Министерство науки и высшего образования Российской Федерации
ФГАОУ ВО «Южно-Уральский государственный университет (НИУ)»
Высшая школа электроники и компьютерных наук
Кафедра «Информационно-аналитическое обеспечение управления
в социальных и экономических системах»

РАБОТА ПРОВЕРЕНА
Рецензент,
начальник отдела web-разработки
_____ / Н.А. Филипп/
«_____» _____2020 г.

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой,
д.т.н., профессор
_____ / О.В. Логиновский /
«_____» _____2020 г.

Снижение затрат ИТ-предприятия за счет управления процессом проектирования архитектуры программного обеспечения на базе методологии Domain Driven Design

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
ЮУрГУ – 09.04.01.2020.663 ПЗ ВКР

Руководитель ВКР,
к.х.н., доцент
_____ / А.В. Голлай/
«___» _____ 2020 г.

Автор ВКР,
Студент группы КЭ-221
_____ /Д.С. Жаворонков/
«___» _____ 2020 г.

Нормоконтролер,
к.т.н., доцент
_____ / В.Н. Любицын/
«___» _____ 2020 г.

Челябинск 2020

АННОТАЦИЯ

Жаворонков Д.С. Снижение затрат ИТ-предприятия за счет управления процессом проектирования архитектуры программного обеспечения на базе методологии Domain Driven Design. –
Челябинск: ЮУрГУ, КЭ-221,
2020, 99 с., 26 ил., 8 табл., библиогр.
список – 30 наим..

В дипломной работе рассмотрены особенности проектирования архитектуры программного обеспечения на IT-предприятие, рассмотрены подходы и современные методологии построения информационных систем.

Цель работы - снизить затраты IT-предприятия посредством управления процессом проектирования архитектуры программного обеспечения.

В рамках выполнения ВКР рассмотрены и решены следующие задачи:

● Произведен анализ особенностей проектирования архитектуры ПО.

● Рассмотрено понятие IT-предприятия.

● Рассмотрены современные подходы к разработке ПО, проанализированы плюсы и минусы каждого из подходов, сделан выбор.

● Разработаны требования к «устойчивой» архитектуре ПО.

● Снижены затраты предприятия посредством использования методологии Domain Driven Design при разработке программного обеспечения и сделаны выводы по снижению затрат предприятия.

● Проанализированы результаты затрат IT-предприятия на предмет эффективности использования методологии Domain Driven Design.

ОГЛАВЛЕНИЕ

АННОТАЦИЯ 2
ОГЛАВЛЕНИЕ 3
ВВЕДЕНИЕ 8
1 ОСОБЕННОСТИ ПРОЕКТИРОВАНИЯ АРХИТЕКТУРЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 9
1.1 Что такое IT-предприятие 9
1.2 Понятие ПО 9
1.2.1 Классификация программного обеспечения 10
1.2.2 Жизненный цикл ПО 11
1.3.3 Этапы жизненного цикла ПО 11
1.3 Архитектура программного обеспечение 12
1.3.1 Задачи архитектуры программного обеспечения 13
1.3.2 Типы архитектуры программного обеспечения 14
Многоуровневая архитектура 14
Событийно-ориентированная архитектура 16
Микроядерная архитектура 17
Микросервисная архитектура 18
Монолитная архитектура 20
1.3.3 Критерии «хорошей» архитектуры 21
1.3.4 Критерии «плохой» архитектуры 22
1.4 Управление процессом проектирования архитектуры IT-предприятия 22
1.5 «Технический долг» и почему он появляется 23
1.5.1 Понятие «технического долга» 23
1.5.2 Причины появления технического долга 24
1.6 Волатильность финансовых издержек предприятия при проектировании ПО 25
Выводы по разделу один 27

2. ОБЗОР СОВРЕМЕННЫХ ПОДХОДОВ К РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И МЕТОДОЛОГИЙ ПРОЕКТИРОВАНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ 29
2.1 Основные модели разработки ПО 29
2.1.1 Waterfall (каскадная модель, или «водопад») 29
2.1.2 V-образная модель (разработка через тестирование) 30
2.1.3 Инкрементная модель 31
2.1.4 Спиральная модель 33
2.1.5 «Agile Model» (гибкая методология разработки) 34
2.1.6 Сравнительный анализ моделей разработки 34
2.2 Методологии проектирования архитектуры информационных систем на базе Agile 38
2.2.1 Test-driven development 38
2.2.2 Behaviour-driven development 39
2.2.3. Domain-driven design 40
2.2.4 Сравнительный анализ методологий проектирования 41
2.3 Глоссарий Domain Driven Design 43
Выводы по разделу два 53
3. РАЗРАБОТКА ТРЕБОВАНИЙ К АРХИТЕКТУРЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 54
3.1 Архитектурные требования 54
3.2 Функциональные требования системы установки видеокамер 57
3.3 Лингвистические требования 57
3.4 Варианты использования системы 57
Выводы по разделу три 58
4. СНИЖЕНИЕ ЗАТРАТ ПРЕДПРИЯТИЯ ПОСРЕДСТВОМ УПРАВЛЕНИЯ ПРОЕКТИРОВАНИИ ПО С ПОМОЩЬЮ МЕТОДОЛОГИИ DOMAIN DRIVEN DESIGN 59
4.1 Диаграмма бизнес-процесса в нотации BPMN 59
4.2 Разработка системы автоматизации установки камер 60
4.2.1 Внедрение единого языка 61

4.2.2 Составление карты контекстов 62
4.2.3 Практическая разработка продукта 62
Проектирование базы данных 62
Агрегаты, DTO, entity, repositories 63
4.3 Экономика IT - предприятия 63
4.3.1 Затраты на разработку без использования методологии Domain Driven Design 64
4.3.2 Затраты на разработку с применением методологии Domain Driven Design 66
4.3.3 Анализ полученных результатов 68
Выводы по разделу четыре 69
ЗАКЛЮЧЕНИЕ 71
БИБЛИОГРАФИЧЕСКИЙ СПИСОК 73
ПРИЛОЖЕНИЕ – Листинг программных элементов. 76

ВВЕДЕНИЕ

С каждым годом все больше и больше предприятий Российской Федерации проводят цифровизацию, переходят на электронный учет и автоматизируют бизнес-процессы. Для реализации данных задач создаются IT-предприятия, которые, применяя современные технологии, производят на свет программное обеспечение для управления предприятием и сопутствующих процессов.

Для создания и сопровождения таких систем мало нанять технических специалистов, необходимо внедрить управление процессом проектирования программного обеспечения, научиться создавать горизонтально-масштабируемы продукты. Данная ВКР освещает эту актуальную проблему.

Цель работы – снизить затраты IT-предприятия посредством управления процессом проектирования архитектуры программного обеспечения.

Для достижения цели в работе были поставлены следующие задачи:

- Рассмотреть понятие IT-предприятия.
- Провести анализ особенностей проектирования архитектуры ПО.
- Рассмотреть современные подходы к разработке ПО, проанализированы плюсы и минусы каждого из подходов, сделан выбор.
- Выработать требования к «устойчивой» архитектуре ПО.
- Снизить затраты предприятия посредством управления процессом разработки программного обеспечения с использованием методологии Domain Driven Design и сделать выводы по снижению затрат предприятия.

Объект работы – управление процессом проектирования архитектуры ПО на IT-предприятии.

Предмет работы – методологии управление процессом проектирования архитектуры ПО на IT-предприятии.

Результаты работы: рекомендуется использовать методологии Domain Driven Design при проектировании архитектуры ПО на IT-предприятии.

1 ОСОБЕННОСТИ ПРОЕКТИРОВАНИЯ АРХИТЕКТУРЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1 Что такое IT-предприятие

Информационные технологии - это отрасль, которая охватывает все, что касается поиска, хранения и обработку информации. Соответственно IT-предприятие - это предприятие, производящее инструментарий для работы с информацией, именуемое программным обеспечением (далее ПО) [1].

1.2 Понятие ПО

Программное обеспечение (ПО) - это совокупность всех программ и соответствующей документации, обеспечивающая использование ЭВМ в интересах каждого ее пользователя.

Различают системное и прикладное ПО. Схема представления программного обеспечения представлена на рисунке 1.1:

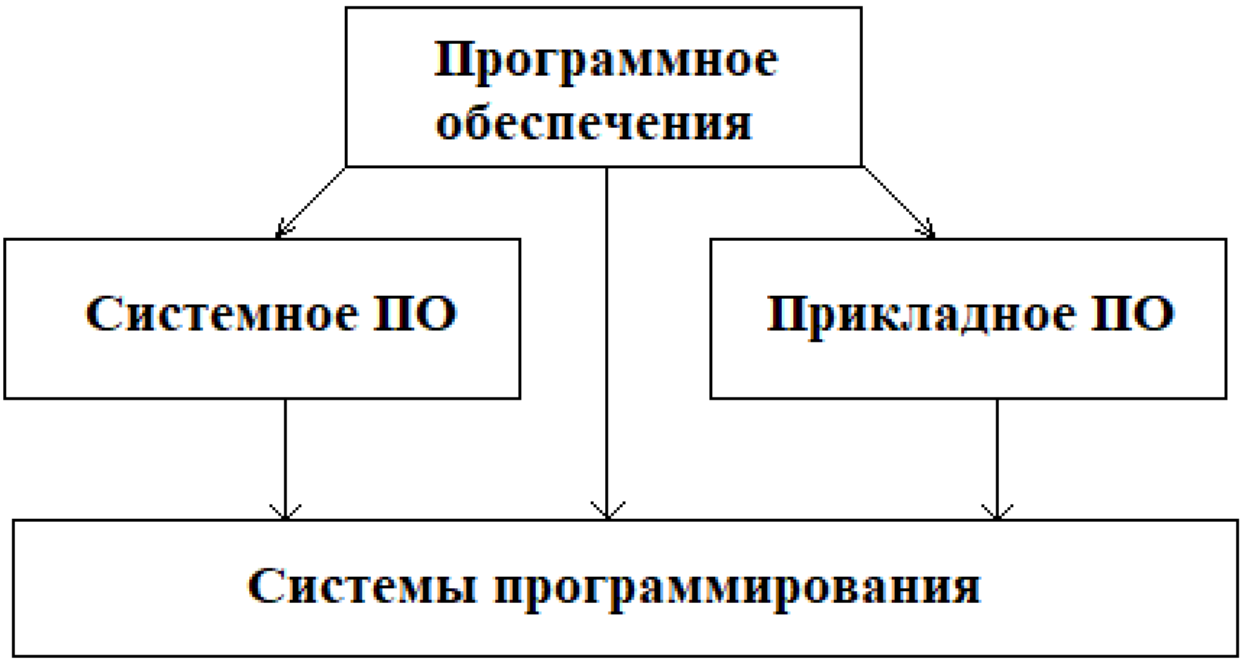

Рисунок 1.1 – Схема представления ПО

1.2.1 Классификация программного обеспечения

Системное ПО – это совокупное множество программ для обеспечения слаженной работы компьютера. Различают сервисное и базовое ПО. Системные программы являются главными звеньями для управления работой вычислительной системы, выполняют различные опциональные функции (тестирования, форматирования, копирования, выдачи справок и т. д) [2].

Базовое программное обеспечение инкапсулирует в себя:

- оболочки;
- операционные системы;
- сетевые операционные системы.

Сервисное ПО включает в себя программы (утилиты):

- антивирусные;
- диагностики;
- архивирования;
- обслуживания сети;
- обслуживания носителей.

Прикладное ПО – это совокупность программ, основной задачей которых является решение проблем предметной области. Например, программа Mathcad для визуализации алгебраических вычислений является прикладным ПО Прикладное ПО работает только при наличии системного ПО [3].

Прикладные программы называют приложениями. Они включают в себя:

- базы данных;
- текстовые процессоры;
- интегрированные пакеты;
- табличные процессоры;
- системы графики делового стиля (графические процессоры);
- экспертные системы;
- обучающие программы;
- программы математических расчетов, анализа и моделирования;

- игры;
- коммуникационные программы.

1.2.2 Жизненный цикл ПО

Следует начать с определения. Жизненный цикл программного обеспечения (Software Life Cycle Model) – это период времени, который начинается с момента принятия решения о создании программного продукта и заканчивается в момент его полного изъятия из эксплуатации. Этот цикл – процесс построения и развития ПО.

1.3.3 Этапы жизненного цикла ПО

У любого программного обеспечения есть жизненный цикл – этапы, через которые оно проходит с начала создания до конца разработки и внедрения. Чаще всего это подготовка, проектирование, создание и поддержка [4]. Этапы могут называться по-разному и дробиться на более мелкие стадии. На рисунке 1.2 изображены этапы жизненного цикла ПО:

Рисунок 1.2 – Этапы жизненного цикла ПО

Автор проводит исследование этапов жизненного цикла ПО на примере интернет-магазина.

Подготовка. Предприниматель принимает решение о запуске книжного e-commerce магазина и проводит анализ, существующих решений в сети интернет. Аккумулирует информацию о трафике и функционалу решений.

Проектирование. Выбрав компанию, которая будет брать подряды на работу, предприниматель обсуждает со специалистами архитектуру и будущего e-commerce приложения, рассматривает дизайн.

Реализация. Заключение договора между предпринимателем и компанией подрядчиком. Компания начинает писать код, создавать прототипы, дизайн, писать документацию.

Тестирование. После проведения тестов разработчиков, начинается так называемое зеленое тестирование. Заказчик проверяет UI/UX системы, проверяет функционал.

Поддержка. «Подписание акта приемки». Подрядчик размещает интернет-магазин на «боевом» сервере. Целевая аудитория начинает посещать данный ресурс, попутно сообщая об ошибках в техническую поддержку, а разработчики - оперативно исправляют.

1.3 Архитектура программного обеспечение

Архитектура ПО – это структура программы или вычислительной системы, характеризующая ее работу на различных высоких уровнях, например: программные компоненты, аппаратные блок, доступные извне свойства этих компонентов, связи между ними, документация системы Документирование архитектуры существенно упрощает взаимодействие между участниками проекта, позволяет зафиксировать принятые на ранних этапах проектирования решения о высокоуровневом дизайне системы и переиспользовать блоки этого дизайна и паттерны повторно в других проектах.

Для разработки архитектуры системы привлекаются специалисты со следующими ролями: системный архитектор (проектирует систему в целом, а также отдельные ее компоненты), архитектор базы данных (занимается

проектированием БД и ее структуры), системный аналитик (участвует в проектировании, подготавливает документацию), администраторы (участвуют в проектировании аппаратной части системы).

Application architect или архитектур приложения обладает значимой ролью при разработке ПО. Если представленная им архитектура приложения не будет реализовывать поставленные бизнесом цели, то это может, например, увеличить сроки выполнения проекта, а следовательно - снизить прибыль.

1.3.1 Задачи архитектуры программного обеспечения

На основании исследования автор утверждает, что на архитектуру приложения ложатся важные задачи, а именно:

● Оптимизация продуктивности бизнес-процессов. Классическими ожиданиями заказчика от интеграции ПО являются: уменьшение времени выполнения задач; автоматизация технологического процесса;

● Уменьшение затрат. Одной из целей разработки может стать уменьшение затрат, необходимых при совершении каких-либо действий. Это может осуществляться как за счет повышения продуктивности процессов, так и за счет ускорения выполнения операций.

● Улучшение операционной деятельности. Операционная деятельность обычно связана с выполнением рутинных типовых операций (например, работа кассира в магазине, прием коммунальных платежей и т.д.). Автоматизируя (упрощая, ускоряя) такую операционную деятельность, можно снижать затраты либо увеличивать производительность системы.

● Повышение эффективности управления. Зачастую правильно составленная архитектура положительно влияет на внутренние процессы бизнеса. Например, электронный документооборот позволяет эффективно взаимодействовать при децентрализации предприятия.

● Диверсификация рисков. Предпринимательская деятельность всегда связана с рисками. Одной из целей разработки архитектуры приложения является

их снижение. Примером автор выделяет двухфакторную авторизацию в банковских электронных системах.

● Повышение продуктивности работы пользователей. Под пользователями можно понимать, как сотрудников самой компании (в этом случае повышение продуктивности можно отнести к целям, затрагивающим процессы), так и клиентов компании, которые будут пользоваться разработанным ПО (чем комфортней клиентам, тем меньше вероятность, что они перейдут к конкурентам).

● Уменьшение стоимости «поддержки» жизненного цикла ПО.

1.3.2 Типы архитектуры программного обеспечения

Существует множество типов, классификаций архитектур ПО, каждая со своими достоинствами и недостатками. Автор проведет анализ нескольких, самых популярных из них.

Многоуровневая архитектура

Многоуровневая архитектура [7] является одной из самых популярных архитектур. Это обусловлено достаточно «низким» порогом вхождения и эффективностью использования. Одним из самых известных примеров данной архитектуры является сетевая модель OSI.

Приложение разделяется на уровни, каждый из которых взаимодействует лишь с двумя соседними.

Архитектура подразумевает произвольно количество слоев (уровней). Хорошей практикой считается использовать трехуровневые системы: уровень представления, уровень бизнес-логики и уровень данных.

Схематичное изображение многоуровневой архитектуры представлено на рисунке 1.3:

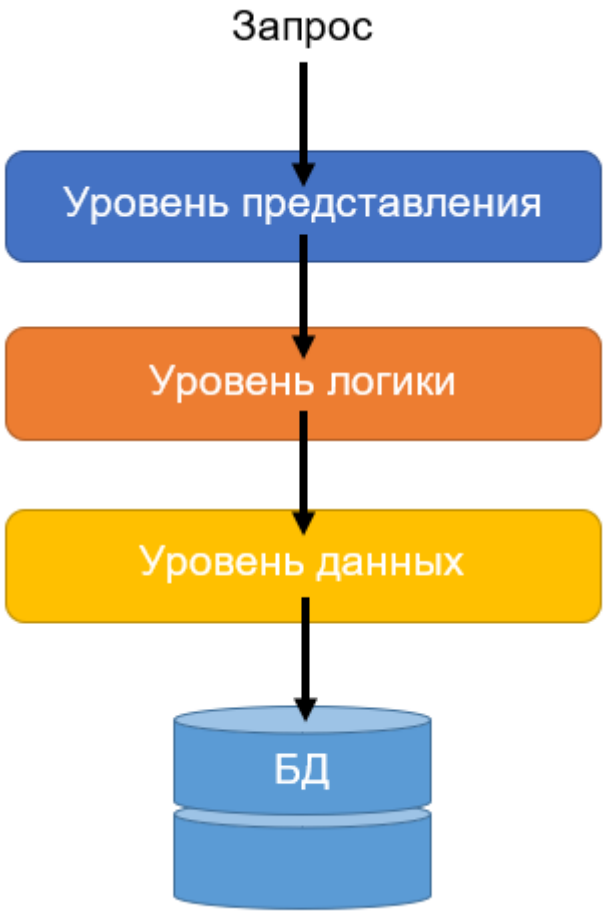

Рисунок 1.3 – Многоуровневая архитектура ПО.

Достоинства:

● Инкапсуляция данных между уровнями. Каждый слой выполняет строго свой спектр задач и изменение его содержимого не влияет на соседние. Изоляция слоев обусловливается разным функционалом от уровня к уровню.

● Системы с многоуровневой системой являются очень распространенной моделью и, как следствие, повлияли на создание генераторов шаблонов. Примерами могут служить такие продукты, как LASG для Visual Studio. Инструмент предлагает методы генерации кода, избавляющие от рутинной работы и позволяя строить application layers.

Недостатки:

● Данная модель подразумевает использование нового уровня абстракции для решения любой проблемы. К сожалению такой подход чреват путаницей в кодовой базе и не слаженной командной работе.

● Скорость разработки. Количество слоев увеличивает количество этапов разработки, ведь каждая операция должна находиться на своем уровне. «Анти-паттерн», который на практике проявляется, как неистовое количество операций и кода для, казалось бы, простого функционала.

● Отладка работы. Если ошибка находится на каком-то промежуточном уровне, прежде чем попасть в базу данных, необходимо будет моделировать полный список действий для имитации пресловутой ошибки.

Событийно-ориентированная архитектура

Архитектура, характеризующаяся событиям [8] является паттерном проектирования архитектуры программного обеспечения, в основе которого лежат события и реакция системы на них.

Событие – некое значительно изменение состояния системы. Для примера рассмотрим покупку в интернет-магазине. Приобретая телефон, покупатель инициирует событие, которое переводит состояние телефона из состояния «продается» на состояние «проданный». Данная архитектура воспринимает это изменение как событие и реагирует, как-то требует бизнес. В частном случае это может вызывать другие события, моделирующие реальную жизнь.

Представленный паттерн отлично подходит для систем со слабо связанными компонентами. Основные понятия: источник события, потребитель, событие, реакция.

Главная особенность данной архитектуры заключается в возможности интерактивно представлять процесс, а также асинхронно взаимодействовать.

Схема событийно-ориентированной архитектуры представлена на рисунке 1.4:

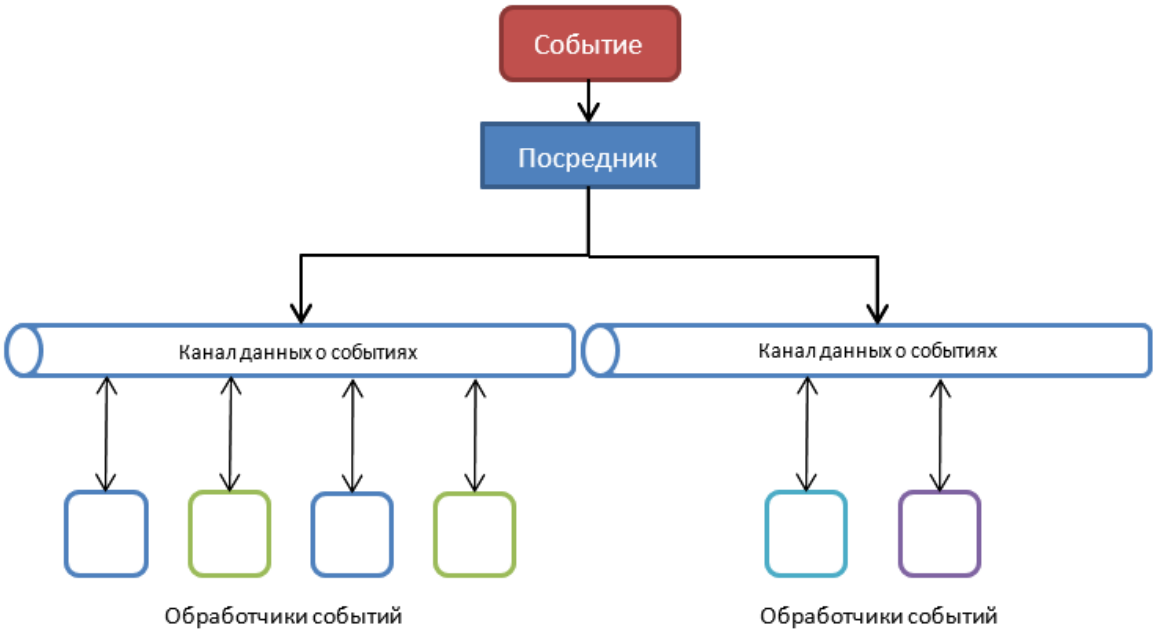

Рисунок 1.4 – Событийно-ориентированная архитектура

Достоинства архитектуры:

● Асинхронность модулей. Так называемый StateLess. Модуль собирается как конструктор с помощью специальных строителей - билдеров, что позволяет повысить производительность системы и легко масштабировать ее.

Недостатки:

● Сложность отладки. Имитация цепочки событий- трудоемкий процесс.

● Требуется отлаженный механизм транзакционности.

● Сложная структура логов.

● Сложные условия обработки ошибок.

Микроядерная архитектура

Микроядерная архитектура имплементирует два компонента: плагины и ядро системы. Плагины реализуют в себе все действия с бизнес-логикой, в то время как ядро управляет их взаимодействием.

Схема микроядерной архитектуры изображена на рисунке 1.5:

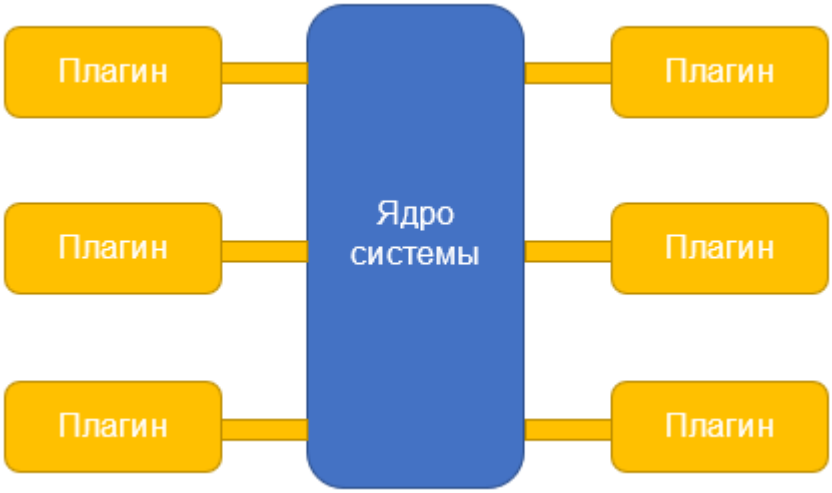

Рисунок 1.5 – Микроядерная архитектура

Достоинства архитектуры:

● Кроссплатформенность. Простота переноса системы между средами, обусловливается разделением политик высокого уровня и низкоуровневых механизмов.

Недостатки:

● Количество модулей может отрицательно сказаться на производительности приложения. При этом может быть проблематично достигнуть консенсуса между числом задач микроядра и количеством плагинов.

● Роль микроядра системы переоценена. Если в начале разработке было принято неправильное решение о дроблении микроядра, может быть слишком поздно что-то менять.

Микросервисная архитектура

В основе данной архитектуры лежит дробление системы на маленькие, независимые части - микросервисы [10]. Каждый блок этой системы может быть реализован с помощью разных технологий, децентрализован по расположению. Общение между такими частями реализовывают с помощью какого-нибудь протокола в купе с технологией, например, HTTP – REST API.

На практике микросервисы выделяют в так называемые контейнеры, которые инкапсулируют их содержимое друг от друга. Примерами систем контейнеризации являются Docker и Kubernetes. Рассмотрим достоинства и недостатки данного типа архитектуры.

Достоинства:

● Масштабируемость системы. Новый функционал подразумевает новый, отдельный сервис, который никак не связан с системой. Если функционал устарел - можно безболезненно удалить.

● Ввиду малой связности сервисов между собой, разработку продукта можно легко распределять между независимыми командами программистов.

Схема данной архитектуры изображена на рисунке 1.6:

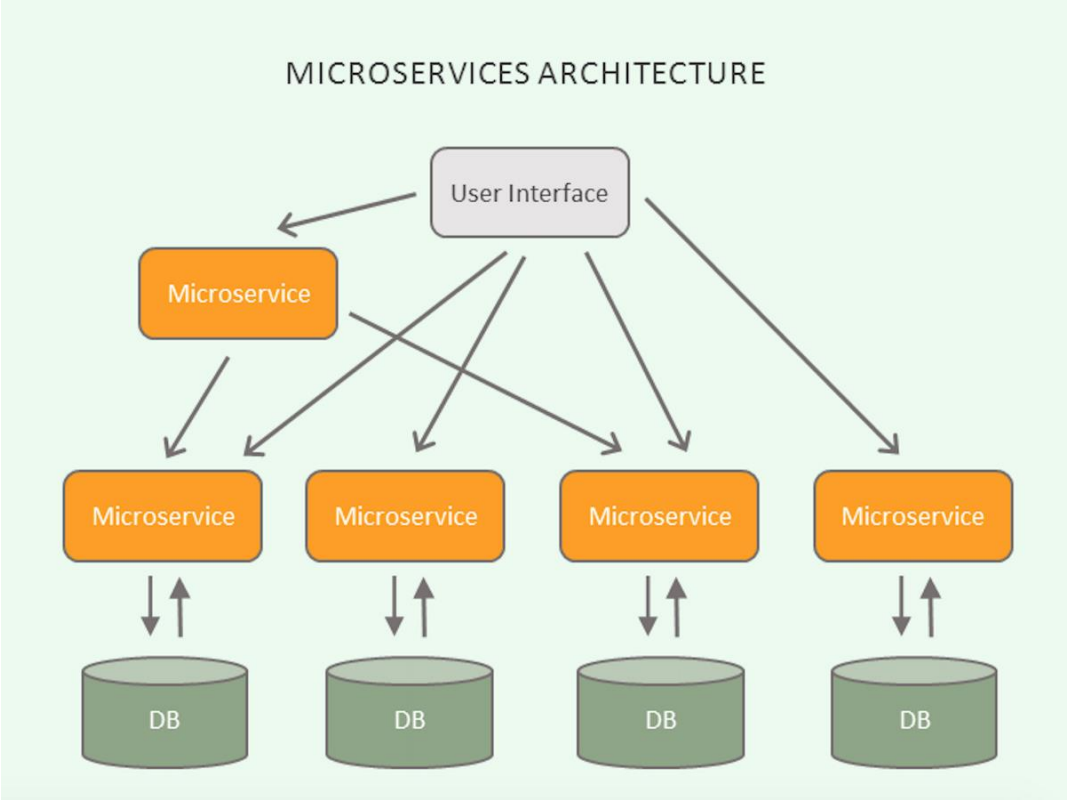

Рисунок 1.6 – Микросервисная архитектура

Недостатки:

● Сложность отлаживания ошибок.

● Сетевые издержки взаимодействия микросервисов.

● Согласованность в конечном счёте. Так как каждый микросервис имеет собственное хранилище, к которому обращаются другие микросервисы, может возникнуть условия гонки, когда часть микросервисов обладает устаревшими данными.

Монолитная архитектура

Монолит дословно означает большой неделимый камень. Несмотря на широкой популярности данного термина, обозначает он одно и тоже. В программировании архитектура монолита относится к единой атомарной частице. Смысл монолитного ПО - различные компоненты приложения объединяются в одну программу на одной платформе. Классически монолит включает в себя UI интерфейс, базу данных, сервер. Все части программного обеспечения унифицированы, и все его функции управляются в одном месте. Монолитная архитектура представлена на рисунке 1.7:

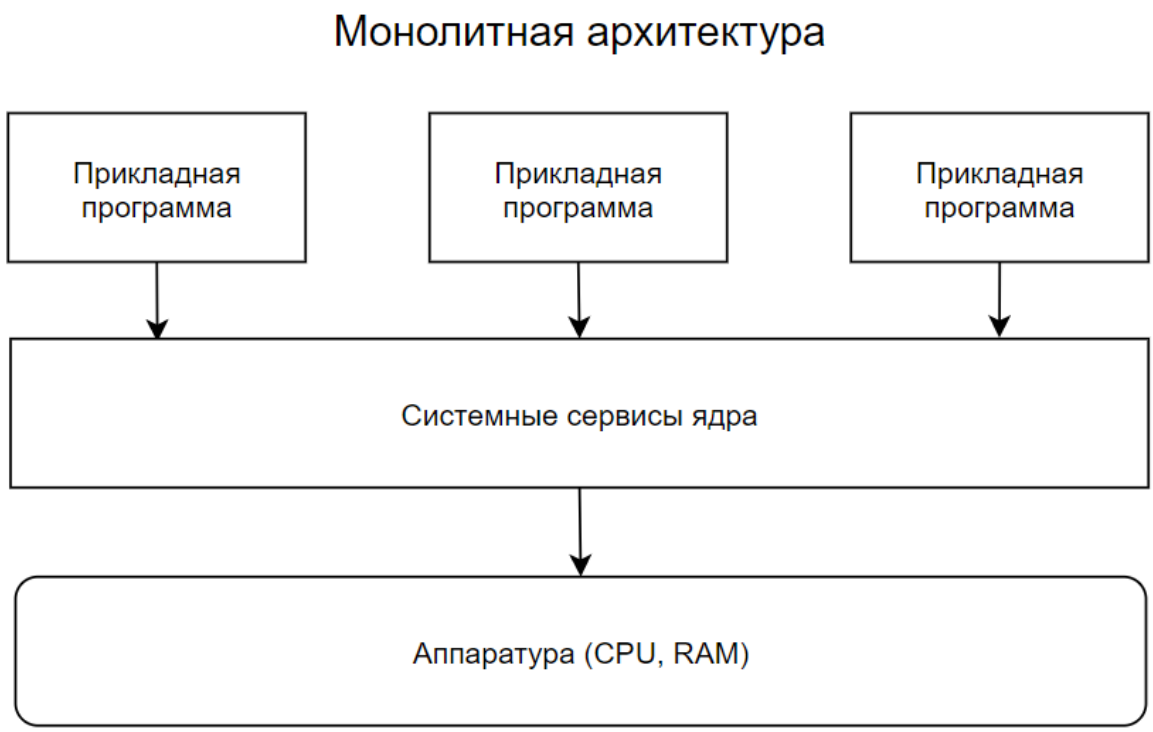

Рисунок 1.7 – Монолитная архитектура

Достоинства

● Одним из преимуществ монолита является то, что его реализация довольно проста. Интеграция бизнес-логики с необходимым функционалом происходит, практически, сразу после начала цикла разработки.

● Простота реализации сквозных тестов.

● Простота развертывания и горизонтального масштабирования.

Недостатки

● Монолитная архитектура подвержена в превращение, так называемого «большого комка грязи». Нарушения архитектуры неизменно приводят к краху системы.

● Увеличение «технического долга».

● Невозможность масштабировать отдельные части системы.

● Отсутствие изоляции компонентов системы.

● Высокая связность компонентов системы.

1.3.3 Критерии «хорошей» архитектуры

Следствием хорошей архитектуры является простота поддержки программного обеспечения, доступность расширения, тестирования и отладки. Далее автор приводит список критериев, характеризующий архитектуру надежной [6]:

● Эффективность системы. Одним из первых показателей надежной архитектуры является ее эффективность. Если система бесперебойно выполняет задачи бизнеса можно сделать вывод и при этом постоянно растет функционально, можно сделать вывод, что в данной системе грамотно спроектированная архитектура.

● Гибкость. Специфика разработки такова, что любая система будет подвержена доработке, изменениям. Если внедрение нового функционала происходит быстро и без финансовых потерь - архитекруа обладает необходимой гибкость, она конкуретноспособна.

● Способность к расширению: модуль должен обладать возможность быть расширенным, но при этом нежелательно его модифицировать. Данное требование представлено в принципах SOLID под литерой «O» (Open-Closed Principle).

● Масштабируемость процесса разработки. Возможность сократить срок разработки за счёт добавления к проекту новых людей. Архитектура должна позволять распараллелить процесс разработки, так чтобы множество людей могли работать над программой одновременно.

● Возможность тестирования кода.

● Переиспользование. Архитектура модулей приложения должна позволять переиспользовать их повторно в других системах.

● Сопровождаемость кодовой базы. Т.к. написание кода в большом проекте - командная работа, приложение должно универсально читаться любыми разработчиками. Исчерпывающая документация, соблюдение стандартом написания и проектирования лучше всего решают эту задачу.

1.3.4 Критерии «плохой» архитектуры

● Неподвижность. Отсутствие возможности переиспользовать код, плохо поддается рефакторингу, невозможность миграции на другую технологию.

● Жесткость. Компоненты имеют сильную связанность между собой, изменяя один - обязательно необходимо изменить большое количество других.

● Хрупкость. Небольшие изменения функционала приводят к поломкам всей системы. «Проценты» по кредиту стоят больше, чем само изменение.

1.4 Управление процессом проектирования архитектуры IT-предприятия

Проектирование архитектуры ПО - это работа, требующая определенной квалификации и опыта. Обычно эта работа делегируется человеку, должность которого называется архитектор приложений. В зависимости от масштабов предприятия, его целей и задач выбирается оптимальная стратегия вложения ресурсов в процесс проектирование архитектуры, в его управление.

В частном случае автор рассмотрит пример, когда управление проектирование архитектуры ПО было необходимо и к чему приводит пренебрежение в данной ситуации:

Малому предприятию требуется внедрение электронных счетов. Бизнес-процесс довольно емкий. Руководитель, отвечающий за реализацию данного процесса не является техническим специалистом и переносит требования разработчикам. При этом, в команде разработчиков нет архитектора приложений. Изначальные требования звучали? как возможность хранить электронные шаблоны документов и организовать электронную подпись. Разработчики не предполагают дальнейшего развития этой систему ввиду отсутствия опыта, а также подгоняемые сроками от руководство быстро реализуют необходимый функционал. Все прекрасно функционирует, однако, через месяц поступает требование расширить спектр форматом документов и добавить работу с таблицами. Вклиниваюсь в систему и добавляя новый функционал разработчики заметили, что теперь не только необходимо реализовать новые бизнес-требования, но и не сломать старые. С каждой подобной итерацией реализация нового функционала происходила все дольше и, соответственно, дороже.

Данный феномен называется «технический долг». Далее рассматривается его определение и причины появления.

1.5 «Технический долг» и почему он появляется

1.5.1 Понятие «технического долга»

Любая система программного обеспечения имеет свойство превращаться в так называемый «большой комок грязи» (термин, введенный Мартином Фаулером) [3]. Характеризуется это человеческим фактором, скоростью изменения системы.

Технический долг – это метафора, предложенная У. Каннингемом. Метафора объясняет, как относиться к этому мусору, приводя пример кредита в финансовом секторе [7]. Когда мы начинаем добавлять функционал к нашей

системе без должного внимания к архитектуре мы, тем самым, как бы добавляем проценты по кредиту.

1.5.2 Причины появления технического долга

● Требование немедленного выполнения задачи со стороны бизнеса. Зачастую бизнес не понимает важности архитектуры и понятия технического долга. Ему нужен работающий и дешевый продукт, который появится на свет как можно раньше. Это приводит к появлению «грязного кода», «костылей» и заплаток.

● Пренебрежение или непонимание последствий «технического долга». Бизнес не желает или не понимает, что технический долг начисляет проценты. Это способствует замедлению скорости разработки.

● Отсутствие unit-тестов. Отсутствие такого инструмента ведет к появлению мелких ошибок и, как следствие, исправлений в боевой среде. В определенный момент это может вызывать катастрофические последствия.

● Отсутствие кодовой документации ведет к тому, что в проекте появляются некие «хозяева» проекта. Только они разбираются в нем и при увольнении таких сотрудников проект рискует остановиться. Также усложняется привлечение новых специалистов к проектной работе.

● Нарушение коммуникаций между между членами продуктовой команды.

● Пренебрежение рефакторингом.

● Отсутствие контроля за соблюдением стандартов. Новые участники проектов пишут код как хотят, никто не следит за соблюдением стандартов. Тождественно, что такая кодовая база придет к виду «большой комок грязи».

● Отсутствие технических компетенций. Участникам команды не хватает квалификации грамотно проектировать систему и писать код.

1.6 Волатильность финансовых издержек предприятия при проектировании ПО

При расширении функционала информационной системы неизменно будут расти финансовые издержки на поддержку ее работоспособности, причем экспоненциально.

Вначале, когда система еще мала, скорость разработки на MVP (минимально-жизнеспособный продукт), может показаться, что затраты на проектирование не оправданы. В этом тезисе есть доля истины, если ваш проект не будет развиваться далее. На рисунке 1.8 изобразим график затрат на разработку системы без проектирования архитектуры. Ось абсцисс будет отображать время жизни ПО, а по оси ординат - стоимость затрат на разработку нового функционала.

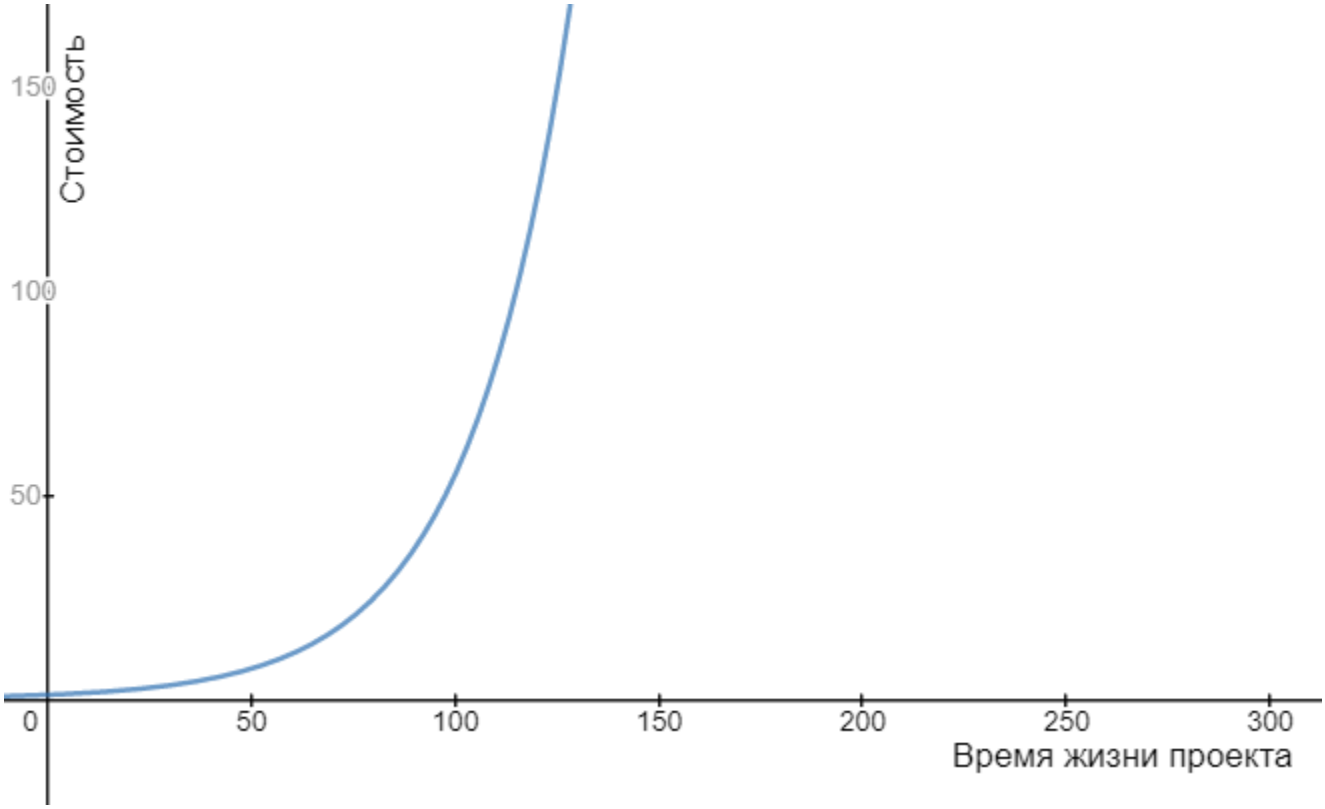

Рисунок 1.8 – График издержек предприятия на разработку нового функционала без проектирования архитектуры.

Как видно на выше представленном графике, со временем финансовые издержки увеличиваются до такого уровня, что разработка небольшого функционала вызывает непомерные траты и усилия. Объясняется это просто: при

разработке без четко выверенных правил нарушается консистентность системы. Более бытовым примером может служить постройка дома. Построить шалаш можно без инструкции и из подручных материалов. Но если вы строите многоэтажное сооружение, то вам придется следовать четкому своду правил, ГОСТам и технической спецификации. В противном случае, в определенный момент такая незначительная доработка, как проведение проводки на одном из этажей может стать в колоссальные расходы для строительной компании.

Одна из главных задач проектирования архитектуры ПО – уменьшение финансовых затрат предприятия на развитие системы. Известна практика, что, вложившись вначале больше, на длинном лаге это принесет в разы больше. Рассмотрим в сравнении ситуации, когда на этапе проектирования было заложено больше времени и финансов и к чему это привело на рисунке 1.9.

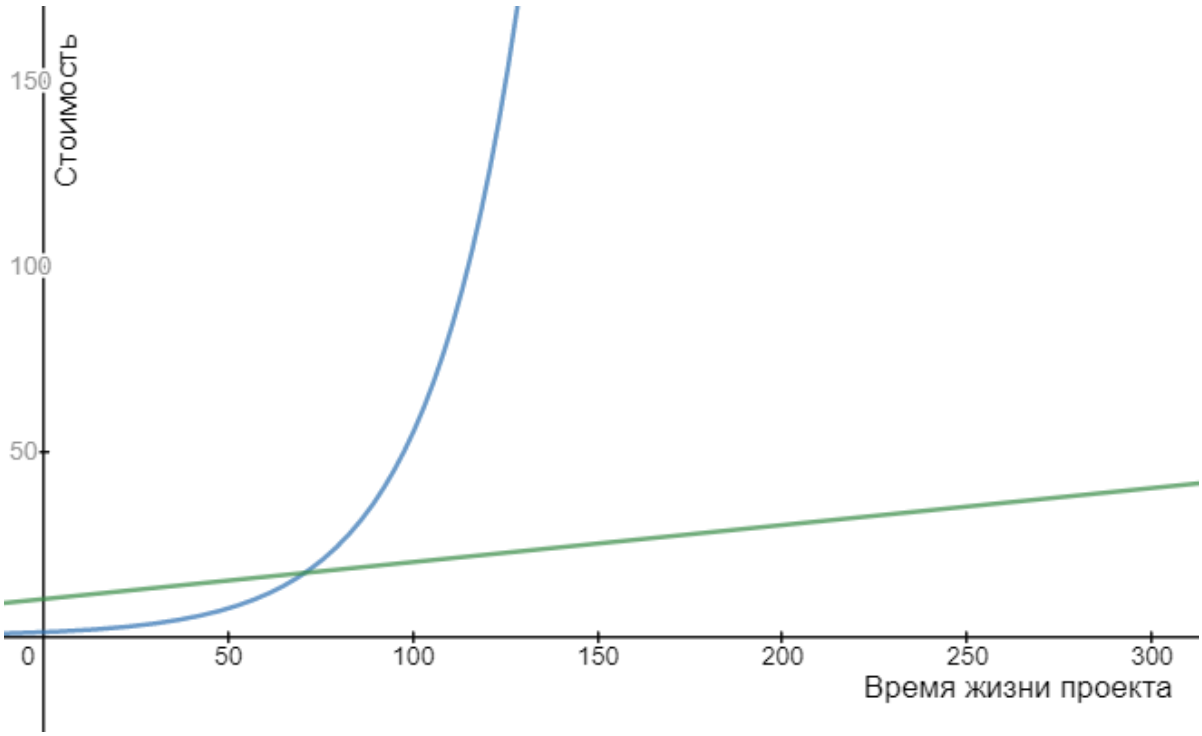

Рисунок 1.9 – Сравнение финансовых издержек при различных подходах проектирования архитектуры ПО

Можно наблюдать, что, вложившись в начале мы выигрываем после некоторой, так называемой, «точки невозврата», после которой на экспоненциальном графике поддерживать ПО становится невозможным. Главный

вопрос при выборе стратегии сводится к определению этой точки. Если информационная система перестанет развиваться до этой точки - смело начинаем с MVP. Но если ожидается длительная поддержка и развитие, то имеет смысл вложить средства и избежать финансового краха.

Выводы по разделу один

Программное обеспечение – сложный многоуровневый организм, который требует порядка, четких правил, у которого есть свои жизненные циклы. В зависимости от этапа жизненного цикла ПО требуется взаимодействие различных подразделений предприятия, вложение финансовых, технологических, материальных ценностей.

Автор рассмотрел и выделил такие этапы жизненного цикла по, как подготовка, проектирование, создание и поддержка.

Рассмотрев понятие «архитектура программного обеспечения» и выделив ее типы можно сделать выводы, что каждая из них имеет свои плюсы и минусы и выбирать ее следует осторожно, рассматривая частные случаи. Несмотря на явные различия между типами архитектуры, можно сделать выводы и ввести такие понятия как «хорошая архитектура» и «плохая архитектура», выделив признаки обоих.

Также автором был рассмотрен термин «технический долг» и выявлены основные причины его появления. На основании теоретических данных были спроецированы с помощью графиков финансовые издержки предприятия на создание программного обеспечения при академическом подходе к проектированию архитектуры в сравнении с издержками предприятия, с подходом «как получится». Графики четко показывают отличие и отвечают на вопрос: «Стоит ли вкладывать деньги на проектирование архитектуры?» Да, стоит, если программное обеспечение ожидает большого развития и роста функционала.

Резюмируя данный раздел, автор отмечает что архитектура программного обеспечения очень важна и по значимости может быть сравнима с проектирование

космической станции – ошибки, допущенные на этапе проектирования выльются в катастрофические последствия.

2. ОБЗОР СОВРЕМЕННЫХ ПОДХОДОВ К РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И МЕТОДОЛОГИЙ ПРОЕКТИРОВАНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ

2.1 Основные модели разработки ПО

Разработка программного обеспечения классифицируется с помощью моделей, описывающих стадии жизненного цикла. Методологии, в свою очередь, характеризуются наборами методов, с помощью которых реализуется управление разработкой.

Классические модели разработки ПО [4]

- Code and fix – модель кодирования и устранения ошибок;
- Waterfall Model – каскадная модель, или «водопад»;
- V-model – V-образная модель, разработка через тестирование;
- Incremental Model – инкрементная модель;
- Iterative Model – итеративная (или итерационная) модель;
- Spiral Model – спиральная модель;
- Chaos model – модель хаоса;
- Prototype Model – прототипная модель.

Далее автор анализирует самые популярные из них.

2.1.1 Waterfall (каскадная модель, или «водопад»)

В каскадной модели разработка делится на этапы: последующая стадия начинается после окончания предыдущей, напоминая водопад [5].

Классическая схема работы государственных учреждений. Внедрена с середины 1970х годов.

Схема каскадной модели представлена на рисунке 2.1:

Рисунок 2.1 – Каскадная модель

2.1.2 V-образная модель (разработка через тестирование)

Является модификацией каскадной модели, отличительной особенностью которой, является то, что заказчик совместно с разработчиками составляет одновременно и требования к системе и описание процесса тестирования Используется, начиная в 1980-х. Схематично представлена на рисунке 2.2

Рисунок 2.2 – V-образная модель

Преимущества V-образной модели

• Минимальное количество ошибок в архитектуре.

Недостатки V-образной модели

• Цена ошибки проектирования архитектуры по-прежнему велика.

Данная модель является отличным кандидатом для проектов, в которых важна надежность и цена ошибки колоссально высока. Пример может служить высокоточное оборудование по лазерной коррекции зрения.

2.1.3 Инкрементная модель

Данная модель олицетворяет разработкиу системы по частям [6]. Используется очень давно, с середины в 1930х годов. Рассмотрим её на примере создания форума.

Заказчик озвучивает требования, что хочет запустить чат систему и оформляет исчерпывающее техническое задание. Разработчики предлогают реализовать основной функционал – пользовательскую страницу с персональными данными и чат. Далее приступить к тестированию на пользователях.

Команда разработчиков презентует окончательный продукт заказчику и отправляет его на рынок. При условии удовлетворенности как заказчиком, так и пользователями чат системой, работа над ней продолжается по частям.

Разработчики параллельно реализовывают функционал для загрузки фотокарточек, отправку документов, просмотр видео и других действий, обговоренных с заказчиком. Инкрементируя функционал, разработчики добавится поставленной в техническом задании цели.

На рисунке 2.3 представлена инкрементная модель.

Рисунок 2.3 Инкрементная модель

Преимущества инкрементной модели:

● Декомпозиция функционала приводит, как следствие, к малым финансовым вложениям для старта проекта.

● Возможность моментального получения качественной обратной связи от пользователей и, как следствие, возможность оперативного обновления технического задания.

● Цена ошибки значительно меньше, чем в каскадной и V-образной модели.

Недостатки инкрементной модели

● Отсутствие согласованности между командами разработчиков, ответственных за разные компоненты системы.

● Повышенные требования и обязанности менеджера проектов.

Инкрементная модель успешно интегрируется в проекты, техническое задание которых прописано уже на старте, а скорость разработки продукта является приоритетным фактором.

2.1.4 Спиральная модель

Данная модель характеризуется серьезным анализом рисков проекта, а также итерационностью его выполнения. Каждая следующая стадия, зависит от предыдущей итерации. Схема изображена на рисунке 2.4

Рисунок 2.4 – Спиральная модель

Преимущества спиральной модели:

● Проработка рисков является приоритетным фактором.

Недостатки спиральной модели:

● Начальный этап может затянуться – бесконечное совершенствование прототипа.

● Цена разработки.

2.1.5 «Agile Model» (гибкая методология разработки)

В «гибкой» методологии разработки после каждой итерации заказчик может наблюдать результат и понимать, удовлетворяет он его или нет. Это одно из преимуществ гибкой модели. К ее недостаткам относят то, что из-за отсутствия конкретных формулировок результатов сложно оценить трудозатраты и стоимость, требуемые на разработку. Экстремальное программирование (XP) является одним из наиболее известных применений гибкой модели на практике.

Достоинства:

- Гибкость проекта под требования.
- Декомпозиция задач способствует мотивации работников (приятнее работать и выполнять маленькие подзадачи).
- Повышенная клиентоориентированность.
- Продуктивное взаимодействие команды, быстрое получение обратной связи.

Недостатки:

- Работа в команде для специалистов, предпочитающих одиночную работу.
- Повышенный порог вхождения.

2.1.6 Сравнительный анализ моделей разработки

Для наглядного отображения всех плюсов и минусов различных моделей разработки автор отображает сводную таблицу 1, в которой лаконично представлены достоинства и недостатки каждой из вышеперечисленных моделей:

Таблица 1 – Сравнение моделей разработки программного обеспечения.

Waterfall	Локаничность структуры управления	Отсутствует возможность корректировки требований
	Документация задач	Невозможность правок в процессе разработки
	Четкое прогнозирование расходов	Отсутсвие взаимодействия продуктивной команды и заказчика
V-образная модель	Минимальное количество ошибок в архитектуре	Критическая цена ошибки
	-	Отсутствует возможность корректировки требований
Инкрементная модель	Минимальные финансовые затраты при старте	Децентрализованность разработки
	Быстрая обратная связь	Зацикленность на деталях продукта

Продолжение таблицы 1.

	Дешевая цена ошибки	Сложность выполнения формального критического анализа
	-	Увеличение "технического долга" за счет желания сделать быстрее
Спиральная модель	Диверсификация рисков при проектировании	Дороговизна интеграции и поддержки
	Документация процесса разработки	Невозможность внедрения в существующую систему
	Возможность подстраиваться под требования	Риск оставаться на текущем этапе (спирали), постоянно улучшая его.

Продолжение таблицы 1.

Agile	Гибкость проекта под требования.	Работа в команде для специалистов, предпочитающих одиночную работу.
	Декомпозиция задач способствует мотивации работников (приятнее работать и выполнять маленькие подзадачи).	Повышенный порог вхождения.
	Повышенная клиентоориентированность.	-
	Продуктивное взаимодействие команды, быстрое получение обратной связи.	-

Рассмотрев современные модели разработки, проанализировав плюсы и минусы каждой из них, автор считает, что оптимальной моделью для разработки ПО на предприятии является agile модель – гибкая разработка. Далее будут рассмотрены методологии разработки на базе этой модели.

2.2 Методологии проектирования архитектуры информационных систем на базе Agile

2.2.1 Test-driven development

TDD, test-driven development – это методология разработки с помощью специального инструмента, который называется «uniy-тестирование». Данная методология базируется на повторении некоторого алгоритма действий: сначала пишет функцию-тест, которая покрывает еще нереализованный бизнес-сценарий, далее реализуется непосредственно код, описывающий желаемое поведение системы и позволяющий пройти тес. Финальной итерацией служит рефакторинг написанного кода.

Тестирование ПО – процедура, позволяющая релевантно оценить работоспособность кода и корректность его работы. Во время проведения unit-тестов приложению передаются на вход данные и запрашивается выполнение определенно команды. Далее происходит проверка полученных результатов и они сравниваются с некоторым эталоном. Запуск тестов является автоматизируемым процессом.

Данная методология разработки через ставит главной задачей организацию автоматического тестирования посредством написания модульных, функциональных и интеграционных тестов.

Цикл разработки по TDD:

● Добавить тест для новой (еще не реализованной) функциональности или для воспроизведения существующего бага

● Запустить все тесты и убедиться, что новый тест не проходит

● Написать код, который обеспечит прохождение теста:

● Запустить тесты и убедиться, что они все прошли успешно: прохождение нового теста подтверждает реализацию нового функционала или исправление существующей ошибки, а прохождение остальных позволяет удостовериться, что ранее реализованный функционал работает по-прежнему корректно.

● Заняться рефакторингом и оптимизацией – улучшение сопровождаемости и быстродействия целесообразно осуществлять уже после того, как получилось добиться проверяемой работоспособности

● Перезапустить тесты и убедиться, что они все ещё проходят успешно

● Повторить цикл

Эта методология позволяет добиться создания пригодного для автоматического тестирования приложения и очень хорошего покрытия кода тестами, так как ТЗ переводится на язык автоматических тестов, то есть всё, что программа должна делать, проверяется. Также TDD часто упрощает программную реализацию: так как исключается избыточность – если компонент проходит тест, то он считается готовым. Если же существующие тесты проходят, но работает компонент не так, как ожидается, то это значит, что тесты пока не отражают всех требований и это повод добавить новые тесты.

Как следствие, архитектура программного обеспечения, разработанная посредством данной методологии, является чистой, устойчивой к масштабированию.

2.2.2 Behaviour-driven development

BDD – это ответвление методологии разработки через тестирование.

Базисом данной методологии является коллаборация технических интересов и бизнес-сценариев, как следствие позволяя менеджерам и разработчикам говорить на одном языке. Реализуется это посредством внедрения естественного языка в предметно-ориентированный, ссылаясь на определенные поведения, приводящие к нужному результату.

Достоинства BDD:

● Читаемость тестов увеличивается

● Гибкость тестов под изменение бизнес-логики.

● Прототипы тестов могут писать не только программисты.

● Результаты тестирования "человечные".

● Кроссплатформенность тестов.

Недостатки BDD:

● сложность соблюдения стандартов написания кода по данной методологии

● цена разработки. Стоимость в переводе на трудочасы разработчиков увеличивается в разы и не всегда оправданно

Считается, что данный подход эффективен, когда предметная область, в которой работает программный продукт, описывается очень комплексно.

2.2.3. Domain-driven design

Предметно-ориентированное проектирование (далее DDD) – это набор правил и принципов, позволяющих оптимально подходить к процессу проектирования информационных систем. В конечно итоге сводится к реализации абстракций, позволяющих охарактеризовать предметную область. Данные абстракции включают в себя бизнес-логику, связи между объектами области применения продукта и кодом.

Зачем же нужен DDD? В первую очередь - это упрощение бизнес-логики. По аналогии с четырьмя уравнениями Максвелла, описывающими всю классическую теорию электромагнетизма, DDD позволяет проще описать частный сложный бизнес-процесс.

Также DDD позволяет уйти от лишней документации.

DDD – это набор правил, которые позволяют принимать правильные проектные решения. Данный подход позволяет значительно ускорить процесс проектирования программного обеспечения в незнакомой предметной области.

Методологи DDD наиболее актуальна, когда разработчик не является экспертом предметной области. Примером может служить разработка ПО сопровождающего клинические операции: разработчик не знает тонкостей данного домена (области).

Данный термин был впервые введен Э. Эвансом в его книге с таким же названием «Domain-Driven Design»[1].

2.2.4 Сравнительный анализ методологий проектирования

Для наглядного отображения всех плюсов и минусов различных методологий разработки автор отображает сводную таблицу 2, в которой лаконично представлены достоинства и недостатки каждой:

Таблица 2 – Сравнение методологий разработки программного обеспечения.

	Достоинства	Недостатки
TDD	Уменьшение времени отладки кода	Несовместимость с крупной архитектурой
	Улучшенный UI	Отсутствие универсальности
	Уменьшает время разработки тестов	Дополнительные, излишние затраты
BDD	Читаемость тестов увеличивается	Излишняя абстракция в коде
	Гибкость тестов под изменение бизнес-логики.	Высокий порог вхождения

Продолжение таблицы 2.

	Прототипы тестов могут писать не только программисты.	Повышенная цена разработки
	Результаты тестирования "человечные".	Сложность интеграции с другими системами
	Кроссплатформенность тестов.	
DDD	Наличие единого языка	Высокий порог вхождения
	Упрощенная возможность отладки	Повышенная цена разработки
	Возможность масштабирования системы	-
	Приближенность бизнес-логики к коду	-
	Простота постановки задачи	-

Рассмотрев основные методологии по разработке ПО, а также проанализировав достоинства и недостатки каждой, учитывая специфику разработки на IT-предприятии, была выбрана методология DDD для дальнейшего практического внедрения. Далее будет рассмотрен глоссарий методологии DDD.

2.3 Глоссарий Domain Driven Design

Основной задачей предметно-ориентированного проектирования (DDD) является применение модели для решения задач, стоящих перед приложением.

Работоспособность модели добивается посредством фильтрации потока хаотичной информации за счет переработки знаний группы разработчиков. Так появляется на свет работоспособная доменная модель.

Проектирование по модели (Model-Driven Design) создает тесную связь между моделью и реализацией. А единый язык (Ubiquitous Language) является каналом, по которому вся нужна информация расходится между программистами, специалистами в предметной области и непосредственно программным продуктом.

В результате возникает программа, богатые функциональные возможности которой базируются на фундаментальном понимании сути предметной области, в которой она работает

Для эффективной работы команды, состоящей из экспертов предметной области, разработчиков, руководителей и инвесторов необходим единый язык, на котором бизнес мог бы общаться с разработчиками и наоборот. Мартин Фаулер придумал данный список шаблонов (паттернов), для наиболее четкого понимания задач и границ ответственности.

Единый язык (Ubiquitous Language).

Является основным паттерном данной методологии. Является полноценным языком, эксплуатируемым целой командой аналитиков, программистов и всех участвующих в реализации системы. Для реализации данного шаблона проектирования, Вон Вернон советует использовать такие способы:

● Использование диаграмм концептуального и физического уровня, реализовывая на них названия и бизнес-сценарии. В отличии от UML-диаграмм являются неформальными.

● Необходим глоссарий с простыми терминами. Также реализовывается глоссарий с альтернативными терминами с оценкой их актуальности. Таким образом разрабатываются устойчивые словосочетания единого языка.

● Использование исчерпывающей документации.

● Обсуждение готовых фраз с остальными членами команды, которые не могут сразу освоить глоссарий или другие письменные документы.

В предметной области, обсуждая модель применения вакцины от гриппа в виде кода, участники говорят: «Фельдшеры назначают вакцины от гриппа в стандартных дозах». Возможные варианты развития событий представлены в таблице 3:

Таблица 3 – Варианты развития событий.

Возможные точки зрения	Итоговый код
«Кого это волнует? Просто программируйте»	*Patient.setShotType(ShotTypes.TPE_FLU);* *Patient.setDose(dose);*
«Мы делаем пациентам прививку от гриппа»	*Patient.giveFluShot();*
«Медсестры назначают вакцины от гриппа в стандартных дозах»	*Vaccinet vaccine =* *vaccines.standartAdultFluDose();*

Ограниченный контекст (Bounded context).

Это второе по значимости свойство DDD после единого языка. Оба эти понятия взаимосвязаны и не могут существовать друг без друга.

Ограниченный контекст является границей, внутри который взаимодействуют доменные модели. Данная граница преобразует единый язык в модель программы.

В каждом ограниченном контексте существует только один единый язык.

Ограниченные контексты являются относительно небольшими, меньше чем может показаться на первый взгляд. ограниченный контекст достаточно велик только для единого языка изолированной предметной области, но не больше.

Предметная область, предметная подобласть, смысловое ядро.

Предметная область (Domain) – это то, что делает организация, и среда, в которой она это делает. Разработчик программного обеспечения для организации обязательно работает в ее предметной области. Следует понимать, что при разработке модели предметной области необходимо сосредоточиться в определенной подобласти, так как практически невозможно создать единственную, всеобъемлющую модель бизнеса даже умеренно сложной организации. Очень важно разделять модели на логические разделенные предметные подобласти (Subdomain) всей предметной области, согласно их фактической функциональности. Подобласти позволяют быстрее определить разные части предметной области, необходимые для решения конкретной задачи.

Также необходимо уметь определять смысловое ядро (Core Domain). Это очень важный аспект подхода DDD. Смысловое ядро – это подобласть, имеющая первостепенное значение для организации. Со стратегической точки зрения бизнес должен выделяться своим смысловым ядром. Большинство DDD проектов сосредоточены именно на смысловом ядре. Лучшие разработчики и эксперты должны быть задействованы именно в этой подобласти. Большинство инвестиций должны быть направлены именно сюда для достижения преимущества для бизнеса и получения наибольшей прибыли.

Пространство задач и пространство решений.

Предметные области состоят из пространства задач и пространства решений. Пространство задач позволяет думать о стратегической бизнес проблеме, которая должна быть решена, а пространство решений, сосредоточится на том, как реализуется программное обеспечение, чтобы решить бизнес проблему.

Пространство задач – части предметной области, которые необходимы, чтобы создать смысловое ядро. Это комбинация смыслового ядра и подобластей, которое это ядро должно использовать.

Пространство решений – один или несколько ограниченных контекстов, набор конкретных моделей программного обеспечения. Разработанный ограниченный контекст – это конкретное решение, представление реализации.

Идеальным вариантом является обеспечение однозначного соответствия между подобластями и ограниченными контекстами. Таким образом, объединяются пространство задач и пространство решений, выделяются модели предметной области в четко определенные области в зависимости от поставленных целей. Если система не разрабатывается с нуля, она часто представляет собой большой комок грязи, где подобласти пересекаются с ограниченными контекстами.

В качестве примера в книге Вона Вернона приводится смысловое ядро для небольшой компании, которая занимается розничной продажей в Интернете. Любой интернет-магазин может улучшить свое положение, если будет использовать механизм прогноза, который будет анализировать историю продаж и данные о запасах для получения прогноза спроса и конкретных объемов оптимальных запасов. (Компания не должна тратить деньги на товары, которые не имеют спроса и занимают дополнительную складскую площадь.) Именно это смысловое ядро делает организацию гораздо более конкурентоспособной, способной быстро идентифицировать выгодные сделки и гарантировать необходимый уровень запасов.

Пространство задач состоит из смыслового ядро и подобластей, связанных с закупкой и запасами, и которые отображены на рисунке 2.4:

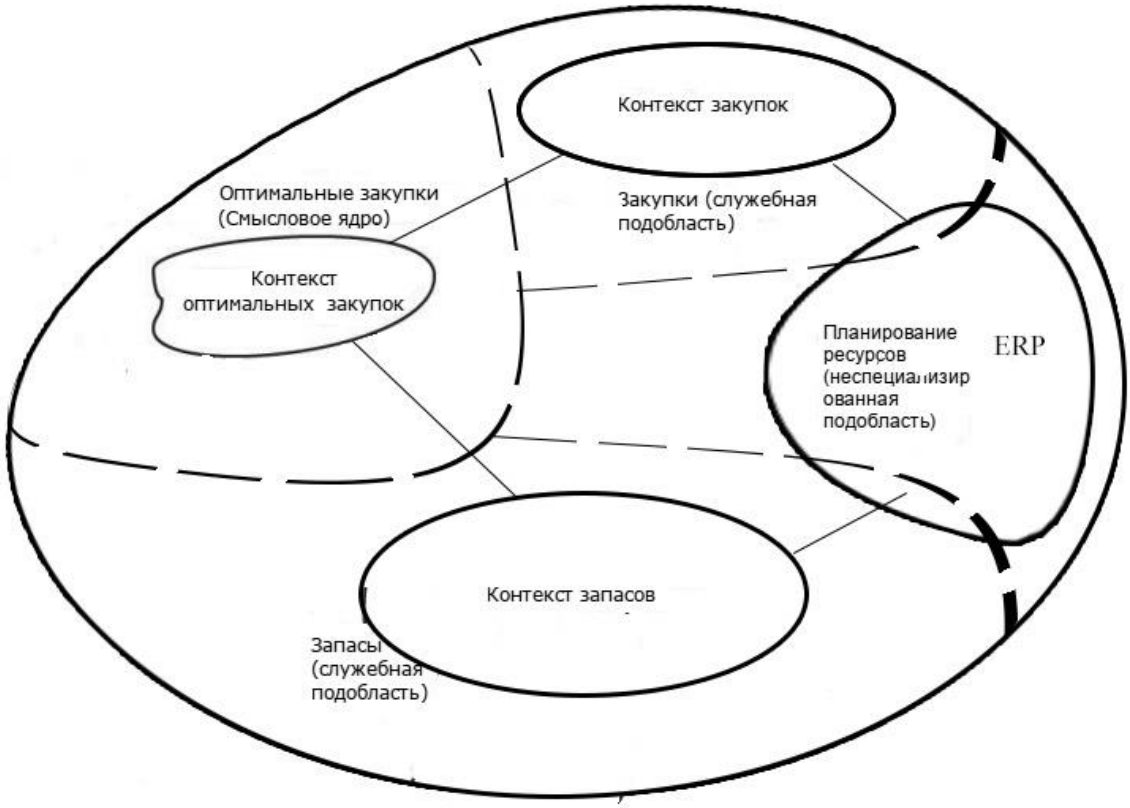

Рисунок 2.4 – Пространство задач.

Карта контекстов.

Следуя подходу DDD, определенная команда должна создать собственную карту, которая отражает пространство решений, в которой находится эта команда. Эта карта состоит из ограниченных контекстов, а также интеграционных связей между ними. Пример продемонстрируем на рисунке 2.5:

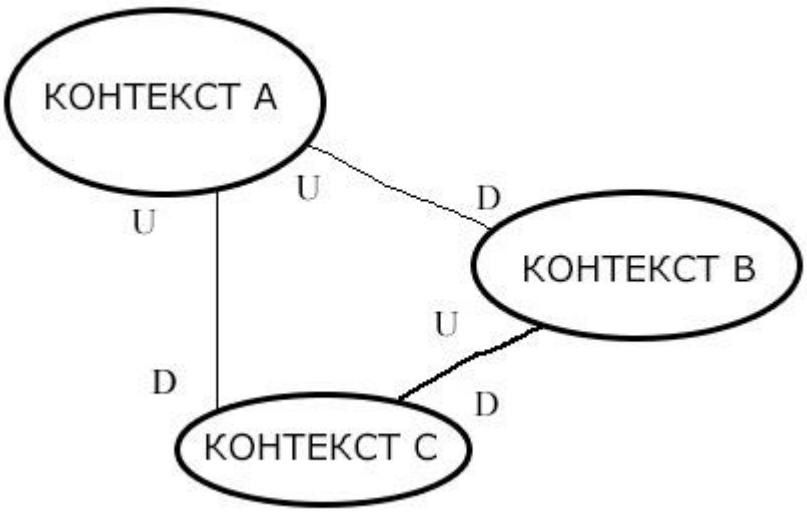

Рисунок 2.5 – Карта контекстов.

Эта карта контекстов (Context map) отображает текущее положение дел, а не то, что будет в будущем. Необходимо избегать формальностей в процессе создания карты. Слишком большое количество деталей только мешает процессу создания.

Естественный ход событий – совпадение границ контекстов с организационным делением команды. Люди, работающие вместе, разделяют один общий контекст модели.

После создания предварительной карты контекстов, ее можно детализировать путем определения отношений между контекстами [26].

Существуют такие отношения между ограниченными контекстами и отдельными командами проекта:

● Партнерство (Partnership). Когда команды в двух контекстах достигают успеха и терпят неудачу вместе, возникает отношение сотрудничества. Они должны сотрудничать в процессе эволюции своих интерфейсов, чтобы учитывать потребности обеих систем.

● Общее ядро (Shared kernel). Общая часть модели и кода образует тесную взаимосвязь. Обозначается четкая граница подмножества модели предметной области, которую команды согласны считать общей. Ядро должно быть маленьким. Оно не может изменяться без консультации с другой командой. Необходимо согласовывать единый язык команд.

● Разработка заказчки-поставщик (Customer-supplier development). Когда две команды находятся в отношении «нижестоящий и вышестоящий», и команды вышестоящие учитывают приоритеты нижестоящих команд.

● Конформист (Conformist). Когда две команды находятся в отношении «вышестоящий и нижестоящий», причем вышестоящая команда не имеет причин учитывать потребности нижестоящей команды. Нижестоящая команда учитывает сложность трансляции между ограниченными контекстами, беспрекословно подчиняясь модели вышестоящей команды.

● Предохранительный уровень (Anticorruption layer). Если управление и коммуникация не соответствуют общему ядру, партнеру, или отношению «Заказчик-поставщик», то трансляция является сложной. Нижестоящий клиент должен создать изолирующий слой, чтобы обеспечить свою систему вышестоящей системы в терминах своей модели предметной области. Этот уровень общается с другой системой с помощью существующего интерфейса, не требуя или почти не требуя модификаций другой системы.

● Служба с открытым протоколом (Open host service). Определяется протокол, который предоставляет доступ к системе как к набору служб. Для учета новых требований интеграции этот протокол расширяется и уточняется.

● Общедоступный язык (Published language). Трансляция между моделями двух ограниченных контекстов требует общего языка. В качестве среды для

коммуникации используется хорошо документированный общий язык, который может выразить необходимую информацию о предметной области, выполняя при необходимости перевод информации с другого языка на этот.

● Отдельное существование (Separate ways). Если между двумя наборами функциональных возможностей нет важного отношения, их можно полностью отсоединить друг от друга. Интеграция всегда дорого стоит, а выгоды бывают незначительны.

● Большой комок грязи (Big ball of mud). Существуют части системы, в которых модели перемешаны, а границы стерты. Необходимо нарисовать границу такой смеси и назвать ее большой комок грязи.

Приведем пример разработки карты контекстов и использования ее в банковском секторе:

На рисунке 2.6 отобразим простую карту контекстов, обозначив границы и связи между ограниченными контекстами учетной записи пользователя и системы персонального управления финансами:

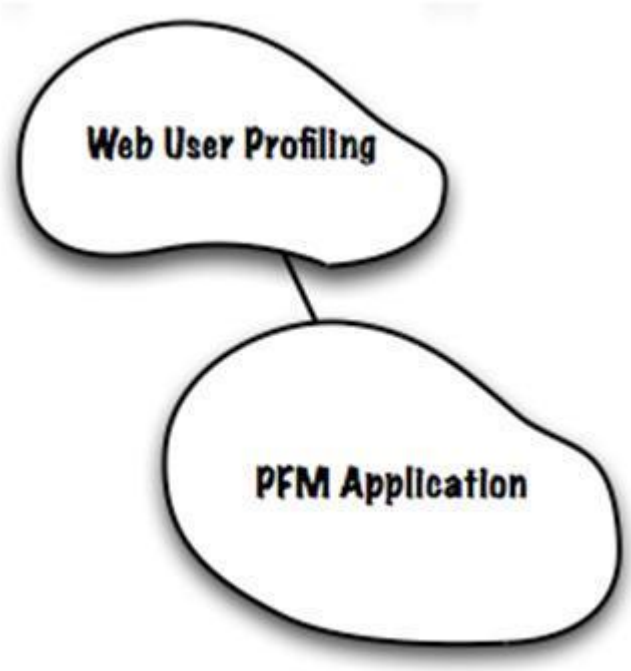

Рисунок 2.6 – Простая карта контекстов

В этих двух контекстах есть различия в концепциях с одинаковым названием. Например, Account в Web User Profiling – это учетная запись

пользователя (логин и пароль). В то же время, для PFM Application (персональное управление финансами) – это сводка, описывающая текущее состояние клиента с точки зрения банка. Иногда, как было указано выше, одна и та же концепция может использоваться в абсолютно разных контекстах, тем самым для них необходимо определить разные модели.

Например, PayeeAccount – это тот же BankingAccount, но с другим поведением (нельзя получить баланс). Таким образом будет создан отдельный контекст учета расходов (expense tracking). Также отдельно, в приведенном примере, создается контекст онлайн сервисов банка (on-line banking services) (такие сервисы, например, как распечатка выписок банка). Наглядная демонстрация этого представлена на рисунке 2.7:

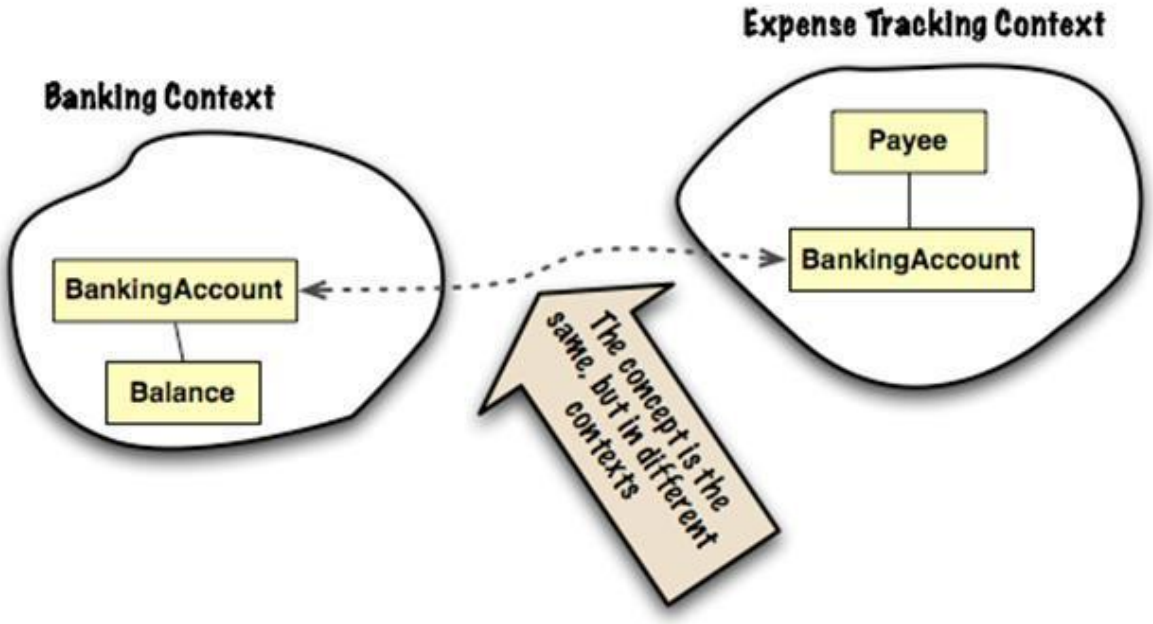

Рисунок 2.7 – Наглядная демонстрация применения карты контекстов.

После первого шага создания карты, можно детализировать все отношения. У каждого отношения есть направление. Вышестоящие (upstream) влияют на нижестоящие (downstream), но не факт, что обратное верно. Соберем воедино все карты и детализируем их на рисунке 2.8:

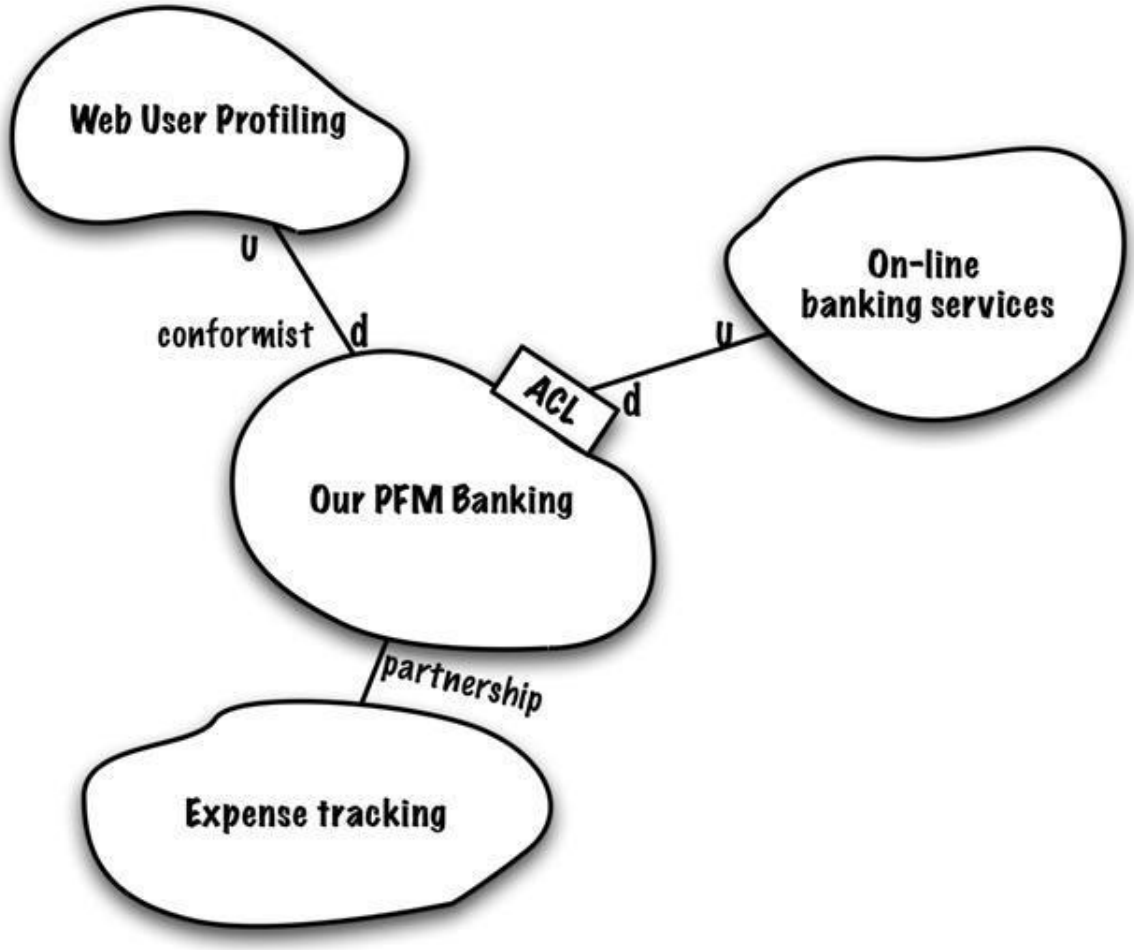

Рисунок 2.8 – Детализация карты контекстов

Так как контекст профайлов веб-пользователей используется, как готовый внешний модуль и он поставляется «как есть», здесь устанавливается отношение конформист (нижестоящий подчиняется вышестоящему).

В случае с контекстом учетов расходов, то здесь лучше всего подходит отношение партнерства, так как существуют общие цели и концепции, но нет направления отношения.

Это простой пример карты контекстов, на деле они бывают намного сложней.

Выводы по разделу два

В данном разделе было рассмотрено:

- основные модели разработки ПО;
- методологии проектирования архитектуры ПО;
- сделан выбор методологии и рассмотрен ее глоссарий;

Спектр методологий и моделей разработки программного обеспечения обширен и впечатляет своим разнообразием. При выборе методологии необходимо рассматривать частную задачу бизнеса и руководствоваться многими факторами.

Однако, в большинстве случаев, автоматизация предприятия подразумевает постоянное развитие функционала и изменений бизнес-условий. Можно сформулировать эту ситуацию фразой: «бизнес диктует – разработчик исправляет». Данная ситуация характерна для разработки на IT-предприятии. Сталкиваясь с проблемами масштабирования функционала системы, когда данные изменения ведут к колоссальным потерям, можно сделать вывод, что данной системе необходимо четкое проектирование архитектуры.

Ввиду рассмотренной информации, автор делает вывод, что современные информационные системы возможно проектировать лишь средствами гибкой методологии (agile). Требования слишком часто меняются, составление технического заданий одного и навсегда подразумевает крах системы. Автор на основании проведенного анализа и сравнительных характеристик выбирает agile-разработку, а также, использования методологии Domain Driven Design. В следующем разделе будут изучены и выдвинуты требования к архитектуре ПО.

3. РАЗРАБОТКА ТРЕБОВАНИЙ К АРХИТЕКТУРЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Как выше отмечает автор, при выборе подхода необходимо учитывать частную ситуацию, конкретный «кейс» бизнеса. Исследование данной выпускной квалификационной работы будет рассмотрено на примере реального «бизнес кейса» - автоматизация процесса установки камер у юридических и физических лиц. Далее будут разработаны различные требования к системе, автоматизирующей данный бизнес-процесс.

3.1 Архитектурные требования

При разработке больших систем выделяют множество критериев, которым стоит уделить внимание, однако главной задачей считается, все-таки, снижение сложности с помощью декомпозиции. Функционально сложная система обязана строиться из маленьких подсистем, каждая из которых, в свою очередь, состоит из частей все меньшего размера, и т.д., до тех пор, пока части системы не станут настолько атомарны и просты в понимании, насколько это возможно.

Выделим основные требования для архитектуры системы, которая автоматизирует процесс установки камер:

● Scalability (Масштабируемость). Необходимо иметь возможность расширять систему, добавляя новый функционал, разрабатывая новые модули, увеличивая при этом производительность.

● Maintainability (Ремонтопригодность). Изменив один модуль, не должно «родить» требование менять другие блоки системы.

● Swappability (Взаимозаменяемость). Блок системы легко можно заменить на другой блок.

● Unit Testing (Модульное тестирование). Модуль системы атомарен и его можно протестировать изолированно от остальных модулей.

● Reusability (Переиспользование). Наглядно объясняется в принципе программирования DRY - Don't repeat yourself (Не повторяйся). Стройте систему так, чтобы модули можно было переиспользовать.

● Maintenance (Сопровождаемость). Архитектура ПО должна быть легко сопровождаемая.

Данные требования, на самом деле, отражены в принципах проектирования SOLID. На рисунке 3.1 кратко представлены эти принципы:

Рисунок 3.1 – Принципы проектирования SOLID.

Спроецируем каждое требование подробнее на бизнес-процесс, рассматриваемый автором:

1. Установка видеокамер и сопутствующие ей процессы должны иметь возможность легко расширяться. При необходимости разработчики должны легко и без задержек интегрировать новый функционал не нарушая работоспособность системы. Это требование реализуется в принципе «открытости/закрытости» и трактуется, как: «класс должен быть открыт для расширения, но закрыт для модификации». На практике это можно представить, что компании понадобится вместе с камерами работать с близким по смыслу оборудованием - домофонами.

2. Малая связанность блоков системы позволяет реализовать требование ремонтопригодности. Принципы разделения интерфейсов и единой ответственности помогут добиться этого требования. В процессе установки видеокамер, это будет выражаться отсутствием зависимостей между мало связанными процессами (например, процесс настройки клиентского роутера не должен зависеть от сохранения камеры в таблице базы данных и наоборот).

3. К требованию взаимозаменяемости требуется подходить с особой осторожностью, ведь он прямо нарушает предыдущего требования малой связности частей системы. Однако в нашем случае необходимо разрабатывать модули так, чтобы их можно было переиспользовать.

4. Модульное тестирование будет заключаться в написании тестов к методам классов и реализовываться с помощью «моков». Независимость от базы данных, будет реализована с помощью паттерна «репозиторий». Каждый тест соответствует методу класса.

5. Сопровождаемость архитектуры выражается в возможности смены команды, а следовательно изменения взглядов. Апогеем проектирования будем

считать, что каждая новая бизнес-идея будет реализована одинаковым алгоритмом и похожим кодом.

Рассмотрев общие требования к архитектуре, проанализируем функциональные требования к системе, автоматизирующий столь сложный процесс - установка видеокамеры.

3.2 Функциональные требования системы установки видеокамер

Ниже представлены функциональные требования:

- Обеспечить надежное хранение камер;
- Обеспечить возможность удаленной настройки камер;
- Обеспечить возможность удаленной настройки роутера;
- Обеспечить хранение информации о камерах пользователя;
- Реализовать возможность управления абонентской платой абонента;
- Реализовать интерфейс управления камерами;
- Учесть возможность расширения функционала, настройкой другого оборудования;

3.3 Лингвистические требования

Для реализации данной системы использовать в качестве серверного языка - PHP, базы данных - Postgresql, клиентского языка - JavaScript, HTML, CSS.

3.4 Варианты использования системы

На основании требований системы, разработаем UML-диаграмму вариантов использования системы, выделим основных акторов.

Представлена UML-диаграмма на рисунке 3.2:

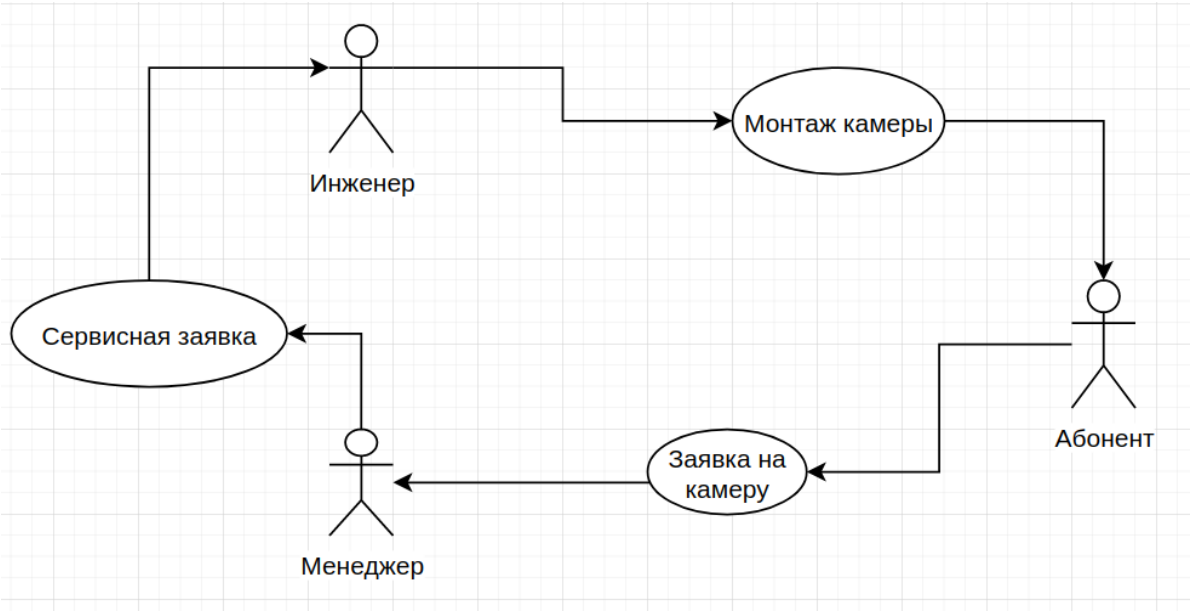

Рисунок 3.2 – Варианты использования

Выводы по разделу три

Процесс управления разработкой всегда начинается с требований. Необходимо спрашивать бизнес, задавать вопросы. В своей практике, автор сталкивался с ситуациями, когда излишне подробные вопросы принимают за техническую безграмотность. Это может отпугнуть некоторых разработчиков. Это категорически неверно. Вашим лучшим поведением для бизнеса будет катастрофическое количество вопросов, уточнений и предложений. Это все должно документировать и регламентироваться в требованиях к системе.

В данном разделе были рассмотрены основные виды требований: архитектурные, функциональные, лингвистические, представлены варианты использования системы.

Резюмируя данный раздел, автор предъявляет требования к системе установки камер, представленные выше и считает их принятыми к сведению.

Далее в четвертом разделе будет рассмотрен процесс управления разработкой на практике с помощью методологии Domain Driven Design, оценен экономический эффект и сделаны выводы.

4. СНИЖЕНИЕ ЗАТРАТ ПРЕДПРИЯТИЯ ПОСРЕДСТВОМ УПРАВЛЕНИЯ ПРОЕКТИРОВАНИИ ПО С ПОМОЩЬЮ МЕТОДОЛОГИИ DOMAIN DRIVEN DESIGN

Прежде чем приступать к разработке, необходимо разобраться с бизнес-процессом, который мы хотим автоматизировать. Для наглядной демонстрации бизнес-процесса автор работу использует BPMN нотацию и прилагающийся к ней инструментарий.

4.1 Диаграмма бизнес-процесса в нотации BPMN

Для начала представим словесное описание процесса установки камер абонентам на предприятии. Установка камеры - некая последовательность действий, которая приведет к функционированию услуг по видеоконтролю у абонента с одной стороны и фиксация монтажа, хранение информации о камерах пользователей, абонентской платы с другой стороны провайдера данной услуги. Далее некоторые технические подробности будут упрощаться. Установка камер будет происходить по следующему алгоритму:

1. Инициирование заявки на установку камеры.
2. Физический монтаж камеры.
3. Настройка камеры.
4. Настройка сетевого оборудования под данную камеру.
5. Создание записи в хранилище, уникальную для этой камеры.
6. Создание записи в хранилище, связывающей уникальную камеру и абонента.
7. Добавление абонентской платы по данному абоненту.
8. Камера установлена, видеоконтроль осуществляется, абонентская плата и камера зафиксированы за абонентом.

На рисунке 4.1 представлен бизнес-процесс в нотации BPMN:

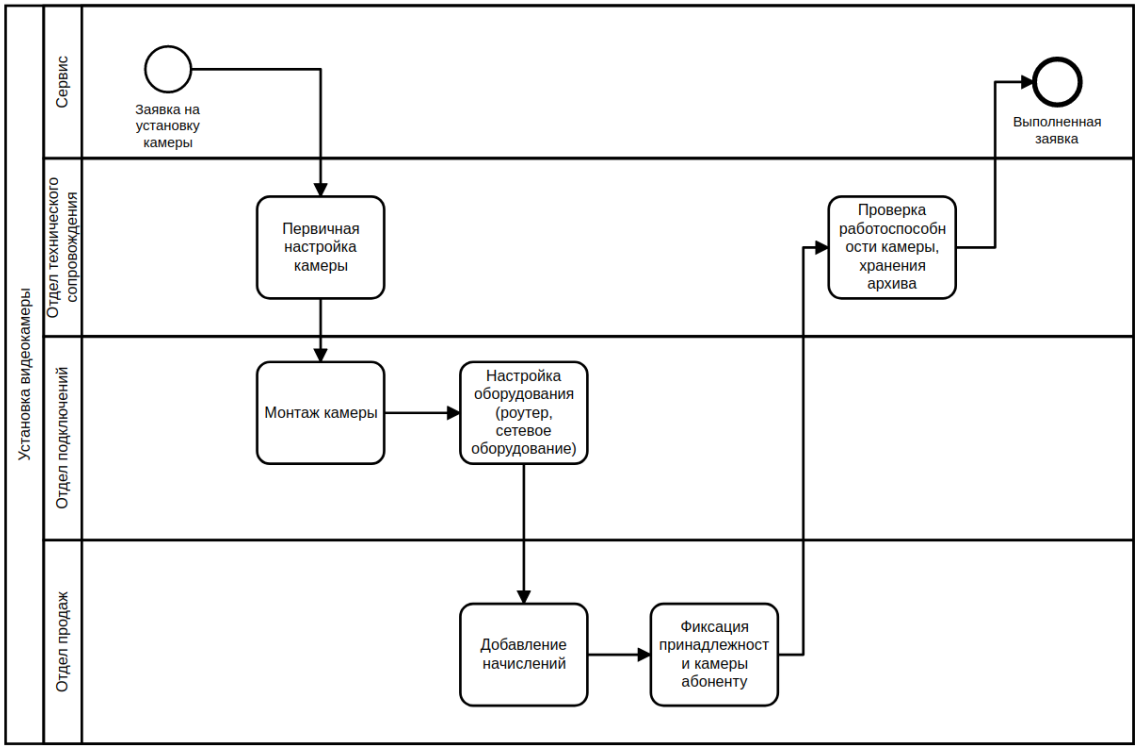

Рисунок 4.1 – Процесс установки видеокамеры в нотации BPMN.

Далее перейдем к разработке системы, автоматизирующей данный бизнес-процесс.

4.2 Разработка системы автоматизации установки камер

Любая разработка начинается с требований. Не с написания кода, ни с паттернов, ни с архитектуры, а именно с требований. Проблема и одновременно сложность этого этапа, обуславливается тем, что у всех звеньев команды свое видение на бизнес-процесс. Для того, чтобы не возникало таких ситуаций и все участники проекта могли взаимодействовать с бизнес-логикой необходимо составить единый язык - инструмент, который является основой, базисом методологии DDD.

4.2.1 Внедрение единого языка

Единый язык очень удобный паттерн, позволяющий не только всем участникам команды говорить на одном языке, но и позднее на этапе разработки кода - использовать понятия и переносить их в код. Как это реализовать? Совместное общение экспертов предметной области, аналитиков, стейкхолдеров и разработчиков позволяет составить этот язык. В таблице 1 представлен единый язык рассматриваемого процесса.

Таблица 1 – Единый язык процесса «Установка камер»

Камера (Cams)	Может обозначать как физическую камеру, так и виртуальную запись в хранилище.
Пользователь (User)	Физическое или юридическое лицо, уникально записанное в хранилище
Заявка (Order)	Заявка на некую услугу, может обозначать покупку, продажу. Также имеет значение уникальной записи в хранилище
Начисление (Accrual)	Абонентская плата за услугу видеоконтроля. Уникальная запись в хранилище .
Роутер (Router)	Маршрутизатор для создания беспроводного соединения.
Коммутатор (Switch)	Сетевое оборудование, обеспечивающее коммутацию.

Однако единый язык сам по себе ничего не стоит без контекста. Как видно из таблицы 1 у некоторых терминов может быть несколько значений. Далее автор разрабатывает карту контекстов.

4.2.2 Составление карты контекстов

Доменный ядром (core domain) данного процесса является установка камер. Выделив ядро предметной области, можно разделить на контексты нашу работу с камерами.

Карта контекстов состоит из доменного ядра, помещенного между ограниченными контекстами, которые характеризуют модули системы. Данный модули в рамках карты связаны друг с другом различного рода отношениями: партнерство (partnership), общее ядро (shared kernel), «заказчик - поставщик», конформист и так далее. На рисунке 4.2 представлена карта контекстов с объявленными между модулями отношениями:

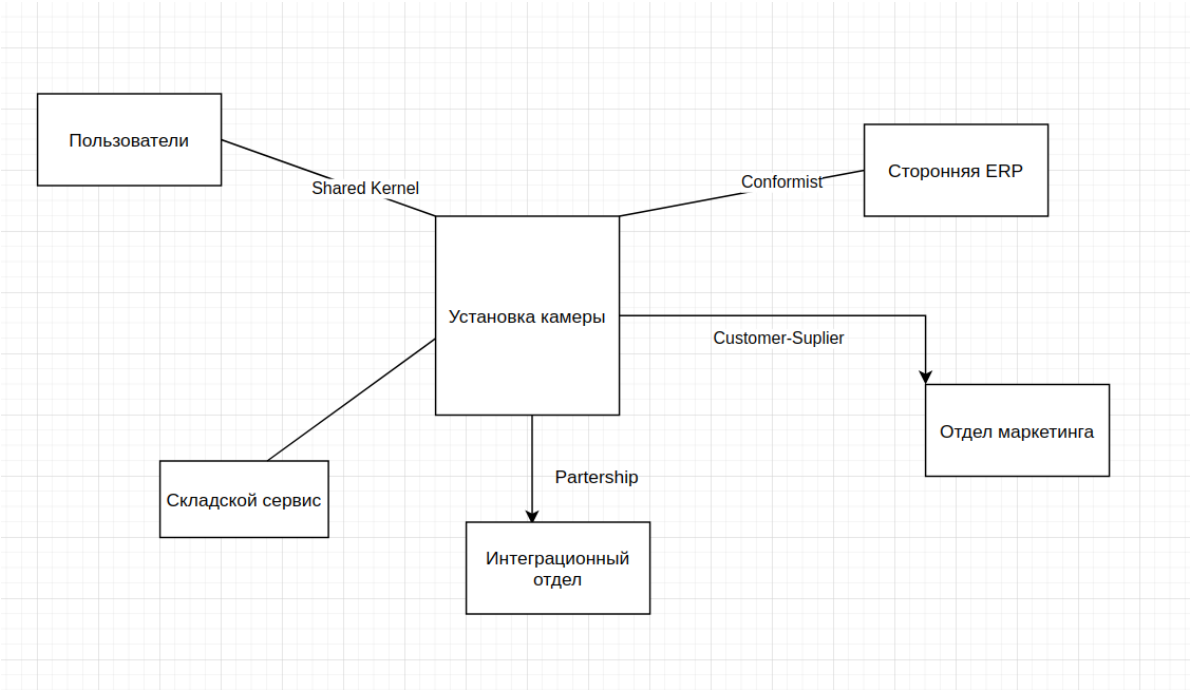

Рисунок 4.2 – Карта ограниченных контекстов установки камер

4.2.3 Практическая разработка продукта

Проектирование базы данных

Самым важным этапом разработки является проектирование базы данных. От того как вы проектируете хранение данных будет зависеть надежность системы, качество архитектуры и долговечность продукта.

В данном случае, для установки камер выделим основные таблицы и связи, представив их на рисунке 4.3:

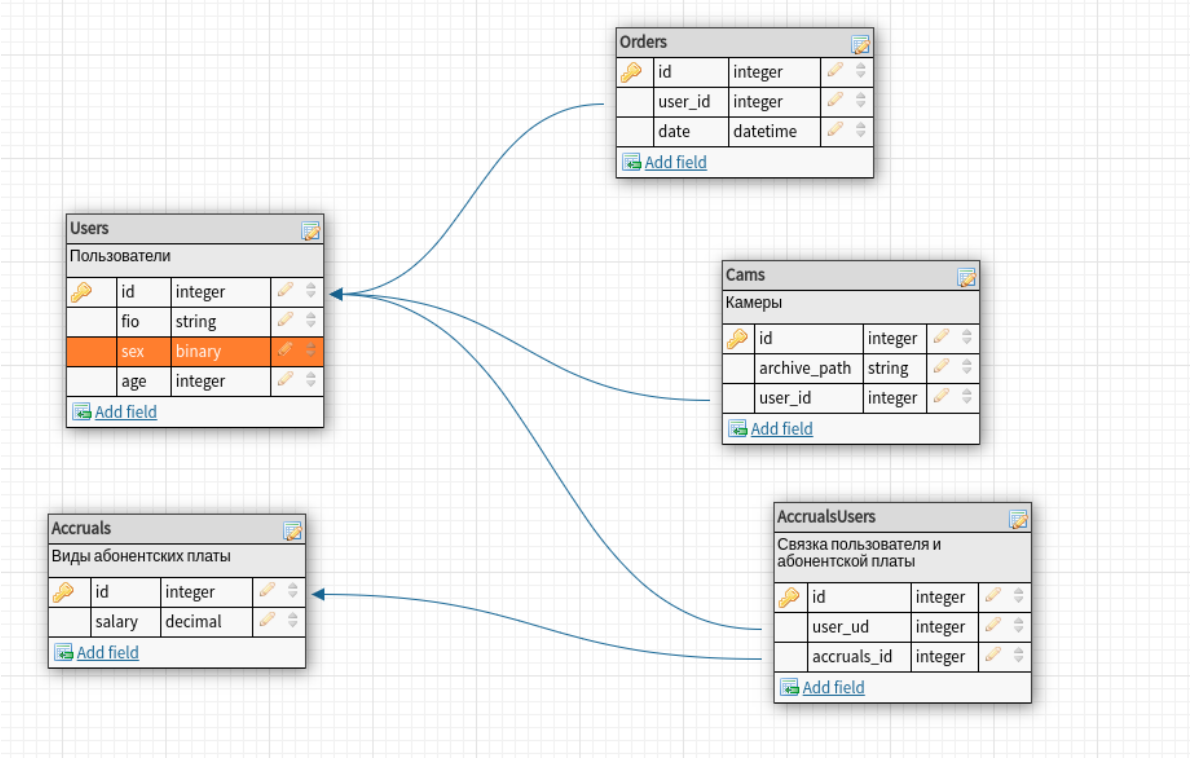

Рисунок 4.3 - Схема базы данных.

Агрегаты, DTO, entity, repositories

Паттерн Data Transfer Object помогает передавать от уровня к уровню данные о сущности камера. Паттерн Repositories инкапсулирует работу с хранилищем данных и помогает избавиться от высокой связности с источником данных. Агрегаты, исходя из названия собирают в себе работы с сущностями с помощью сервисов и через репозиторий отправляют данные в источник.

В приложении А представлен листинг кода основных компонентов системы.

4.3 Экономика IT - предприятия

Необходимость применения методологии DDD обуславливается в первую очередь предположительным количеством сценариев вашей системы. Если эта цифра равна 10-15, значит ваш бизнес процесс достаточно прост и лаконичен. Он не требует создания единого языка, карты контекстов и т.д [27]. Однако если количество UX-кейсов вашего проекта насчитывается более 20, то финансовые

затраты на внедрение рассматриваемого инструмента окупятся на длинном лаге. Рассмотрим развитие системы установки камер в двух ипостасях: с применением DDD и без нее. Сравнивать два подхода можно только введя некие ограничения, используя метод «при прочих равных». Ниже представлены ограничения, введенные автором:

- 20 бизнес-сценариев изначально, в планах дальнейшее развитие системы.
- Сроки старта проекта - 2 рабочих недели.
- Штат разработчиков - 4 программиста, 1 архитектор приложений, 2 продуктовых менеджера.
- Динамическое расширение функционала на протяжении 6 месяцев.
- Максимальное количество запросов в секунду (RPS) - 5000.
- Цена одного трудочаса специалиста - 1000р.
- Разработка по agile модели.

Затраты на реализацию данного проекта упрощенно будем считать через количество трудочасов умноженное на их стоимость. Некоторые условности, которые приводит автор (например, добавление бизнес-сценариев) добавляют погрешность измерений, однако на результат в целом не влияют.

4.3.1 Затраты на разработку без использования методологии Domain Driven Design

Предположим, что разработка MVP началась с разработки пользовательского интерфейса 2 разработчиками, параллельно с этим оставшиеся программисты занялись подготовкой MVP. Реализация бизнес-сценариев, описанных в начальных требованиях, происходит, предположительно за рабочую неделю двух разработчиков - 80 трудочасов. UI/UX систему реализовывают за то же время. Проектирование базы данных под нужный функционал занимается архитектор и занимает это 20 трудочасов. Предположительно, реализация каждого бизнес-сценария стоит 4 трудочаса программиста. Главная проблема заключается в том, что на практике при таком подходе, стоимость каждого последующего кейса

увеличивается экспоненциально. Следовательно, стоимость поддержки будет также увеличиваться экспоненциально. Ограничимся рамками, заданными в условиях до шести месяцев, предполагаю, что каждый месяц добавляется по четыре бизнес-сценария.

Для визуализации затрат автор приводит таблицу 1 с затратами на старт проекта и таблицу 2 с затратами на расширение функционала системы:

Таблица 1 – Затраты на MVP продукт.

Наименование работы	Затраты, руб
Проектирование базы данных	40 000
UI/UX	80 000
Реализация первых 20 бизнес-сценариев	80 000
Тестирование системы	20 000

Итого, на старт MVP продукта необходимо приблизительно 220 000 рублей. Далее представлена таблица 2 с затратами на расширение функционала в течение шести месяцев.

Таблица 2 – Затраты на расширение функционала в течение шести месяцев.

Пакет бизнес-сценариев	Затраты, руб
Бизнес-сценарии 21-25	20000
Бизнес-сценарии 26-30	40000

Продолжение таблицы 2.

Бизнес-сценарии 31-35	80000
Бизнес-сценарии 36-40	160000
Бизнес-сценарии 41-45	320000
Бизнес-сценарии 46-50	Невозможность поддерживать продукт

Итого, сумма вложений до перехода критической точки невозврата составляет сумма затрат на MVP - 220 тыс. рублей плюс затраты на расширение функционала 620 тыс. рублей - 840 тыс. рублей. При этом последующее развитие проекта обойдется значительно дороже его полного переписывания.

4.3.2 Затраты на разработку с применением методологии Domain Driven Design

Ключевым отличием применения этой технологии является расстановка приоритетов: на длинном лаге важнее проектировочные работы нежели MVP. Сначала специалисты в составе архитектора приложений и двух доменных экспертов составляют в течении одной недели единый язык, на котором будут общаться все участники команды. Параллельно с этим можно разрабатывать UI/UX системы. Далее составляется карта контекстов. И только после этого проектируется база данных и начинается разработка.

В таблице 3 представлена краткая сводка затрат на старт проекта:
Таблица 3 – Затраты на старт проекта.

Наименование работы	Затраты, руб
Составление единого языка	120000
Составление карты контекстов	240000
Проектирование базы данных	40000
UI/UX	80000
Реализация первых 20 бизнес-сценариев	80000
Тестирование системы	10000

Итого, на старт MVP продукта необходимо приблизительно 570 000 рублей. Далее представлена таблица 4 с затратами на расширение функционала в течение шести месяцев.

Таблица 4 – Затраты на расширение функционала в течение шести месяцев.

Пакет бизнес-сценариев	Затраты, руб
Бизнес-сценарии 21-25	20000
Бизнес-сценарии 26-30	20000
Бизнес-сценарии 31-35	20000
Бизнес-сценарии 36-40	20000

Продолжение таблицы 4.

Бизнес-сценарии 41-45	20000
Бизнес-сценарии 46-50	20000

Итого затраты на создание MVP и поддержку системы суммарно равны 690 тыс. рублей с возможностью горизонтального масштабирования [28]. При развитии функционала и добавлении бизнес-сценариев стоимость разработки не будет существенно изменяться.

4.3.3 Анализ полученных результатов

Для наглядности анализа автор приводит диаграммы 1 и 2 затрат на разработку продукта с использованием методологии DDD и без нее:

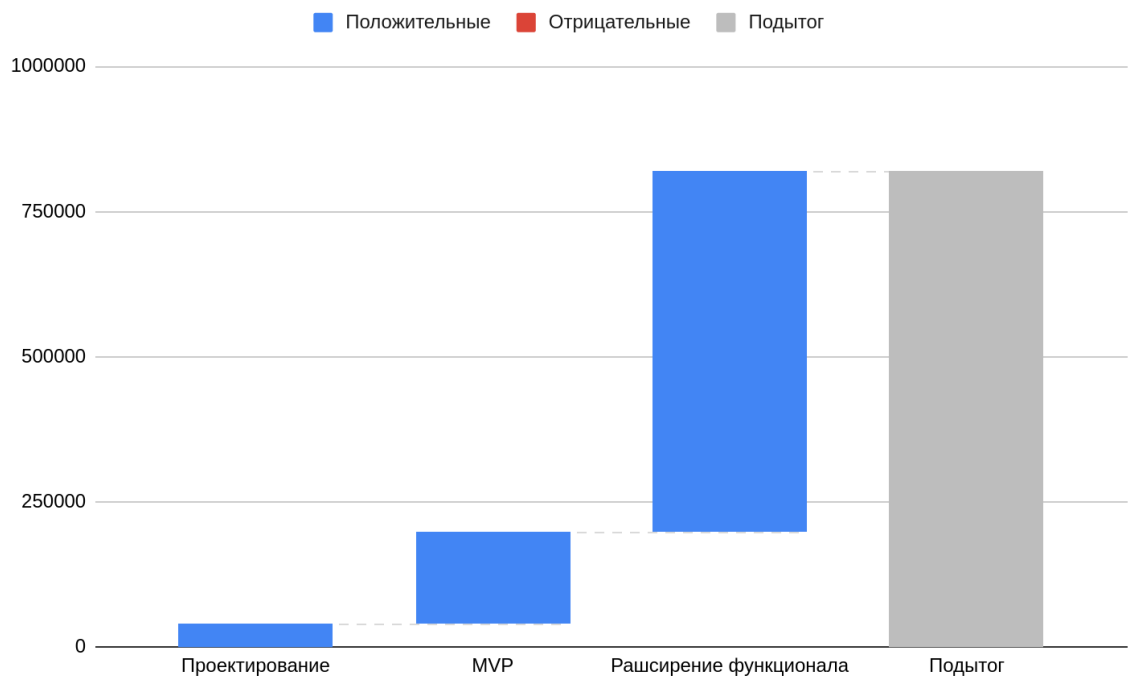

Диаграмма 1 – Затраты на разработку продукта без использования Domain Driven Design.

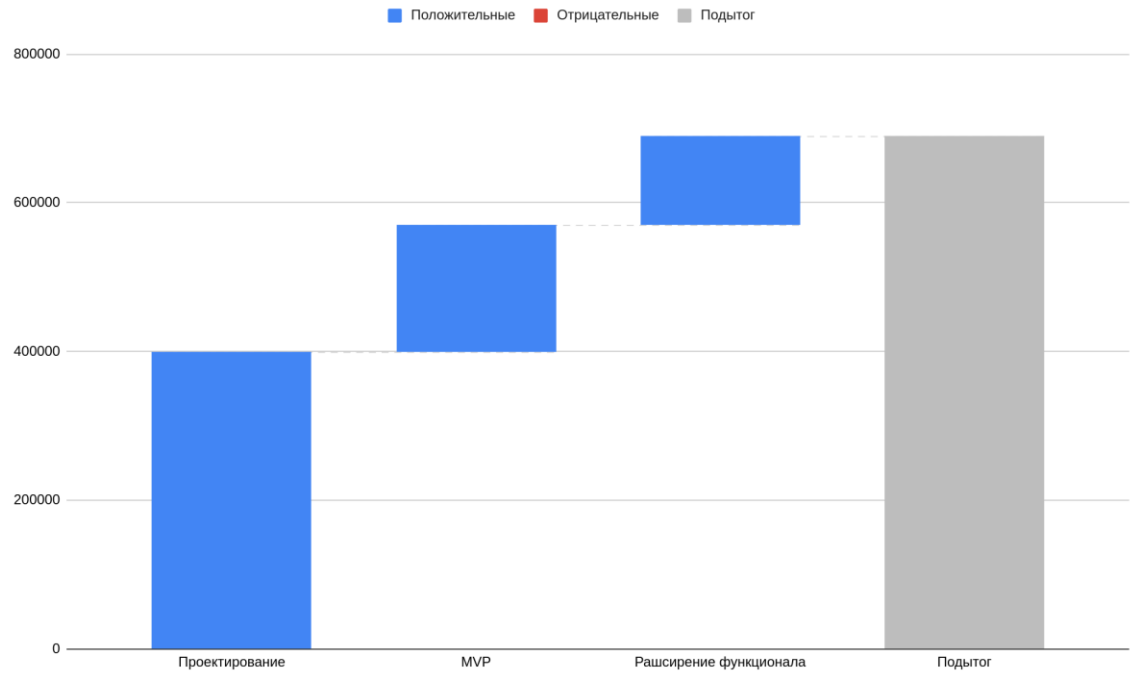

Диаграмма 2 – Затраты на разработку продукта с использованием Domain Driven Design.

Как видно из диаграмм, затраты на старт проекта без использования онной методологии значительно ниже и могут быть оправданы, если не планируется развитие продукта. Однако если развитие бизнес-сценариев планируется и притом активное - необходимо воспользоваться методологией Мартина Фаулера [5]. Отчетливо видно, что в разработке продукта без DDD подхода затраты на расширение функционала растут экспоненциально, а, следовательно, на какой-то итерации будет смерть проекта.

Выводы по разделу четыре

Разработка ПО на IT-предприятии – неизменно существующий, ресурсоемкий процесс. От того, каким путем пойдет разработка зависят затраты производства на будущую поддержку программного обеспечения, масштабирование и тестирование [29]. В данном разделе автор приводит пример, как можно снизить затраты производства в конкретной ситуации при проектировании и разработки ПО с помощью методологии Domain Driven Design.

Данная методология, как видно выше не только про написание кода, но и про взаимодействие разработчиков с бизнесом. Финансовые показатели строятся из многих факторов: срок жизни проекта, скорость донесения требований от бизнеса до технических специалистов, аффектированность взаимодействия продуктовых команд, проектирование базы данных и т.д [30]. Методология DDD объясняет, как эффективно работать с каждым их этих факторов, создавая полноценный продукт. В данном разделе была рассмотрена диаграмма бизнес-процесса по автоматизации установки камер абонентам, на практике рассмотрен пример разработки такой системы, а именно: составление единого языка, карты контекстов, спроектирована база данных, разработаны основные модули системы согласно паттернам проектирования методологии DDD. Рассмотрена экономика предприятия с данной методологией и без нее, сделаны выводы.

Резюмируя данный раздел, автор делает акцент, что применять DDD необходимо четко изучив требования к проекту и рассмотрев альтернативы, однако в частных ситуациях применение данной технологии оправдано, несмотря на начальные повышенные траты ресурсов.

ЗАКЛЮЧЕНИЕ

IT-предприятие - это сложная, многофункциональная фабрика специфической продукции - программного обеспечения. Для того, чтобы удовлетворить рынок, к программному обеспечению выдвигают определенные требования: масштабируемость, долговечность, отказоустойчивость и т.д. Зачастую, неправильное проектирование ПО может привести к банкротству предприятия. Избежать негативных последствий помогают методологии проектирования архитектуры, выбранные исходя из задач и частных факторов.

В рамках данной работы была проанализирована актуальная проблема IT-предприятий - снижение затрат при разработке программного обеспечения, не пренебрегая при этом его качеством.

Были выявлены цель и задачи, рассмотрен инструментарий, доступный для решения задач, были выработаны требования к системе и рассмотрены практический пример внедрения методологии Domain Driven Design при разработке ПО на IT-предприятии. Автор предоставляет следующий краткие выводы:

● Введено понятие IT-предприятия и проанализированы особенности проектирование ПО на нем.

● Были изучены современные подходы к разработке ПО, проанализированы достоинства и недостатки каждого их них.

● Был выбран бизнес-процесс «Автоматизация установки камер абонентам» для практического применения.

● Были выдвинуты различные требования к разрабатываемому продукту: архитектурные, функциональные, лингвистические. Рассмотрены варианты использования системы.

● Детализирован процесс разработки программного обеспечения посредством выбранной методологии Domain Driven Design.

● Был проведен анализ экономического эффекта использования выбранной методологии, сделаны выводы.

Таким образом, в результате данной работы автором была достигнута цель снижения затрат IT-предприятия, были выполнены поставленные задачи, доказана гипотеза эффективности использования методологии Domain Driven Design.

В заключение, автор рекомендует к использованию методологию Domain Driven Design при проектировании архитектуры программного обеспечения, когда количество бизнес-сценариев превышает порядок в 15-20 шт.

Данная работа обладает практической ценностью и рекомендована к апробации на IT-предприятии.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. IT-Предприятие (IT-Enterprise). [Электронный ресурс]. – Режим доступа: http://www.tadviser.ru/index.php/.

2. Модели разработки ПО. [Электронный ресурс]. – Режим доступа: https://geekbrains.ru/posts/methodologies.

3. Вендров, А.М. Проектирование программного обеспечения экономических информационных систем / А.М. Вендоров. –М.: Финансы и статистика, 2005. – 544 с.

4. Уилсон, С. Принципы проектирования и разработки программного обеспечения. Учебный курс / С. Уилсон, Б. Мэйплс, Т. Лэндгрейв. – М.: Русская редакция, 2002. –412 с.

5. Каскадная модель. [Электронный ресурс] – Режим доступа – https://qalight.com.ua/baza-znaniy/kaskadnaya-model-waterfall-model/.

6. Инкрементная модель. [Электронный ресурс] – Режим доступа – https://bytextest.ru/2017/11/23/incremental-model/.

7. Технический долг. [Электронный ресурс]. – Режим доступа: https://refactoring.guru/ru/refactoring/technical-debt/.

8. Ubiquitous Language и Bounded Context в DDD. [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/232881/.

9. Вернон, В. Implementing Domain Driven Design – М.: Вильямс, 2017. – 688 с.

10. Богданов, В. В. Управление проектами. Корпоративная система – шаг за шагом / В.В. Богданов. – М.: Манн, Иванов и Фербер, 2012. – 248 с

11. Фаулер, М. Шаблоны корпоративных приложений – М.: Вильямс, 2016 – 544 с.

12. Гвоздева, Т.В. Проектирование информационных систем: учеб. пособие / Т.В. Гвоздева, Б.А. Баллод, 2009 – 508 с.

13. Брукс, Ф. Мифический человек-месяц, или как создаются программные системы / Ф. Брукс, 2010 – 612 с

14. Фокс, Дж. Программное обеспечение и его разработка – М.: Мир, 1985. – 368 с.

15. Мартин, Р. «Чистый код» – М.: Санкт-Петербург, 2010. – 464 с.

16. Сазерленд, Д. Scrum революционный метод управления / Д.Сазерленд; пер. с англ. М. Гескиной. – М.: Манн, Иванов и Фербер, 2016. – 205 с

17. Бурков, В.Н. Математические основы управления проектами: Учебн. пособие / С.А. Баркалов, В.И. Воропаев, Г.И. Секлетова и др. под ред. В.Н. Буркова. – М.: Высшая школа, 2015. – 423 с.

18. Баронов, А.В. Информационные технологии и управление предприятием / В.В. Баронов, Г.Н. Калянов, Ю.И. Попов, И.Н. Титовский. – М.: Компания АйТи, 2006. — 328 с

19. Скрипкин, К. Г. Экономическая эффективность информационных систем / К.Г. Скрипкин. – Издательство: ДМК Пресс, 2002. — 256 с.

20. Гуц, А.К. Компьютерное моделирование. Инструменты для исследования социальных систем: Учебное пособие / А.К. Гуц, В.В. Коробицын, А.А. Лаптев, Л.А. Паутова, Ю.В. Фролова. – Изд. ОмГУ, 2001. – 92 с.

21. Коренная, К.А. Интегрированные информационные системы промышленных предприятий: монография / К.А. Коренная, О.В. Логиновский, А.А. Максимов; под ред. д-ра техн. наук, проф. А.Л. Шестакова. — Челябинск: Издательский центр ЮУрГУ, 2012. — 319 с.

22. Basecamp. [Электронный ресурс] – Режим доступа: – https://basecamp.com/.

23. Пример написания функциональных требований к Enterprise-системе. [Электронный ресурс] – Режим доступа: – https://habr.com/ru/post/245625/.

24. Программные требования спецификации - Software requirements specification. [Электронный ресурс] – Режим доступа: – https://ru.qwe.wiki/wiki/Software_Requirements_Specification/.

25. Похилько, А.Ф. CASE-технология моделирования процессов с использованием средств BPWin и ERWin учебное пособие / А.Ф. Похилько, И.В. Горбачев. — Ульяновск: УлГТУ, 2008. — 120 с

26. Как написать требования к программному обеспечению. [Электронный ресурс] – Режим доступа: – http://translatedby.com/you/how-to-write-a-software-requirements-specification/into-ru/.

27. Domain Driven Design на практике. [Электронный ресурс] – Режим доступа – https://habr.com/ru/post/334126/.

28. Простое о сложном – Domain Driven Design. [Электронный ресурс] – Режим доступа – https://fwdays.com/event/php-fwdays-17/review/domain-driven-design.

29. Корпоративная ИТ-инфраструктура: как снизить затраты. [Электронный ресурс] – Режим доступа http://lib.secuteck.ru/articles2/inegr_sistemy/korporativnaya-it-infrastruktura-kak-snizit-zatraty-chast-pervaya/.

30. Оптимизация расходов на IT-предприятии. [Электронный ресурс] – Режим доступа – https://alp-itsm.ru/interesting/sovetyi-po-optimizatsii-raskhodov-na-it-infrastrukturu/.

ПРИЛОЖЕНИЕ – Листинг программных элементов.

Класс DTO камер:

```
<?php
namespace app\modules\cams\models\dto;

use app\modules\cams\models\cams\BaseCams;
use yii\base\Model;

/**
* DTO для камеры
*
* @property $ip
* @property $mac
* @property $login
* @property $password
* @property $name
* @property $modelId
* @property $protocol
* @property $httpPort
* @property $rtspPort
* @property $type
* @property $audio
* @property $chargeTypeId
* @property $serialNum
*/
class Camera extends Model
{
  /** @var string Регулярка для мака */
  const MAC_PATTERN = '/^(?:[\dABCDEF]{2}:){5}[\dABCDEF]{2}$/';
  /** @var string Сценарий подключения физика */
  const FIZ_SCENARIO = 'fiz';
  /** @var string Дефолтный логин */
  const DEFAULT_LOGIN = 'admin';
  /** @var string Дефолтный пароль */
  const DEFAULT_PASSWORD = '7YtKjvfqcz5';
  /** @var int Дефолтный HTTP порт */
```

```
const DEFAULT_HTTP_PORT = 80;
/** @var int Дефолтный RTSP порт */
const DEFAULT_RTSP_PORT = 554;
/** @var int Тип камеры - клиентская */
const DEFAULT_TYPE_CAMERA = 3;
/** @var int Дефолтное значение записи звука */
const DEFAULT_AUDIO = 1;
/** @var int ONVIF compatible (dictionary ID = 541) */
const DEFAULT_PROTOCOL = 1;
/** @var string Домен камер */
const CAMS_DOMEN = '.cctv.is74.ru';

/** @var string */
public $ip;
/** @var string */
public $mac;
/** @var string */
public $login;
/** @var string */
public $password;
/** @var string */
public $name;
/** @var string */
public $modelId;
/** @var string */
public $protocol;
/** @var string */
public $httpPort;
/** @var string */
public $rtspPort;
/** @var int */
public $type;
/** @var int */
public $audio;
/** @var int */
public $chargeTypeId;
/** @var string */
```

```
    public $serialNum;

    /**
     * @return array
     */
    public function rules()
    {
        return [
            [['ip', 'mac', 'login', 'password', 'name'], 'required'],
            [['modelId', 'protocol', 'rtspPort', 'httpPort', 'type', 'audio', 'chargeTypeId'], 'integer'],
            [['ip', 'mac', 'login', 'password', 'name', 'serialNum'], 'string'],
            [['protocol'], 'default', 'value' => self::DEFAULT_PROTOCOL],
            [['audio'], 'default', 'value' => self::DEFAULT_AUDIO],
            [['rtspPort'], 'default', 'value' => self::DEFAULT_RTSP_PORT],
            [['httpPort'], 'default', 'value' => self::DEFAULT_HTTP_PORT],
            [['type'], 'default', 'value' => self::DEFAULT_TYPE_CAMERA],
            [['mac'], 'validateRegexMac'],
            [['chargeTypeId'], 'required', 'on' => self::FIZ_SCENARIO]
        ];
    }

    /**
     * Валидация мак адресов
     *
     * @return bool
     */
    public function validateRegexMac()
    {
        if (!preg_match(self::MAC_PATTERN, $this->mac)) {
            $this->addError('mac', 'MAC-адрес камеры "' . $this->name . '" указан не верно. Пример:
00:22:BE:6B:90:80');
            return false;
        }

        return true;
    }
```

```
/**
 * @return array
 */
public function attributeLabels()
{
    return [
        'ip' => 'IP/DDNS',
        'mac' => 'MAC-адрес',
        'modelId' => 'Модель',
        'protocol' => 'Протокол',
        'login' => 'Логин (от камеры)',
        'password' => 'Пароль (от камеры)',
        'name' => 'Название',
        'httpPort' => 'HTTP порт',
        'rtspPort' => 'RTSP порт',
        'chargeTypeId' => 'Начисление',
        'serialNum' => 'Серийный номер',
    ];
}

/**
 * Получить инициализированный объект дто
 *
 * @param null $mac
 * @return Camera
 */
public static function getInitDto($mac = null)
{
    $cams = new self();

    if ($mac) {
        $cams->ip = self::generateIpByMac($mac);
        $cams->mac = strtoupper($mac);
    }

    $cams->httpPort = self::DEFAULT_HTTP_PORT;
    $cams->rtspPort = self::DEFAULT_RTSP_PORT;
```

```
        $cams->login = self::DEFAULT_LOGIN;
        $cams->password = self::DEFAULT_PASSWORD;

        return $cams;
    }

    /**
     * Формирует IP для камеры по MAC
     *
     * @param $mac
     * @return string
     */
    private static function generateIpByMac($mac)
    {
        return str_replace(':', '', $mac) . self::CAMS_DOMEN;
    }
}
```

Репозиторий для работы с базой данных:

```
<?php

namespace app\modules\cams\models\repositories;

use app\models\ActiveRecordModels\Cams;
use app\models\ActiveRecordModels\CamsInternal;
use app\modules\cams\models\dto\Camera;
use yii\helpers\ArrayHelper;

/**
 * Работа с камерами через базу данных
 */
class CamsDbRepository implements CamsRepository
{
    /**
     * @inheritDoc
     * @throws \yii\db\Exception
     */
```

```
public function getDtoById($id): Camera
{
    $dto = new Camera();
    if ($cams = Cams::findOne($id)) {
        $dto->name = $cams->NAME;
        $dto->login = $cams->LOGIN;
        $dto->rtspPort = $cams->PORT;
        $dto->httpPort = $cams->HTTP_PORT;
        $dto->password = $cams->PASSWORD;
        $dto->modelId = $cams->userCams->MODEL_ID ?? null;
        $dto->serialNum = $cams->userCams->SERIAL_NUM ?? null;
        $dto->ip = $cams->IP;
        $dto->mac = $cams->MAC;
        $dto->chargeTypeId = $this->getChargeTypeId($cams->ID);
    }

    return $dto;
}

/**
 * @inheritDoc
 * @throws \yii\db\Exception
 */
public function getUserCams($userId): array
{
    $ids = array_map(function ($item) {
        return $item['CAMERA_ID'];
    }, \Yii::$app->db->createCommand(__DIR__ . '/../sqls/getCamsUserIds.sql')
        ->bindValue('userId', $userId)
        ->queryAll());

    return Cams::findAll(['ID' => $ids]);
}

/**
 * @param $id
 * @param $userId
```

```
     * @return bool
     * @throws \yii\db\Exception
     */
    public function activateCams($id, $userId): bool
    {
        $transaction = \Yii::$app->db->beginTransaction();
        try {
            if (!$camera = Cams::findOne($id)) {
                $transaction->rollBack();
                return false;
            }
            // Делаем камеру неактивной
            $camera->ACTIVE = 1;
            if (!$camera->save()) {
                throw new \Exception('Ошибка активации камеры');
            }

            // Создаем связку юзера с камерой
            if ($model = CamsInternal::findOne($id)) {
                $model->USER_ID = $userId;
                if (!$model->save()) {
                    throw new \Exception('Не удалось удалить связку пользовтаеля с камерой');
                }
            }

        } catch (\Exception $e) {
            \Yii::error('Не удалось активировать камеру: ' . $e->getMessage());
            $transaction->rollBack();
            return false;
        }

        $transaction->commit();

        return true;
    }

    /**
```

```
     * @param $id
     * @return bool
     * @throws \Throwable
     * @throws \yii\db\Exception
     */
    public function deactivateCams($id): bool
    {
        $transaction = \Yii::$app->db->beginTransaction();
        try {
            if (!$camera = Cams::findOne($id)) {
                $transaction->rollBack();
                return false;
            }
            // Делаем камеру неактивной
            $camera->ACTIVE = 0;
            if (!$camera->save()) {
                throw new \Exception('Ошибка деактивации камеры');
            }

            // Удаляем связку юзера с камерой
            if ($model = CamsInternal::findOne($id)) {
                if (!$model->delete()) {
                    throw new \Exception('Не удалось удалить связку пользовтаеля с камерой');
                }
            }

        } catch (\Exception $e) {
            \Yii::error('Не удалось деактивировать камеру: ' . $e->getMessage());
            $transaction->rollBack();
            return false;
        }

        $transaction->commit();

        return true;
    }
```

```
/**
 * @param int $id
 * @return mixed
 * @throws \yii\db\Exception
 */
public function getChargeIdByCams($id)
{
    $res = \Yii::$app->db
        ->createCommand(__DIR__ . '/../sqls/getChargeByCams.sql')
        ->bindValue('camsId', $id)
        ->queryOne();

    return $res['CHARGE_ID'];
}

/**
 * @param $chargeId
 * @param $camsId
 * @param $userId
 * @return mixed|void
 * @throws \yii\db\Exception
 */
public function addCharges($chargeId, $camsId, $userId)
{
    \Yii::$app->db->createCommand(__DIR__ . '/../sqls/mergePeriodicPrc.sql')
        ->bindValue('userId', $userId)
        ->bindValue('chargeTypeId', $chargeId)
        ->bindValue('comment', Cams::tableName())
        ->bindValue('camsId', $camsId)
        ->execute();
}

/**
 * @param $chargeId
 * @return int|mixed
 * @throws \yii\db\Exception
 */
```

```
public function removeCharges($chargeId)
{
    return \Yii::$app->db->createCommand(__DIR__ . '/../sqls/removePeriodicPrc.sql')
        ->bindValue('chargeId', $chargeId)
        ->execute();
}

/**
 * Получить ID типа начисления по камере
 *
 * @return false|null|string
 * @throws \yii\db\Exception
 */
public function getChargeTypeId($id)
{
    $res = \Yii::$app->db
        ->createCommand(__DIR__ . '/../sqls/getChargeByCams.sql')
        ->bindValue('camsId', $id)
        ->queryOne();

    return $res ? $res['CHARGE_TYPE_ID'] : null;
}

/**
 * @inheritDoc
 * @throws \yii\db\Exception
 */
public function getChargesList($userId)
{
    return \Yii::$app->cache->getOrSet(self::class . $userId, function () use ($userId) {
        $list = \Yii::$app->db
            ->createCommand(__DIR__ . '/../sqls/listCamsCharge.sql')
            ->bindValue('userId', $userId)
            ->queryAll();
        return ArrayHelper::map($list, 'CHARDGE_TYPE_ID', 'TYPE_DESCR');
    }, 3600);
}
```

```
    /**
     * @param Camera $camsDto
     * @param int $camsId
     * @return bool
     */
    public function saveCamsUser(Camera $camsDto, int $camsId, $userId): bool
    {
        $model = CamsInternal::findOne($camsId) ?? new CamsInternal(['CAMS_ID' => $camsId]);

        $model->USER_ID = $userId;
        $model->SERIAL_NUM = $camsDto->serialNum;
        $model->MODEL_ID = $camsDto->modelId;

        return $model->save();
    }

    /**
     * @inheritDoc
     */
    public static function findOne($id)
    {
        return Cams::findOne($id);
    }
}
```

Интерфейс репозиториев камер:

```
<?php

namespace app\modules\cams\models\repositories;

use app\models\ActiveRecordModels\Cams;
use app\modules\cams\models\dto\Camera;

/**
 * Интерфейс репозиториев для камер
 */
```

```
interface CamsRepository
{
    /**
     * Получить dto по id камеры
     * @param int $id ID камеры
     * @return Camera
     */
    public function getDtoById($id): Camera;

    /**
     * Получить камеры юзера
     * @param $userId
     * @return array
     */
    public function getUserCams($userId): array;

    /**
     * Получить камеру по id
     * @param $id
     * @return Cams|null
     */
    public static function findOne($id);

    /**
     * Активация камеры
     *
     * @param int $id ID камеры
     * @param $userId
     * @return bool
     */
    public function activateCams($id, $userId): bool;

    /**
     * Деактивация камеры
     *
     * @param $id
     * @return bool
```

```
     */
    public function deactivateCams($id): bool;

    /**
     * Получить id начисление по камере
     * @param int $id ID камеры
     * @return int
     */
    public function getChargeIdByCams($id);

    /**
     * Добавление начислений
     *
     * @param $chargeId
     * @param $camsId
     * @param $userId
     */
    public function addCharges($chargeId, $camsId, $userId);

    /**
     * Удаляем начисление
     *
     * @param $chargeId
     */
    public function removeCharges($chargeId);

    /**
     * Получить id типа начисления по камере
     *
     * @param int $id ID Камеры
     * @return int
     */
    public function getChargeTypeId($id);

    /**
     * Получить список наичслений по userId
     *
```

```
     * @param int $userId
     * @return array
     */
    public function getChargesList($userId);

    /**
     * Сохранение связки камера - юзер
     *
     * @param Camera $camsDto
     * @param int $camsId
     * @param int $userId
     * @return bool
     */
    public function saveCamsUser(Camera $camsDto, int $camsId, $userId): bool;
}
<?php
namespace app\modules\cams\models\devices;

use app\modules\cams\models\cams\BaseCams;
use yii\helpers\ArrayHelper;

/**
 * Класс для конфигурации роутера
 *
 * @property $typeConnect
 * @property $ports
 *
 */
class RouterSetting extends RouterInfo
{
    /** @var int Тип подключения WIFI */
    const WIFI_TYPE = 1;
    /** @var int Тип подключения ETHERNET */
    const ETHERNET_TYPE = 2;
    /** @var int Тип подключения LAN+WIFI */
    const ETHERNET_WIFI_TYPE = 3;
```

```
    /** @var int тип подключения камеры */
    public $typeConnect;

    /** @var array порты клиента */
    public $ports = [];

    /**
     * RouterSetting constructor.
     * @param $userId
     * @param array $config
     * @throws \yii\base\InvalidConfigException
     * @throws \yii\db\Exception
     */
    public function __construct($userId, array $config = [])
    {
        parent::__construct($userId, $config);

        $this->initRouterData();
    }

    /**
     * @return array
     */
    public function rules()
    {
        return ArrayHelper::merge(parent::rules(),
            [
                [['typeConnect'], 'required'],
                ['typeConnect', 'in', 'range' => [self::WIFI_TYPE, self::ETHERNET_TYPE,
self::ETHERNET_WIFI_TYPE]],
                [['ports'], 'required', 'when' => function () {
                    return $this->typeConnect != self::WIFI_TYPE;
                }],
            ]
        );
    }
```

```
    /**
     * @return array
     */
    public function attributeLabels()
    {
        return ArrayHelper::merge(parent::attributeLabels(),
            [
                'ports' => 'Порты',
                'typeConnect' => 'Тип подключения',
            ]
        );
    }

    /**
     * Типы подключений
     *
     * @return array
     */
    public static function getConnectTypes()
    {
        // todo пока нет
        return [
            self::WIFI_TYPE => 'WI-FI',
//          self::ETHERNET_TYPE => 'ETHERNET',
//          self::ETHERNET_WIFI_TYPE => 'WI-FI + ETHERNET'
        ];
    }

    /**
     * Конфигурация роутера в зависимости от выбранного типа подключения
     *
     * @return bool
     * @throws \yii\base\InvalidConfigException
     */
    public function configure()
    {
        if (!$this->settingPort()) {
```

```
            return false;
        }

        switch ($this->typeConnect) {
            case self::WIFI_TYPE:
                return $this->configureWifi();
            case self::ETHERNET_TYPE:
                return $this->configureEthernet();
            case self::ETHERNET_WIFI_TYPE:
                return $this->configureWifiAndEthernet();
            default:
                return false;
        }
    }

    /**
     * Настройка порта абонента
     *
     * @return bool
     */
    public function settingPort()
    {
        if (!$this->addVlanPort()) {
            $this->addError('ALL', 'Ошибка добавления VLAN на порт БС');
            return false;
        }

        return true;
    }

    /**
     * Проброс вилана на порт абонента
     *
     * @return bool
     */
    private function addVlanPort()
    {
```

```
            return \Yii::$app->swcmApi->addPortVlan($this->bsName, $this->bsPort, [],
[BaseCams::CAMS_VLAN]);
        }

        /**
         * Конфигурация по Wi-Fi
         *
         * @return bool
         * @throws \yii\base\InvalidConfigException
         */
        private function configureWifi()
        {
            if (!$this->genieAcs->configureRouterToVideoWifi($this->routerId)) {
                $this->addError('routerId', 'Не удалось сконфигурировать роутер по WI-FI');

                return false;
            }

            return true;
        }

        /**
         * Конифгурация по Ethernet
         *
         * @return bool
         */
        private function configureEthernet()
        {
            if (!$this->genieAcs->configureRouterToVideoEthernet($this->routerId, $this->ports)) {
                $this->addError('routerId', 'Не удалось сконфигурировать роутер по Ethernet');

                return false;
            }

            return true;
        }
```

```
    /**
     * Конфигурация по wifi + ethernet
     *
     * @return bool
     * @throws \yii\base\InvalidConfigException
     */
    private function configureWifiAndEthernet()
    {
        if ($res = $this->configureWifi()) {
            $res = $res && $this->configureEthernet();
        }

        return $res;
    }

    /**
     * Инициализация данных с роутера
     * @throws \yii\base\InvalidConfigException
     */
    private function initRouterData()
    {
        $data = $this->genieAcs->getRouterData($this->routerId);
        if (isset($data['isWifiType']) && $data['isWifiType']) {
            $this->typeConnect = self::WIFI_TYPE;
        }

        return;
    }
}
```

```
<?php
namespace app\modules\cams\models;

use app\models\ActiveRecordModels\ExecutorRemainEquipV;
use app\models\ActiveRecordModels\SvcRequest;
use app\models\ActiveRecordModels\SvcRequestItem;
use app\models\RequestSearch;
```

```
use yii\base\Model;
use yii\db\Exception;
use yii\helpers\ArrayHelper;

/**
 * Класс, реализующий логику переноса ТМЦ от инженера к абоненту и обратно
 *
 * @property $fromUserId
 * @property $toUserId
 * @property $items
 */
class TmcTransfer extends Model
{
    /** @var array */
    public $items = [];

    /** @var int */
    private $fromUserId;
    /** @var int */
    private $toUserId;

    /**
     * TmcTransfer constructor.
     * @param $fromUserId
     * @param $toUserId
     * @param array $config
     */
    public function __construct($fromUserId, $toUserId, array $config = [])
    {
        $this->fromUserId = $fromUserId;
        $this->toUserId = $toUserId;

        parent::__construct($config);
    }

    /**
     * @param $userId
```

```
     * @param $back
     * @return array
     */
    public static function getEquipmentList($userId, $back = null)
    {
        return ArrayHelper::map(ExecutorRemainEquipV::findAll(['EXECUTOR_USER_ID' =>
$userId]), 'ID_1C', 'NAME_1C');
    }

    /**
     * @return array
     */
    public function rules()
    {
        return [
            [['items'], 'safe'],
        ];
    }

    /**
     * @return array
     */
    public function attributeLabels()
    {
        return [
            'items' => 'Оборудование',
        ];
    }

    /**
     * @return int
     */
    public function getFromUserId(): int
    {
        return $this->fromUserId;
    }
```

```
/**
 * @return int
 */
public function getToUserId(): int
{
    return $this->toUserId;
}


/**
 * Перенос ТМЦ
 *
 * @return bool
 * @throws Exception
 */
public function transfer()
{
    if (!$this->validate()) {
        return false;
    }

    $transaction = \Yii::$app->db->beginTransaction();

    try {
        // Создаем списание
        if (!$svcRequest = $this->createSvcRequest()) {
            $transaction->rollBack();
            return false;
        }

        // Сохраняем оборудование
        foreach ($this->items as $item) {
            $model = new SvcRequestItem();
            $model->REQUEST_ID = $svcRequest->ID;
            $model->load($item, '');
            if (!$model->save()) {
                $transaction->rollBack();
```

```
                $this->addErrors($model->getErrors());
                return false;
            }
        }

        $transaction->commit();
        return true;
    } catch (\Exception $e) {
        $transaction->rollBack();
        $this->addError('ALL', 'Непредвиденная ошибка списания ТМЦ');
        return false;
    }
}

/**
 * Создаем списание
 *
 * @return SvcRequest|null
 */
private function createSvcRequest()
{
    $request = new SvcRequest();
    $request->TYPE = SvcRequest::TYPE_INSTALL;
    $request->TO_OBJECT_TYPE = SvcRequest::OBJECT_TYPE_CLIENT;
    $request->TO_OBJECT_ID = $this->toUserId;
    $request->FROM_OBJECT_TYPE = SvcRequest::OBJECT_TYPE_USER;
    $request->FROM_OBJECT_ID = $this->fromUserId;
    $request->STATE = SvcRequest::TYPE_NEW;

    if (!$request->save()) {
        $this->addErrors($request->errors);
        return null;
    }

    return $request;
}
```

```
    /**
     * Согласовать требование
     *
     * @param $id
     * @return bool
     */
    private function acceptRequest($id)
    {
        $model = RequestSearch::findOne($id);

        return $model->confirm();
    }
}
```