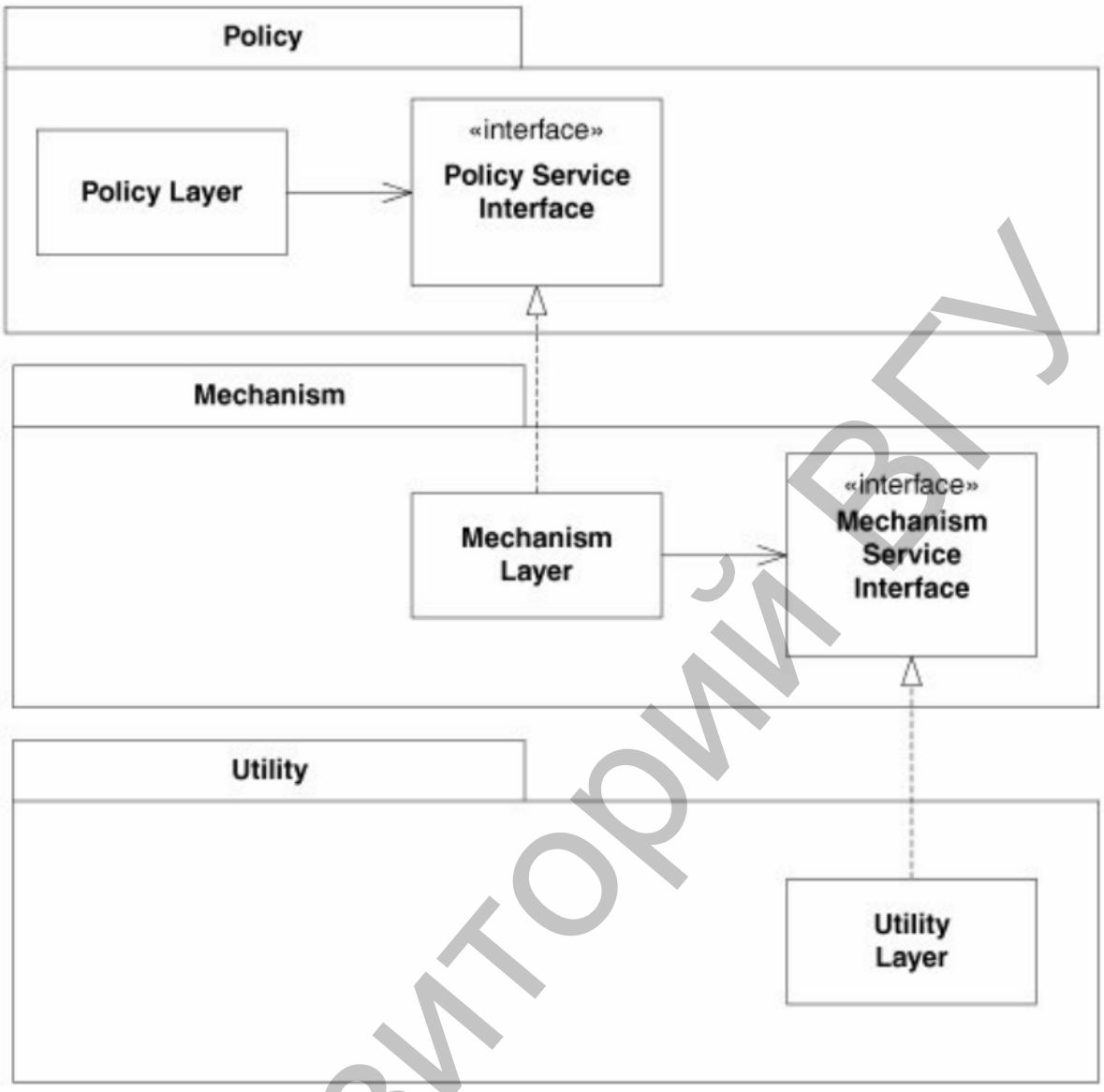

Рисунок 3.6 – **Пример инверсии зависимостей модулей высокого уровня от низкоуровневых модулей через интерфейсы**

На рисунке 3.6 показана более подходящая модель. Каждый уровень верхнего уровня объявляет абстрактный интерфейс для необходимых ему сервисов. Уровни более низкого уровня затем реализуются на основе этих абстрактных интерфейсов. Каждый класс более высокого уровня использует следующий самый нижний уровень через абстрактный интерфейс. Таким образом, верхние слои не зависят от нижних слоев. Вместо этого нижние уровни зависят от абстрактных сервисных интерфейсов, объявленных на верхних уровнях.

3.5 Принцип разделения интерфейсов

Данный принцип решает проблемы возникающий при активном использовании так называемых «толстых» интерфейсов. Если интерфейс класса может быть разбит на группы методов так, что каждая группа обслуживает различный набор клиентов, то это признак "толстого" интерфейса. Принцип разделения интерфейсов признает, что есть классы, интерфейсы которых не обладающих высокой связностью, однако при этом клиенты не должны воспринимать их как один класс. Вместо этого различные клиенты должны знать о различных абстракциях, которые имеют связные интерфейсы, а их объединение совпадает с интерфейсом исходного класса.

Простыми словами принцип разделения интерфейсов может быть сформулирован следующим образом: «Клиенты не должны быть вынуждены зависеть от методов, которые они не используют». Когда клиенты вынуждены зависеть от методов, которые они не используют, эти клиенты подвергаются изменениям в этих методах. Это приводит к непреднамеренной связи между всеми клиентами.

Рассмотрим конкретный пример. Разрабатывается система безопасности, в которой объекты дверей можно блокировать и разблокировать, и знать, открыты они или закрыты. Эта дверь проектируется как интерфейс, так что клиенты могут использовать объекты, которые соответствуют интерфейсу двери, без необходимости зависеть от конкретных реализаций двери.

```
public interface Door
{
     void Lock();
     void Unlock();
     bool IsDoorOpen();
}
```

Теперь учтите, что одна из таких реализаций, TimedDoor, должна подать сигнал тревоги, если дверь оставлена открытой слишком долго. Для этого объект TimedDoor связывается с другим объектом, называемым Timer.

```
public class Timer
{
     public void Register(int timeout, TimerClient
client)
     {/*code*/}
}

public interface TimerClient
{
     void TimeOut();
}
```

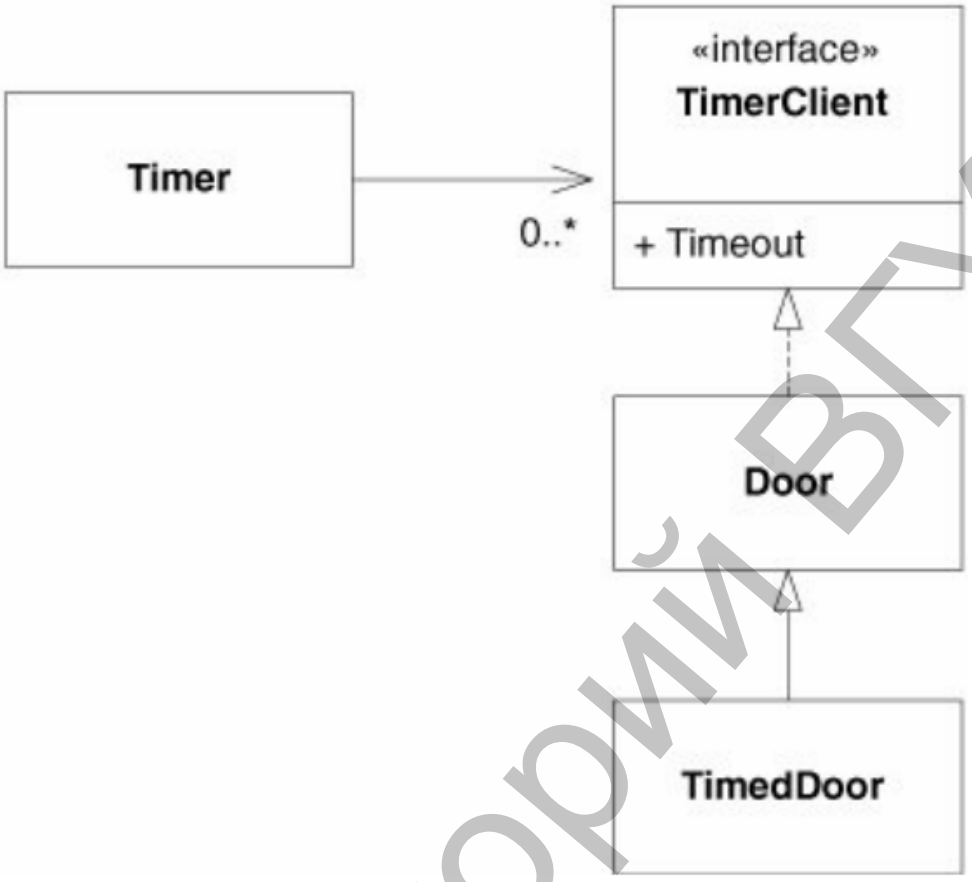

Рисунок 3.7 – **Дизайн системы безопастности**

Когда объект желает получить информацию о тайм-ауте, он вызывает функцию регистрации таймера. Аргументами этой функции являются время ожидания и ссылка на объект TimerClient, функция TimeOut которого будет вызвана по истечении времени ожидания. Как мы можем заставить класс TimerClient взаимодействовать с классом TimedDoor, чтобы код в TimedDoor мог быть уведомлен о времени ожидания? Есть несколько альтернатив. Рисунок 3-7 показывает общее решение. Мы заставляем Door, и, следовательно, TimedDoor, наследоваться от TimerClient. Это гарантирует, что TimerClient может зарегистрироваться в Timer и получить сообщение TimeOut.

Проблема этого решения в том, что класс Door теперь зависит от TimerClient. Не все виды дверей нуждаются в информации о времени. Действительно, оригинальная дверная абстракция не имела никакого отношения к времени. Если создаются производные Door без учета времени, они должны будут предоставить вырожденные реализации для метода TimeOut, что в свою очередь является потенциальным нарушением принципа подстановки Лисков. Кроме того, приложения, использующие эти производные, должны будут импортировать определение класса TimerCli-

ent, даже если он не используется. Такой подход приводит к избыточной сложности.

Это пример загрязнения интерфейса, синдрома, который распространен в статически типизированных языках, таких как C#, C++ и Java. Интерфейс двери был загрязнен методом, который не требуется. Он был вынужден включить этот метод исключительно в интересах одного из его подклассов.

Одним из решений является создание класса, производного от TimerClient и делегирующего TimedDoor (Рисунок 3-8). Когда объект такого класса хочет зарегистрировать запрос на тайм-аут с помощью Timer, объект класса TimedDoor создает DoorTimerAdapter и регистрирует его с помощью Timer. Когда Timer отправляет сообщение TimeOut в DoorTimerAdapter, объект DoorTimerAdapter делегирует сообщение обратно TimedDoor.

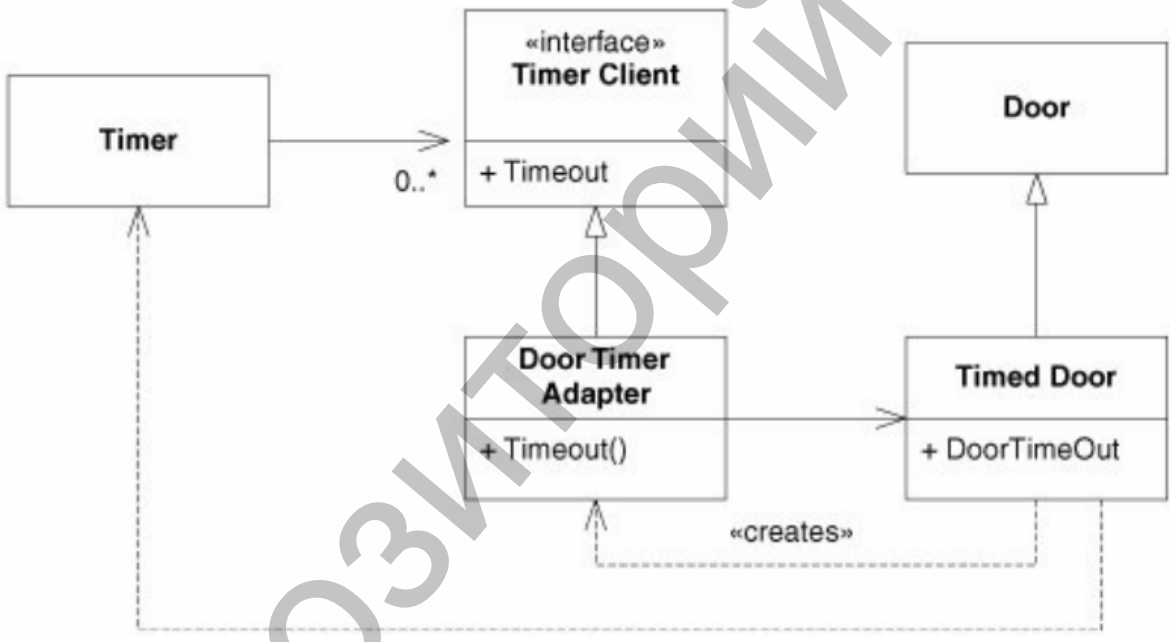

Рисунок 3.8 – **Решение проблемы загрязнения интерфейсов через делегирование**

3.6 Заключение

В данной главе были рассмотрены признаки не гибкой архитектуры исходного когда. Такая архитектура требует применения большого количество усилий и времени для внесения изменений и дополнений в проект. Также было рассмотрено пять основных принципов, которые позволяют создавать намного более гибкую архитектуру. Разработка дополнительного функционала для систем, спроектированных следуя этим принципам, будет требовать гораздо меньшего вложения ресурсов. Стоит отметить, что данные принципы также называют акронимом SOLID (по первым буквам

их названий на английском языке), они получили широкое распространение и на текущий момент их применение являются одной из лучших практик по разработке программного обеспечения. Более подробно с данными принципами, а также другими шаблонами и практиками разработки гибкого программного обеспечения можно ознакомиться, изучив источники [8–10].

3.7 Список контрольных вопросов

1. Назовите признаки архитектуры исходного кода, которые создают дополнительные проблемы при внесении изменений и дополнений в требования проекта.

2. Что такое принцип единственной ответственности?

3. Что такое принцип открытости/закрытости?

4. Что такое принцип подстановки Лисков?

5. Назовите какой-нибудь из способов решения проблемы загрязнения интерфейсов. Приведите пример.

6. Рассмотрите исходный код одного из Ваших учебных проектов. Проанализируйте его на соответствие принципам SOLID. Какие изменения необходимо внести в архитектуру проекта, что бы он лучше соответствовал этим принципам?

ЛИТЕРАТУРА

1. Project Management Institute A guide to the project management body of knowledge (PMBOK guide). / Project Management Institute – 5th edition. – Newtown Square, PA. – 589 p.

2. Гейзлер, П.С. Управление проектами: учеб. пособие / П.С. Гейзлер. – Минск: БГЭУ, 2005. – 255 с.

3. Попов, Ю.И. Управление проектами: учебник для слушателей общеобразоват. учреждений, обучающихся по программе МВА и др. программам подготовки управленческих кадров / Ю.И. Попов; Ин-т экономики и финансов «Синергия». – М.: Инфра-М, 2005. – 208 с.

4. Коваленко, С.П. Управление проектами: практ. пособие / С.П. Коваленко. – Минск: Тетралит, 2013. – 192 с.

5. Макконнелл, С. Совершенный код: [практ. рук-во по разработке программного обеспечения] / [пер. с англ. под общ. ред. В.Г. Вшивцева]. – СПб. [и др.]: Питер, 2007. – 896 с.

6. Beck, K. Manifesto for Agile Software Development [Электронный ресурс] / K. Beck. – Mode of access: https:// https://agilemanifesto.org/. – Date of access: 29.01.2020.

7. Schwaber, K. The Scrum Guide [Электронный ресурс]. – Mode of access: https://www.scrumguides.org/scrum-guide.html – Date of access: 02.02.2020.

8. Фаулер, М. Шаблоны корпоративных приложений / М. Фаулер [и др.]. – М.: Вильямс, 2010. – 544 с.

9. Martin, Robert C. Agile Principles, Patterns, and Practices in C# / Robert C. Martin – Upper Saddle River, NJ: Pearson Education, 2006. – 771 p.

10. Гамма, Э. Приёмы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес; [пер. с англ. А. Слинкин]. – СПб.: Питер, 2014. – 368 с.

Учебное издание

СЕМЕНОВ Максим Геннадьевич
ЕРМОЧЕНКО Сергей Александрович
КОРЧЕВСКАЯ Елена Алексеевна

УПРАВЛЕНИЕ ПРОЕКТАМИ В СФЕРЕ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Методические рекомендации

Технический редактор *Г.В. Разбоева*
Компьютерный дизайн *Л.Р. Жигунова*

Подписано в печать 2020. Формат 60x84 $^{1}/_{16}$. Бумага офсетная.
Усл. печ. л. 2,03. Уч.-изд. л. 2,02. Тираж экз. Заказ .

Издатель и полиграфическое исполнение – учреждение образования
«Витебский государственный университет имени П.М. Машерова».

Свидетельство о государственной регистрации в качестве издателя,
изготовителя, распространителя печатных изданий
№ 1/255 от 31.03.2014 г.

Отпечатано на ризографе учреждения образования
«Витебский государственный университет имени П.М. Машерова».
210038, г. Витебск, Московский проспект, 33.