

Документ подписан простой электронной подписью  
Информация о владельце:

ФИО: Баламирзоев Назим Лиодинович

Должность: Врио ректора

Дата подписания: 19.07.2022 09:23:23

Уникальный программный ключ:

b261c06f25acbb0d1e6de51c04abdded009fd138

## Лекция №1 Программная инженерия

## Введение

УМКД «Методология программной инженерия» предназначен для студентов по направлению подготовки 09.04.04 Программная инженерия с учетом специфики направленности подготовки – «Системы искусственного интеллекта»

# МЕТОДОЛОГИЯ ПРОГРАММНАЯ ИНЖЕНЕРИЯ

## курс лекций

---

## ВВЕДЕНИЕ

В современных условиях очень динамично развивается рынок *интегрированных программных систем (ПС)*, которые позволяют автоматизировать деятельность предприятий и учреждений самого различного профиля (финансовых, промышленных, офисных) и самых различных размеров с разнообразными схемами иерархии, начиная от малых предприятий численностью в несколько десятков человек и завершая крупными корпорациями численностью в десятки тысяч сотрудников. По данным [1] более семи миллионов человек занимаются разработкой ПС, а сотни миллионов активно их используют<sup>1</sup>.

Быстрое *увеличение сложности и размеров* современных комплексов программ при одновременном *повышении ответственности выполняемых функций* резко повысило требования со стороны пользователей к их качеству, надежности функционирования и безопасности применения, а также привело к принципиальному изменению методов в этой сфере: к переходу от технологии индивидуального программирования отдельных небольших программ к коллективному созданию крупных комплексов программ инженерными методами проектирования и разработки [2].

При индустриальном подходе к разработке и сопровождению ПО особый вес приобретают *технологические характеристики* разрабатываемых программ, для получения качественных программных продуктов необходимо руководствоваться следующими принципами [3]:

- *эффективностью* – результаты должны отвечать заданным требованиям и стандартам в условиях ограниченных ресурсов;
- *практичностью* – результаты должны иметь конкретных заказчиков;
- *фундаментальностью* – результаты должны базироваться на знаниях фундаментальных наук;
- *сопровождаемостью* – результаты, находясь в эксплуатации, обязательно должны обслуживаться.

Накопление в мире знаний, опыта разработки и применения огромного количества различных сложных программ для ЭВМ, способствовало *систематизации и обобщению методов и технологий их разработки*, сокращению дефектов и неопределенностей в характеристиках и качестве поставляемых и применяемых программных продуктов. В результате сформировалась современная *методология и инженерная дисциплина*

---

<sup>1</sup> Программы следующего десятилетия // Открытые системы. – Декабрь, 2001. – С.60–71

обеспечения процессов жизненного цикла сложных программных продуктов – **программная инженерия** для различных областей применения [3].

Начало работ в данной области относится к концу 60-х – началу 70-х годов, когда рост сложности ПС стал приводить к снижению качества их функционирования и появлению большого количества ошибок. Сложность ПС постоянно увеличивалась из-за:

- увеличения объемов кода (миллионы строк);
- увеличения количества связей между элементами систем;
- увеличения количества разработчиков (сотни человек);
- увеличения количества пользователей (сотни и тысячи).

Первые существенные результаты при решении данной проблемы были получены А. П. Ершовым, В. М. Глушковым, Е. А. Жоголевым, В. В. Липаевым, в работах которых описаны базовые методологии и технологии программирования того времени: структурное программирование и методология процедурной (алгоритмической) декомпозиции. К 90-м годам появилась новая парадигма в программировании, построенная на объектной декомпозиции предметной области, которая привела к методологии объектно-ориентированного анализа и проектирования. Появляется понятие программной инженерии как практического приложения научных знаний в проектировании и конструировании ПС, а также для создания документации, необходимой для их разработки, эксплуатации и сопровождения.

Основной концепцией программной инженерии стало понятие *жизненного цикла* (ЖЦ) ПО. В настоящее время применяется несколько моделей жизненного цикла, которые отличаются набором фаз (этапов, стадий) проекта по созданию ПО, отдельных процессов, операций и задач. В настоящее время используются как классические модели жизненного цикла: каскадная (водопадная) и спиральная, так и их модификации, которые охватывают все этапы жизненного цикла ПО и успешно применяются для решения практических задач.

Только скоординированное, комплексное применение в проектах (с начала проектирования до внедрения программных систем) современных методов и промышленных технологий позволит достичь высокого качества, необходимого для использования их в сложных системах обработки информации.

## 1 ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ КУРСА

### Программирование

На протяжении всего времени обучения на факультете мы изучаем программирование. *Программирование (Computer science)* - молодая, активно развивающаяся область.

Долгое время человечество волнует вопрос о том, к какому роду деятельности относится программирование. В 60-х-70-х годах XX века данный вопрос активно обсуждался на научных конференциях. Существовало 2 популярных точки зрения: «программирование - это искусство» и «программирование - это наука». К единому мнению придти так и не удалось. В настоящий момент мы можем добавить к этим популярным трактовкам еще одну: «программирование - это бизнес». Чтобы понять, что программирование это бизнес, достаточно посмотреть, какими числами выражаются доходы современных IT-компаний. Так, например, по данным [www.microsoft.com](http://www.microsoft.com) доход корпорации Microsoft за 2008 финансовый год составил более 60 млрд. \$.

**Программа (program)** – это набор операторов, который может быть представлен как единое целое в некоторой вычислительной системе и который используется для управления поведением этой системы.

**Программирование (в узком смысле)** – процесс кодирования и отладки программы в рамках реального проекта.

**Программирование (programming) (в широком смысле)** – все технические операции, необходимые для создания программы, включая анализ требований и все стадии разработки и реализации.

### Программы и программное обеспечение (программные продукты)

В 1958 г. известный статистик Джон Тьюкей (JoHn Tukey) впервые ввел термин software – программное обеспечение.

**Программное обеспечение (Software)** - набор компьютерных программ, процедур и связанной с ними документации и данных [4].

Таким образом, программное обеспечение (ПО) - это не просто программа. Это еще и *конфигурационные данные*, необходимые для корректной работы программы, и *вся сопутствующая документация* (в том числе и руководство пользователя).

**Программные системы (ПС)** состоят из совокупности программ, файлов конфигурации, необходимых для установки этих программ, и документации, которая

описывает структуру системы, а также содержит инструкции для пользователей, объясняющие работу с системой, и чисто адрес web-узла, где пользователь может найти самую последнюю информацию о данном программном продукте [5].

Вместо словосочетания «программное обеспечение» часто используют другое – «программный продукт» (ПП). Будем далее считать, что это одно и то же. Одно из главных свойств программного продукта - *продаваемость*. Продаваемость - залог успеха бизнеса по разработке программного обеспечения. Если вы собираетесь что-то разработать, это должно быть востребовано на рынке. В противном случае вы потратите деньги на разработку (зарплата сотрудников, накладные расходы, налоги, аренда помещения...) и ничего не получите взамен. Вы можете написать замечательную программу, реализовать там новый быстрый алгоритм. Она может великолепно работать, но если она никому не нужна, то вы (как компания) на пути банкротству. Допустим, в таких программах, как ваша, действительно есть потребность. Допустим, вы год упорно работали, и вот, казалось бы, настал ваш звездный час: все готово, все модули написаны, отлажены, собраны вместе и, как вам кажется, работают. Один «маленький» момент портит всю картину - если у вас нет хорошего (!) руководства пользователя (инструкции), желательно, в русскоязычном и англоязычном вариантах, то вашу программу никто не купит, особенно за границей. Если у вас все есть, но нет специалистов по рекламе, то про вашу программу никто не узнает. Если ...

Программные продукты делятся на два типа [5]:

1. *Общие ПП*. Это автономные программные системы, которые созданы компаниями по производству ПО и продаются на открытом рынке программных продуктов любому потребителю, способному их купить (так называемое «Коробочное ПО»). Примерами таких ПП могут служить системы управления базами данных (СУБД), текстовые и табличные процессоры (MS Office), графические пакеты и т.п.
2. *ПП, выполненные под заказ*. Это ПС, выполненные по заказу определенного потребителя согласно заключенному контракту. Примерами таких ПП могут служить системы поддержки определенных бизнес-процессов, системы управления воздушным транспортом и т.п.

Важное отличие между этими типами ПП заключается в том, что при создании общих ПП спецификация требований для них разрабатывается компаниями производителей, а для ПП, выполненных под заказ, - организацией-заказчиком. Спецификация необходима разработчикам ПО для создания любого ПП.

Подытожим: **программный продукт** - это программа со всей сопутствующей документацией, программа, которую можно продать, либо извлечь из нею финансовую выгоду другим образом.

### ИТ-проекты

Будем понимать под **ИТ-проектами** проекты в области информационных технологий. Будем далее рассматривать лишь те ИТ-проекты, целью которых является разработка программного обеспечения.

Для того чтобы бизнес был успешным, необходимо (но не достаточно) выполнение многих условий:

- Продукт должен выходить на рынок
  - ✓ надлежащего качества;
  - ✓ вовремя;
  - ✓ интересным потенциальным пользователям.
- Расходы должны соответствовать изначальному бюджету.

К сожалению, ситуация такова, что многие проекты не удовлетворяют этим, казалось бы естественным, условиям. Рассмотрим некоторую статистику (рисунок 1.1).

### Успешные проекты нечасты в ИТ



Статистика по 30,000 проектам по разработке ПО в американских компаниях.

Рисунок 1.1 - Статистика успешности ИТ-проектов

(по данным The Standish Group International, Extreme Chaos)

Приведем расшифровку степени успешности проектов:

- *Проваленные*: закончились неудачей - цель вообще не была достигнута.
- *Испытавшие большие проблемы*: закончились созданием продукта, но превысили бюджет или (и) не уложились во время или (и) имеют лишь частичную функциональность.

- *Успешные*: закончились созданием продукта, уложились в бюджет и время. Вся планируемая функциональность реализована.

Как видно из диаграммы, доля успешных проектов неуклонно возрастает, оставаясь по-прежнему сенсационно малой. И это притом, что в 2004 году на разработку программных средств ушло около 3 700 000 000 \$.

Рассмотрим еще одну диаграмму (рисунок 1.2), из которой видно, что с ростом размера проекта (бюджет характеризует в данном случае размер и сложность задачи) шансы на его успех катастрофически падают.

## Project Success

Smaller initiatives fare better at reaching goals than larger projects do.

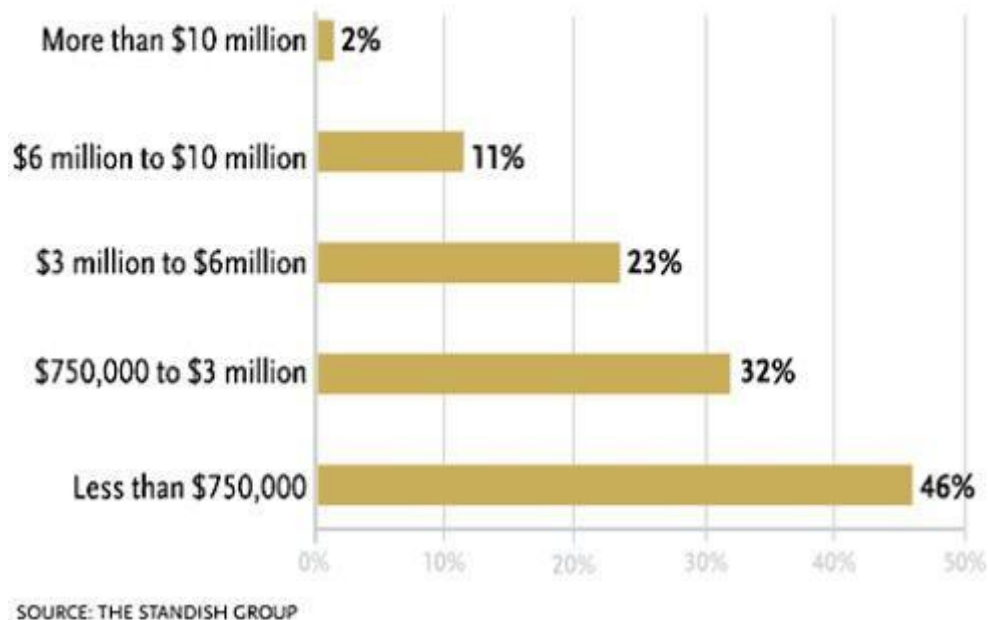


Рисунок 1.2 - Доля успешных проектов в зависимости от их бюджета

### Причины неудачи IT-проектов

Почему IT-проекты терпят неудачи: почему, казалось бы, хорошо спланированный проект не укладывается во временные рамки; почему по прошествии некоторого времени выясняется, что имеющегося бюджета недостаточно; почему полученный в итоге продукт не пользуется спросом?

Проблема сложна и многогранна. Трудно перечислить все возможные причины неудачи. Остановимся кратко на некоторых из них, представляющихся нам наиболее существенными.

#### Причина 1. Нереалистичные временные рамки.

Правильно оценить время, необходимое для выполнения проекта, - сложная задача, решение которой часто не под силу даже опытным менеджерам. Существуют

специальные критерии, которые помогают принимать правильные решения, такие как учет времени в человеко-часах и т.д. Тем не менее, задача остается сложной, колоссальное значение в ней имеет грамотный учет рисков (далее мы поговорим про это подробнее).

### **Причина 2. Недостаток количества исполнителей.**

Иногда менеджер решает сэкономить, иногда переоценивает возможности своих сотрудников, иногда в ходе разработки выясняется, что задача сложнее, чем казалось на самом деле, - проблема недостатка рабочих рук, так или иначе, возникает достаточно часто.

### **Причина 3. Размытые границы проекта.**

Одна из наиболее серьезных причин неудачи проекта - нечетко сформулированные цели, неоднократно меняющиеся в ходе разработки. Если вам доведется управлять проектом - сделайте все, чтобы четко сформулировать требования к системе в соответствии с пожеланиями пользователя. Мы поговорим про это подробнее в подразделе «Управление требованиями».

### **Причина 4. Недостаток средств.**

Известны две крайности при планировании бюджета: чрезмерное раздувание (подход пессимиста) и чрезмерное уменьшение (подход оптимиста). Использование первого подхода чаще всего (если только ваш заказчик не совсем дилетант) приводит к тому, что ваша команда теряет проект («Слишком дорого, сэр. Мы идем к Вашим конкурентам»). Второй подход часто применяется не только в силу оптимизма менеджмента, но и в рекламных целях, чтобы любой ценой выиграть проект. Представляется разумным оценивать бюджет реально с некоторой перестраховкой на случай непредвиденных ситуаций (заболел ключевой сотрудник, вышло из строя дорогостоящее оборудование...).

### **Причина 5. Нехватка квалифицированных кадров.**

Нехватка квалифицированных специалистов - одна из существенных проблем отрасли. Технологии развиваются с такой скоростью, что профессионалы вынуждены все время обновлять свои знания. Относительная новизна самой области ИТ, с одной стороны, становящееся повсеместным внедрение информационных технологий во все сферы человеческой деятельности, с другой, а, значит, все возрастающий спрос на специалистов ведут к существенной нехватке квалифицированных кадров. Конечно, все хотят принять на работу лучших. Но опыт показывает, что их не так много, и на всех не хватает. Умение из потока кандидатов выбрать тех, кто вам нужен, очень важное качество специалистов по



кадрам. Часто к подбору сотрудников рекомендуют привлекать всех членов команды. То, как новичок впишется в коллектив, совсем не последнее дело.

Ситуация в России существенно изменилась к лучшему за последние несколько лет. Объем экспорта программного обеспечения из России в 2005 г. превысил 1 млрд.\$ (для сравнения экспорт в автомобильной отрасли составил 380 млн.\$, в атомной энергетике – 850 млн. \$). Объем IT-рынка в 2004 г. составил 9,2 млрд. \$, в 2005 г. рост составил 22,1% (в то время как мире всего порядка 6%). Конечно, доля нашего рынка в объемах мирового IT-рынка по-прежнему невелика (объем IT-рынка в мире в 2005 г. составил 900 млрд. \$), но тенденция выглядит обнадеживающе. При этом объем рынка разработки ПО в России в 2005 г. составил 1,4 млрд. \$ (от всего IT-рынка). В среднем этот показатель в России растет на 40-50% в год.

Таким образом, основные тенденции на сегодняшний день представляются следующими:

- Быстрый рост объемов IT-рынка, рынка ПО.
- Укрепление позиций российских компаний.
- По-прежнему малая доля в мировых объемах.

Для того чтобы повысить объемы рынка, необходимо повысить показатель успешности проектов, для чего отрасль нуждается не только в новых технологиях, но и в грамотных специалистах, способных эти технологии применять.

### **Инженерия программного обеспечения (программная инженерия)**

Современные ПС являются по составу и структуре очень разнородными, ориентированными на реализацию разных предметных областей, начиная от операционных систем (ОС) и заканчивая прикладными бизнес-системами. Примерно каждые 10 лет происходит смена языков программирования и операционных сред для описания и функционирования программ, что предполагает перевод ранее разработанных и функционирующих программ на новые языки и операционные среды. На это тратятся огромные людские и финансовые ресурсы.

#### **Немного истории**

Уже в 60-х-70-х годах XX было ясно, что ввиду роста сложности решаемых при помощи компьютера задач неимоверно возрастает стоимость разработки программ. Причем, если стоимость аппаратуры растет умеренными темпами, а иногда и вовсе падает, то со стоимостью разработки программ ничего поделать не удастся. Именно тогда вопрос о том, как оптимизировать процесс разработки, вышел на первый план.

Создание любой ПС выполняется по некоторой схеме, которая представляет собой последовательность стандартных этапов (очень приблизительно эта схема может выглядеть так: анализ, проектирование, разработка, тестирование, модификация). Именно на этих этапах и возникают существенные финансовые затраты. Для их оптимизации необходимо было понять, что программирование есть обычный технологический процесс, по характеру возникающих проблем мало чем отличающийся от, скажем, строительства дома или корабля. Для сокращения затрат необходимо было конкретизировать схему, упорядочить действия, выполняемые на каждом этапе, разработать методы решения возникающих на разных этапах проблем. Для решения данных проблем была создана такая дисциплина, как программная инженерия.

На сегодняшний день нет единого определения понятия «программная инженерия» [6]. Термин «Инженерия программного обеспечения» появился впервые в 1968 г. на Конференции НАТО «Инженерия программного обеспечения» (г. Гармиш, Германия), на которой обсуждались проблемы существующего в то время «кризиса программного обеспечения». На конференции присутствовало 50 профессиональных разработчиков ПО из 11 стран. Рассматривались проблемы проектирования, разработки, распространения и поддержки программ. Приведем несколько таких определений, данных крупными специалистами в этой области, или зафиксированные в документах ведущих организаций.

**Программная инженерия – это**

- установление и использование обоснованных инженерных принципов (методов) для экономного получения ПО, которое надежно и работает на реальных машинах [7].
- та форма инженерии, которая применяет принципы информатики (computer science) и математики для рентабельного решения проблем ПО [8].
- применение систематического, дисциплинированного, измеряемого подхода к разработке, использованию и сопровождению ПО [9].
- дисциплина, целью которой является создание качественного ПО, которое завершается вовремя, не превышает выделенных бюджетных средств и удовлетворяет выдвигаемым требованиям [10].

В 1972 году IEEE<sup>2</sup> выпустил первый номер Transactions on Software Engineering – Труды по Программной Инженерии. Первый целостный взгляд на эту область

---

<sup>2</sup> IEEE - Institute for Electrical and Electronic Engineers (Институт инженеров по электронике и электротехнике)

профессиональной деятельности появился в 1979 году, когда Компьютерное Общество IEEE подготовило стандарт IEEE Std 730 по качеству программного обеспечения. После 7 лет напряженных работ, в 1986 году IEEE выпустило IEEE Std 1002 «Taxonomy of Software Engineering Standards».

**Программная инженерия (Software Engineering)** является *отраслью информатики (computer science)*, это инженерная дисциплина, которая изучает вопросы построения компьютерных программ, отражает закономерности развития программирования, обобщает опыт программирования в виде комплекса знаний и правил регламентации инженерной деятельности разработчиков ПО.

В этом определении выделим два основных аспекта.

1) Программную инженерию (ПИ) можно рассматривать как *инженерную дисциплину*, в которой инженеры применяют теоретические идеи, методы и средства при разработке ПО, создают продукты в соответствии со стандартами, регламентирующими процессы их проектирования и разработки.

2) ПИ описывает *методы управления программным проектом, качеством и рисками*. Применение таких методов позволяет достичь высокого качества программных продуктов. Эта инженерная дисциплина предоставляет всю необходимую информацию и стандарты для выбора наиболее подходящего метода и процессов жизненного цикла ПО для реализации конкретного проекта.

Как *инженерная дисциплина*, она охватывает все аспекты создания ПО, начиная от формирования требований до создания сопровождения и снятия с эксплуатации ПО, а также включает инженерные методы оценки трудозатрат, стоимости, производительности и качества, то есть речь идет именно об *инженерной деятельности в программировании*, поскольку ее сущность близка к определению инженерной деятельности в толковом словаре:

- *инженерия* - это способ применения научных результатов, что позволяет получать пользу от свойств материалов и источников энергии;
- *инженерия* - деятельность по созданию машин для предоставления полезных для потребителя услуг и изделий.

**Инженер** (франц. *ingenieur*, от лат. *ingenium* - способность, изобретательность), специалист с высшим техническим образованием. Первоначально - название лиц, управлявших военными машинами<sup>3</sup>.



Статуя инженера Р.Фултона в Капитолии

<sup>3</sup> Большая Советская энциклопедия

**Общие для всех инженерных дисциплин характеристики [11]**

[1] Инженеры в своей деятельности *принимают ряд решений*, тщательно оценивая альтернативы и выбирая в каждой точке принятия решения подход, *оптимально соответствующий решаемой задаче с учетом существующего контекста*. Выбор подхода осуществляется в процессе анализа альтернатив, во время которого тщательно сопоставляются возможные затраты и ожидаемая прибыль.

[2] Инженеры, по возможности, работают с использованием *измеримых количественных характеристик*; они совершенствуют и уточняют существующие методы измерений и при необходимости выдают приближенные решения на основе опыта и эмпирических данных.

[3] Инженеры придают особое значение использованию дисциплинированного процесса при осуществлении проекта и понимают важность вопросов *эффективной организации командной работы*.

[4] Инженеры могут отвечать за *выполнение самого широкого спектра задач*, начиная с исследований, разработки, проектирования, производства, тестирования, внедрения, эксплуатации и управления, и заканчивая продажами, консультированием и обучением.

[5] Инженеры в процессе выполнения своих обязанностей широко используют *инструментальные средства*. Поэтому выбор и использование подходящих средств является крайне важным вопросом.

[6] Объединяясь в профессиональные сообщества, инженеры способствуют развитию своей отрасли путем *разработки и внедрения рекомендаций*, аттестационных принципов, стандартов, распространению хорошо зарекомендовавших себя подходов (best practices).

[7] Инженеры *повторно используют результаты проектирования* и проектные решения (паттерны).

**Инженеры** в программной инженерии - это специалисты, выполняющие *практические работы* по реализации программ с применением теории, методов и средств компьютерной науки (computer science). Знание компьютерной науки необходимо специалистам в области программного обеспечения так же, как знание физики - инженерам-электронщикам. Если для решения конкретных задач программирования не существует подходящих методов или теории, инженеры применяют свои знания, накопленные ими в процессе разработок конкретных ПО, а также используют опыт

применения соответствующих инструментальных программных средств, и зачастую это оказывается наиболее эффективным способом построения высококачественных ПС.

Инженеры, как правило, работают в условиях заключенных контрактов и выполняют задачи проекта с учетом этих условий и ограничений на сроки, время, стоимость и др. В отличие от науки, цель которой - получение знаний, для инженерии знание - это способ получения некоторой **пользы**. Кроме программистов, занимающихся непосредственно разработкой ПО, в программной инженерии используются:

1. *менеджеры*, которые планируют и руководят проектом, отслеживают сроки и затраты;
2. *инженеры службы ведения библиотек* и репозитариев компонентов;
3. *технологи*, которые определяют инженерные методы и стандарты, создают для проекта модель ЖЦ, удовлетворяющую его целям и задачам;
4. *тестировщики* (контролеры), которые проверяют правильность выполнения процесса проектирования путем тестирования и на основе собранных данных проводят измерения разных характеристик качества, включая оценку надежности ПО;
5. *верификаторы*, которые проверяют правильность реализации функций в проекте;
6. *валидаторы*, проверяющие ПО на соответствие заданным требованиям.

Разработку программных систем можно считать инженерной деятельностью, но она имеет некоторые отличия от традиционной инженерии:

- ветви инженерии имеют высокую степень специализации, а в программной инженерии специализация коснулась только отдельных областей (например, операционные системы, трансляторы, редакторы и т.п.);
- программирование объектов основывается на стандартах, с помощью которых отражаются типовые требования заказчиков, т.е. типизация объектов и артефактов в сфере программирования;
- технические решения классифицированы и каталогизированы, а в программной инженерии каждая новая разработка - это новая проблема, для реализации которой устанавливают аналогию с ранее разработанными системами. Одним из инженерных решений каталогизации в программировании является *паттерн проектирования*.

Для превращения программной инженерии в специальность мировая компьютерная общественность создала профессиональные комитеты, регламентирующие аспекты

процесса программирования: ядро знаний SWEBOK<sup>4</sup> [12], этический кодекс программиста [13], учебные курсы (Curricula -2001, 2004) по подготовке специалистов в области программной инженерии, обучение специальности и сертификация специалистов.

Таким образом, возникновение программной инженерии как дисциплины разработки ПО определено следующими важными факторами:

- значительным объемом накопленных знаний в области создания ПО;
- появлением новых методов анализа, моделирования и проектирования ПО;
- совершенствованием методов обнаружения ошибок в ПО;
- эффективной организацией коллективов разработчиков ПО и оценки их трудовой деятельности;
- использованием готовых программных компонентов, высокотехнологических средств и инструментов разработки ПО;
- необходимостью эволюционного развития компонентов и систем, а также их адаптацией к новым изменяющимся условиям операционных сред и компьютерных сетей.

Программная инженерия делает акцент на повышении качества и производительности ПО за счет применения: новых и усовершенствованных методов проектирования ПО; готовых компонентов и методов их генерации; методов эволюции, верификации и тестирования ПО; инструментальных средств; методов управления проектами, оценки качества и стоимости.

### Область действия программной инженерии

Подведем итоги и ответим на часто задаваемые вопросы, касающиеся программной инженерии. В западной литературе часто используются термины: software engineering, system engineering и computer science. В чем разница?

Таблица 1.1 – Часто задаваемые вопросы о программной инженерии [5]

Вопрос	Ответ
Что такое программная инженерия?	Это <i>инженерная дисциплина</i> , охватывающая все аспекты разработки ПО.
В чем различие между ПИ и информатикой (computer science)?	Информатика охватывает теорию и методы построения вычислительных и программных систем, включая аппаратное и программное обеспечение, тогда как ПИ рассматривает вопросы <u>практического построения ПО</u> .

<sup>4</sup> SWEBOK - Software Engineering Body of Knowledge

В чем различие между ПИ и системотехникой (system engineering)?	Системотехника (точнее, технология создания вычислительных систем) охватывает все аспекты разработки вычислительных систем (включая создание аппаратных средств и ПО) и соответствующие технологические процессы. <u>Технологии ПИ являются частью этих процессов.</u>
Что такое <i>методы программной инженерии</i> ?	Это структурные решения, предназначенные для разработки ПО и включающие системные модели, формализованные нотации и правила проектирования, а также способы управления процессом создания ПО.
Что такое <i>технологический процесс создания ПО</i> ?	Это совокупность процессов, ведущих к созданию или развитию ПО.
Что такое <i>модель технологического процесса создания ПО</i> ?	Формализованное упрощенное представление технологического процесса создания ПО.
В чем отличие программной инженерии от других инженерных дисциплин?	ПИ качественно отличается от других инженерных дисциплин <u>нематериальностью программного обеспечения</u> и дискретной природой его функционирования. ПИ интегрирует принципы математики и информатики с инженерными подходами, разработанными для производства материальных объектов.

## 2 ОБЛАСТИ ЗНАНИЙ ПРОГРАММНОЙ ИНЖЕНЕРИИ

Рассмотрим взаимосвязанные аспекты инженерии ПО:

- **теоретический и интеллектуальный базис** (методы, принципы, средства и методологии и др.) проектирования, представленный в **ядре знаний SWEBOK**, способствующий созданию высококачественных программных продуктов, удовлетворяющих заданным заказчиком функциональным и нефункциональным требованиям;
- стандартный подход к разработке программных проектов, состоящий в использовании **моделей жизненного цикла** (ЖЦ), в процессы которых встроены методы проектирования, верификации, тестирования и оценивания промежуточных рабочих продуктов, а также проверки планов и времени выполнения работ на этих процессах для того, чтобы регулировать сроки и затраты, а также возможные риски и недостатки.

### **SE2004: совокупность знаний по программной инженерии**

1. Основы информационных технологий (CMP)
2. Основы математики и инженерии (FND)
3. Профессиональная практика (PRF)
4. Моделирование и анализ программного обеспечения (MAA)
5. Проектирование программного обеспечения (DES)
6. Верификация и аттестация программного обеспечения (VAV)
7. Эволюция программного обеспечения (EVL)
8. Процессы разработки программного обеспечения (PRO)
9. Качество программного обеспечения (QUA)
10. Управление программными проектами (MGT)

Ядро знаний SWEBOK является основополагающим научно-техническим документом, который отображает мнение многих зарубежных и отечественных специалистов в области программной инженерии и согласуется с современными регламентированными процессами ЖЦ ПО стандарта ISO/IEC 12207. В этом ядре знаний содержится *описание 10 областей*, каждая из которых представлена согласно принятой всеми участниками создания этого ядра общей схеме описания, включающей определение понятийного аппарата, методов и средств, а также инструментов поддержки инженерной деятельности. В каждой области описывается определенный запас знаний, который должен практически использоваться в соответствующих процессах ЖЦ.



Для наглядного представления понятийного аппарата областей знаний SWEBOK проведем условное разбиение областей на основные (пять для проектирования ПС, рисунок 2.1) и дополнительные организационные методы и подходы, которые отображают инженерию управления проектированием ПС (конфигурацией, проектами, качеством - рисунок 2.2).

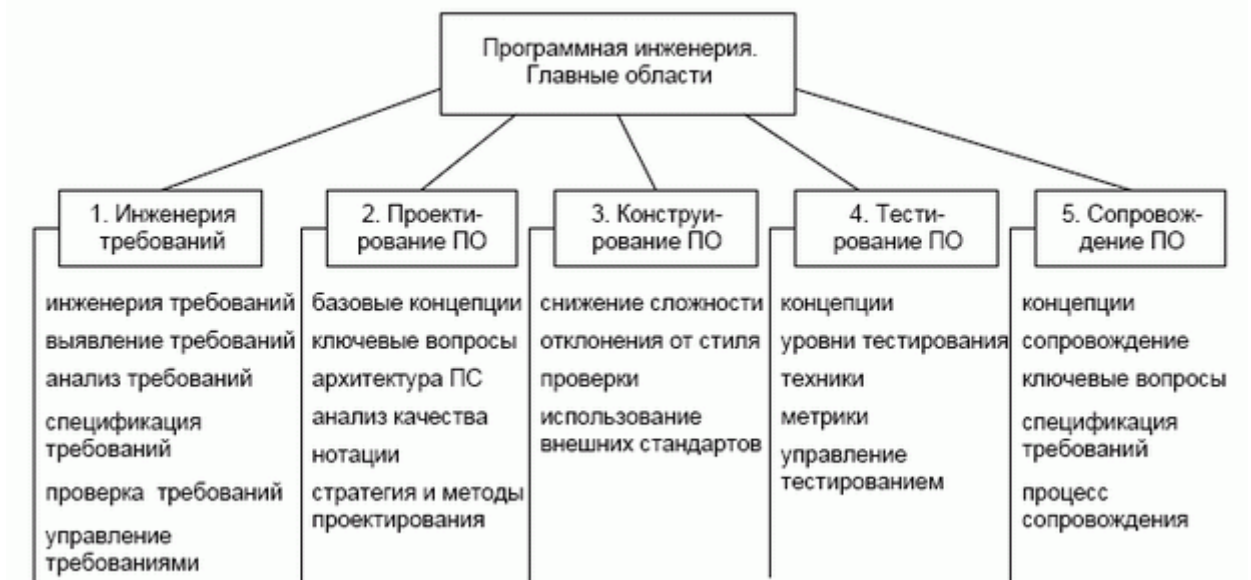


Рисунок 2.1 - Основные области знаний SWEBOK

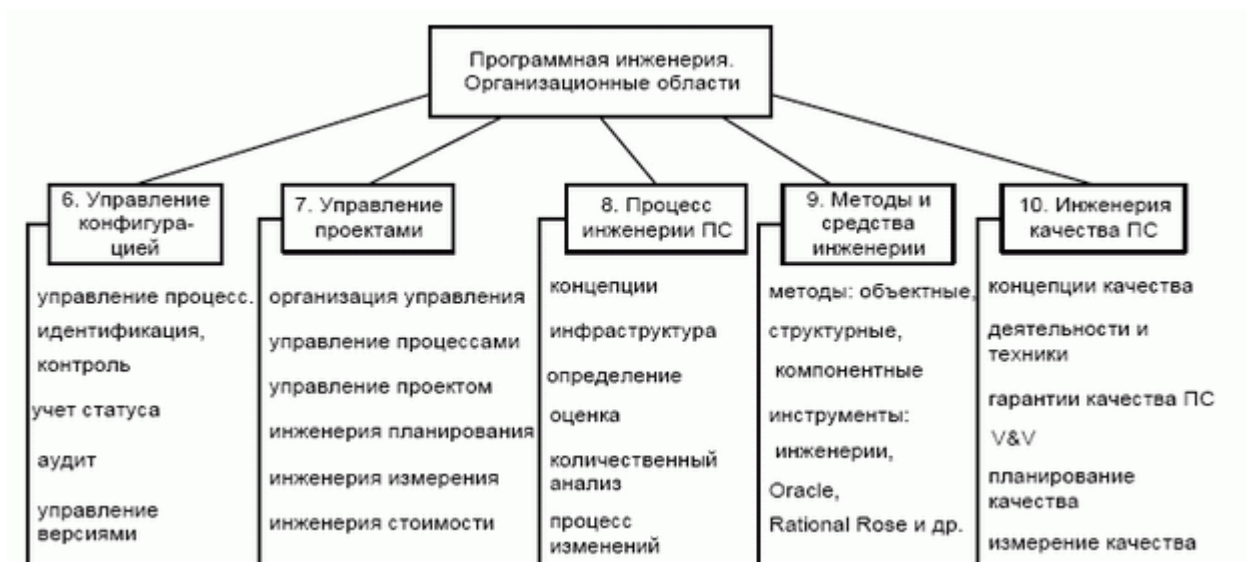


Рисунок 2.2 - Организационные области знаний SWEBOK

В каждой области приведены ключевые понятия, подходы и методы проектирования разных типов ПС. Данное разбиение областей на основные и вспомогательные соответствует структуре разбиения процессов стандарта ISO/IEC 12207 (см. раздел 1.2), выполнение которых определяется знаниями, содержащимися в ядре SWEBOK.

Далее приводится обзор каждой области ядра знаний SWEBOOK, определяется ее роль в проектировании и реализации программных продуктов. В некоторых подразделах показана связь с положениями соответствующих стандартов, которые регламентируют и регулируют выполнение процессов проектирования программных систем.

## Описание основных областей знаний SWEBOOK

### *Требования к ПО (Software Requirements)*

Первое, с чего необходимо начать данную тему, это дать определение понятию «требование» и выявить его основные характеристики. Обобщая определения, которые даются разными авторами и с учетом стандарта IEEE Standard Glossary of Software Engineering Terminology (1990), дадим следующее определение требования:

**Требования** - это свойства, которыми должно обладать ПО для адекватного определения функций, условий и ограничений выполнения ПО, а также объемов данных, технического обеспечения и среды функционирования.

Опыт индустрии информационных технологий показывает, что вопросы, связанные с управлением требованиями, оказывают критически-важное влияние на программные проекты, в определенной степени – на сам факт успешного завершения проектов. Только систематическая работа с требованиями позволяет корректным образом обеспечить моделирование предметной области (задач реального мира) и формулирование необходимых приемочных тестов для того, чтобы убедиться в соответствии создаваемых ПС критериям, заданным реальными практическими потребностями.

Требования отражают потребности заказчиков, пользователей и разработчиков, заинтересованных в создании ПО. Заказчик и разработчик совместно проводят сбор требований, их анализ, пересмотр, определение необходимых ограничений и документирование. Различают требования к *продукту* и к *процессу*, а также *функциональные*, *нефункциональные* и *системные* требования.

Требования к продукту и к процессу определяют условия функционирования и режимы работы ПО в операционной среде, ограничения на структуру и память компьютеров, на принципы взаимодействия программ и компьютеров и т.п.

*Функциональные требования* определяют назначение и функции системы, а *нефункциональные* - условия выполнения ПО и доступа к данным. *Системные требования* описывают требования к программной системе, состоящей из взаимосвязанных программных и аппаратных подсистем и разных приложений. Требования могут быть количественные (например, количество запросов в сек., средний

показатель ошибок и т.п.). Значительная часть требований относится к атрибутам качества: безотказность, надежность и др., а также к защите и безопасности как ПО, так и данных.

Рассмотрим классический пример высокоуровневого структурирования групп требований как требований к продукту, который предложен одним из классиков дисциплины управления требованиями Карлом Вигерсом [14].

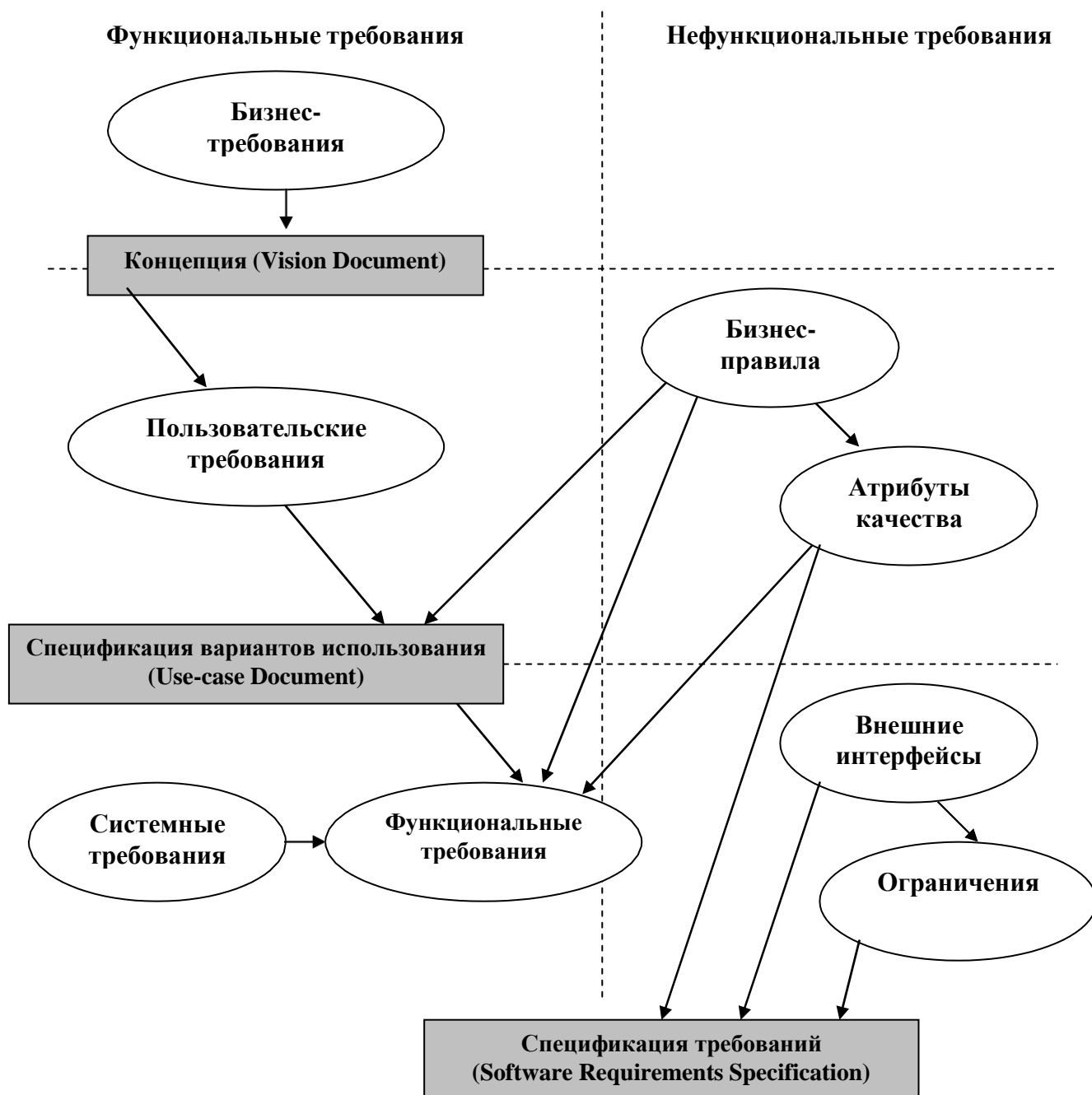


Рисунок 2.3 – Уровни требований по Вигерсу

Систематизируя работы ведущих экспертов в данной области (Вигерса, Лефингвелла и Видрига, Коберна), необходимо привести **классификацию различных категорий (видов) требований** и связанных с ними понятий, важнейших с точки зрения их понимания и дальнейшего практического применения:

- *Потребности (needs)* - отражают проблемы бизнеса, персоналии или процесса, которые должны быть соотнесены с использованием или приобретением системы.
- *Группа функциональных требований*
  - Бизнес-требования (Business Requirements) - определяют высокоуровневые цели организации или клиента (потребителя) - заказчика разрабатываемого программного обеспечения.
  - Пользовательские требования (User Requirements) - описывают цели/задачи пользователей системы, которые должны достигаться/выполняться пользователями при помощи создаваемой программной системы. Эти требования часто представляют в виде вариантов использования (Use Cases).
  - Функциональные требования (Functional Requirements) – определяют функциональность (поведение) программной системы, которая должна быть создана разработчиками для предоставления возможности выполнения пользователями своих обязанностей в рамках бизнес-требований и в контексте пользовательских требований.
- *Группа нефункциональных требований (Non-Functional Requirements)*
  - Бизнес-правила (Business Rules) - включают или связаны с корпоративными регламентами, политиками, стандартами, законодательными актами, внутрикорпоративными инициативами (например, стремление достичь зрелости процессов по СММІ 4-го уровня), учетными практиками, алгоритмами вычислений и т.д. На самом деле, достаточно часто можно видеть недостаточное внимание такого рода требованиям со стороны сотрудников ИТ-департаментов и, в частности, технических специалистов, вовлеченных в проект. Business Rules Group дает понимание бизнес-правила, как «положения, которые определяют или ограничивают некоторые аспекты бизнеса. Они подразумевают организацию структуры бизнеса, контролируют или влияют на поведение бизнеса». Бизнес-правила часто определяют распределение ответственности в системе, отвечая на вопрос «кто будет осуществлять конкретный вариант, сценарий

использования» или диктуют появление некоторых функциональных требований. В контексте дисциплины управления проектами (уже вне проекта разработки программного обеспечения, но выполнения бизнес-проектов и бизнес-процессов) такие правила обладают высокой значимостью и, именно они часто определяют ограничения бизнес-проектов, для автоматизации которых создается соответствующее программное обеспечение.

- Внешние интерфейсы (External Interfaces) - часто подменяются «пользовательским интерфейсом». На самом деле вопросы организации пользовательского интерфейса безусловно важны в данной категории требований, однако, конкретизация аспектов взаимодействия с другими системами, операционной средой (например, запись в журнал событий операционной системы), возможностями мониторинга при эксплуатации - все это не столько функциональные требования (к которым ошибочно приписывают иногда такие характеристики), сколько вопросы интерфейсов, так как функциональные требования связаны непосредственно с функциональностью системы, направленной на решение бизнес-потребностей.
- Атрибуты качества (Quality Attributes) - описывают дополнительные характеристики продукта в различных «измерениях», важных для пользователей и/или разработчиков. Атрибуты касаются вопросов портируемости, интероперабельности (прозрачности взаимодействия с другими системами), целостности, устойчивости и т.п.
- Ограничения (Constraints) - формулировки условий, модифицирующих требования или наборы требований, сужая выбор возможных решений по их реализации. В частности, к ним могут относиться параметры производительности, влияющие на выбор платформы реализации и/или развертывания (протоколы, серверы приложений, баз данных и т.п.), которые, в свою очередь, могут относиться, например, к внешним интерфейсам.
- *Системные требования* (System Requirements) иногда классифицируются как составная часть группы функциональных требований (не путайте с как таковыми «функциональными требованиями»). Описывают высокоуровневые требования к программному обеспечению, содержащему несколько или много

взаимосвязанных подсистем и приложений. При этом, система может быть как целиком программной, так и состоять из программной и аппаратной частей. В общем случае, частью системы может быть персонал, выполняющий определенные функции системы, например, авторизация выполнения определенных операций с использованием программно-аппаратных подсистем.

Необходимо сделать несколько важных замечаний по *бизнес-правилам*. Бизнес правила, как таковые, являются предметом пристального изучения различных специалистов в области как бизнес-моделирования, так и программной инженерии в целом. Практика разработки программных требований включает идентификацию и описание бизнес-правил как самостоятельных артефактов (например, методология RUP выделяет отдельный артефакт Business Rule в рамках дисциплины Business Modeling).

Наравне с представленной классификацией требований, могут использоваться и другие подходы. Даже в рамках этой классификации, существуют и различные взгляды на ее интерпретацию и детализацию. Например, как результат определения целевой аудитории и в рамках маркетинговой стратегии продвижения тиражируемого решения, возможно определять *высокоуровневые возможности (ключевые характеристики, особенности)* – *«фичи»* (features) разрабатываемого продукта. Вигерс, описывает feature как «множество логически связанных функциональных требований, которые предоставляют определенные возможности для пользователя и удовлетворяют бизнес-целям <организации>».

Анализируя различные источники на предмет работы с features, следует отметить следующее: с точки зрения инженерии требований, features являются самостоятельным артефактом, который может быть соотнесен как с функциональными требованиями, так и с нефункциональными (в т.ч. с ограничениями проектирования или атрибутами качества).

Необходимо также отметить, что features обладают определенным дуализмом в своей интерпретации, зависимым от контекста конкретного продукта – с одной стороны это может быть «тот самый список характеристик, указанный на коробке продукта» в случае создания «коробочного ПО», с другой стороны это может список высокоуровневых возможностей системы, например при заказной разработке ПО автоматизации бизнес-процессов конкретной организации.

SWEBOK охватывает не только вопросы структурирования и систематизации требований, но и различных процессов этапов и процессов работы с требованиями, а также некоторые практические соображения (см. рисунок 2.4).

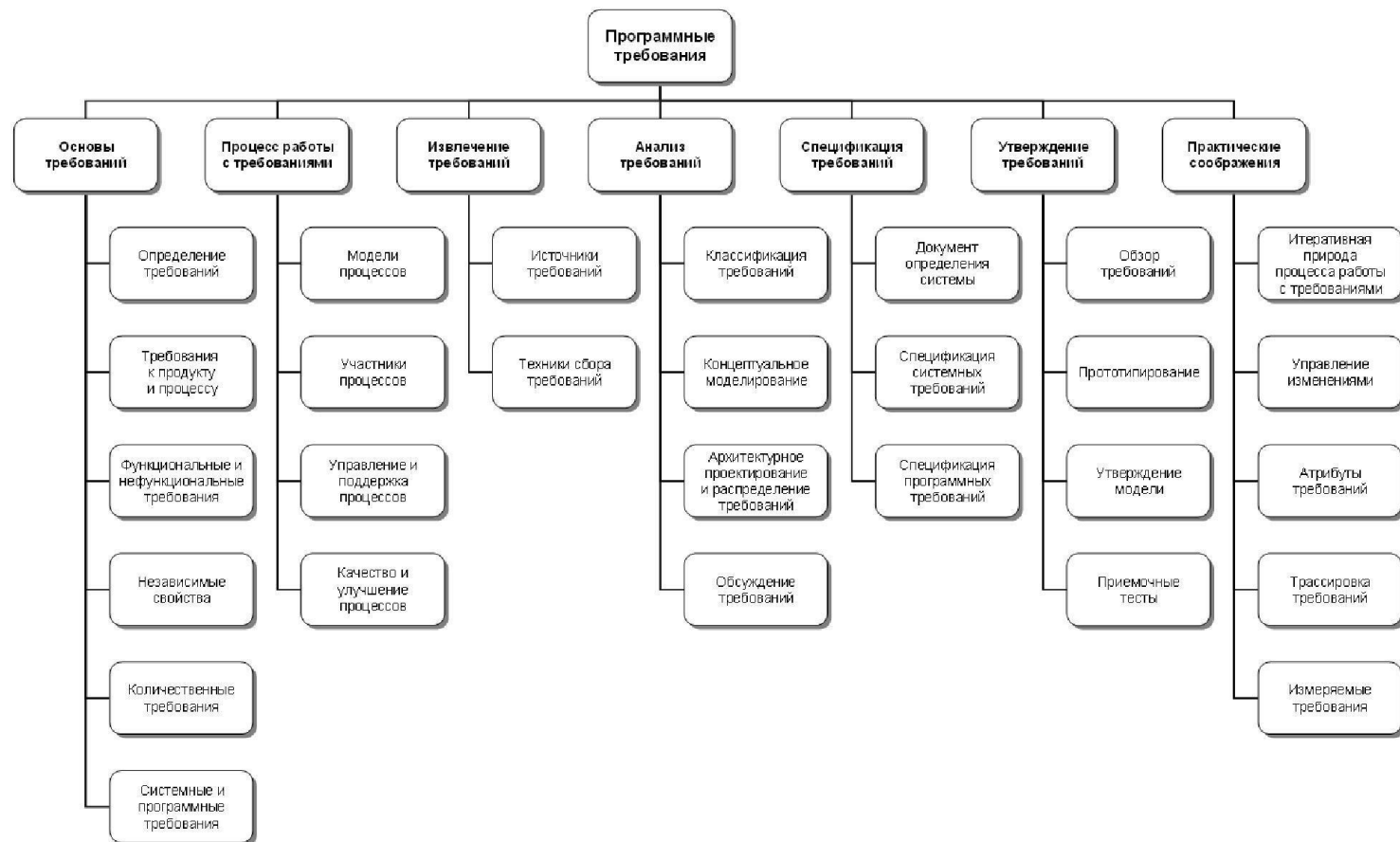


Рисунок 2.4 - Область знаний «Программные требования» [12]

Область знаний «Требования к ПО (Software Requirements)» состоит из следующих разделов:

- инженерия требований (Requirement Engineering);
- выявление требований (Requirement Elicitation);
- анализ требований (Requirement Analysis);
- спецификация требований (Requirement Specification);
- валидация требований (Requirement validation);
- управление требованиями (Requirement Management).

*Инженерия требований к ПО* - это дисциплина анализа и документирования требований к ПО, которая заключается в преобразовании предложенных заказчиком требований к системе в описании требований к ПО и их валидация. Она базируется на модели процесса определения требований и действующих лицах, обеспечивающих управление и формирование требований, а также на методах достижения показателей качества.

*Модель процесса определения требований* - это схема процессов ЖЦ, которые выполняются от начала проекта и до тех пор, пока не будут определены и согласованы требования. Данный процесс не является дискретным; это постоянно действующий процесс на всех этапах ЖЦ ПО. При этом процессом может быть маркетинг и проверка осуществимости требований в данном проекте.

Управление требованиями к ПО заключается в контроле за выполнением требований и планировании использования ресурсов (человеческих, программных, технических, временных, стоимостных) в процессе разработки промежуточных рабочих продуктов на этапах ЖЦ.

*Качество и процесс улучшения требований* - это процесс формулировки характеристик и атрибутов качества (надежность, реактивность и др.), которыми должно обладать ПО, методы их достижения на этапах ЖЦ и оценивания полученных результатов.

*Выявление требований* - это процесс извлечения информации из разных источников (договоров, материалов аналитиков по декомпозиции задач и функций системы и др.), проведения технических мероприятий (собеседований, собраний и др.) для формирования отдельных требований к продукту и к процессу разработки. Исполнитель должен согласовать требования с заказчиком.

*Анализ требований* - процесс изучения потребностей и целей пользователей, классификация и преобразование их к требованиям к системе,



аппаратуре и ПО,

установление и разрешение конфликтов между требованиями, определение приоритетов, границ системы и принципов взаимодействия со средой функционирования. Как говорилось ранее, требования могут быть функциональные и нефункциональные, которые определяют соответственно внешние и внутренние характеристики системы.

*Спецификация требований к ПО* - процесс формализованного описания функциональных и нефункциональных требований, требований к характеристикам качества в соответствии со стандартом качества ISO/IEC 9126-94, которые будут отрабатываться на этапах ЖЦ ПО.

В спецификации требований отражается:

- структура ПО;
- требования к функциям, качеству и документации;
- задается в общих чертах архитектура системы и ПО, алгоритмы, логика управления и структура данных.

Специфицируются также системные требования, нефункциональные требования и требования к взаимодействию с другими компонентами и платформами (БД, СУБД, маршаллинг данных, сеть и др.).

*Валидация требований* - это проверка изложенных в спецификации требований, выполняющаяся для того, чтобы путем отслеживания источников требований убедиться, что они определяют именно данную систему. Заказчик и разработчик ПО проводят экспертизу сформированного варианта требований с тем, чтобы разработчик мог далее проводить проектирование ПО. Один из методов валидации - *прототипирование*, т.е. быстрая отработка отдельных требований на конкретном инструменте и исследование масштабов изменения требований, измерение объема функциональности и стоимости, а также создание моделей оценки зрелости требований.

*Верификация требований* - это процесс проверки правильности спецификаций требований на их соответствие, непротиворечивость, полноту и выполнимость, а также на соответствие стандартам. В результате проверки требований делается согласованный выходной документ, устанавливающий полноту и корректность требований к ПО, а также возможность продолжить проектирование ПО.

*Управление требованиями* - это руководство процессами формирования требований на всех этапах ЖЦ и включает управление изменениями и атрибутами требований, а также проведение мониторинга - восстановления источника требований. Управление изменениями возникает после того, когда ПО

начинает работать в заданной среде и обнаруживаются ошибки в трактовке требований, либо в невыполнении

некоторого отдельного требования и т.п. Неотъемлемой составляющей процесса управления является трассирование требований для отслеживания правильности задания и реализации требований к системе и ПО на этапах ЖЦ, а также обратный процесс отслеживания в полученном продукте реализованных требований. Для исправления некоторых требований или добавления нового требования составляется план изменения требований, который согласуется с заказчиком. Внесенные изменения влекут за собой и изменения в созданный продукт или в отдельные его компоненты.

Говоря о требованиях, нельзя не сказать об *участниках процессов* (Process Actors). Для них вводится понятие «роли» и дается понимание “ролей” для людей, которые участвуют в процессе работы с требованиями. Таких людей также называют «заинтересованными лицами» (в данном контексте - software stakeholders). Заинтересованное лицо - некто, имеющий возможность (в том числе, материальную) повлиять на реализацию проекта/продукта.

Типичные примеры ролей:

- *Пользователи* (Users): группа, охватывающая тех людей, кто будет непосредственно использовать программное обеспечение; пользователи могут описать задачи, которые они решают (планируют решать) с использованием программной системы, а также ожидания по отношению к атрибутам качества, отображаемые в пользовательских требованиях.
- *Заказчики* (Customers): те, кто отвечают за заказ программного обеспечения или, в общем случае, являются целевой аудиторией на рынке программного обеспечения (образуют целевой рынок ПО);
- *Аналитики* (Market analysts): продукты массового рынка программного обеспечения (как и других массовых рынков, например, бытовой техники) не обладают «заказчиками» в понимании персонификации тех, кто «заказывает разработку». В то же самое время, лица, отвечающие за маркетинг, нуждаются в идентификации потребностей и обращению к тем, кто может играть роль <квалифицированных> «представителей» потребителей;
- *Регуляторы* (Regulators): многие области применения («домены») являются регулируемыми, например, телекоммуникации или банковский сектор. Программное обеспечение для ряда целевых рынков (в первую очередь, корпоративного сектора) требует соответствия руководящим документам и прохождения процедур, определяемых уполномоченными органами.

- *Инженеры по программному обеспечению, инженеры-программисты* (Software Engineer): лица, обладающие обоснованным интересом к разработке программного обеспечения, например, повторному использованию тех или иных компонент, библиотек, средств и инструментов. Именно инженеры ответственны за техническую оценку путей решения поставленных задач и последующую реализацию требований заказчиков.

Один из ключевых принципов программной инженерии заключается в обеспечении взаимодействия между пользователями и инженерами. Прежде, чем начинается разработка программного обеспечения, именно специалисты «по требованиям» – аналитики перекидывают тот самый «мостик» между заказчиками и исполнителями, который задает тот уровень коммуникаций и взаимопонимания между ними, который необходим для решения задач проекта.

### ***Проектирование ПО (Software design)***

**Проектирование ПО** - это процесс определения архитектуры, компонентов, интерфейсов, других характеристик системы и конечного состава программного продукта.

Результат процесса проектирования – дизайн. Рассматриваемое как процесс, проектирование есть инженерная деятельность в рамках ЖЦ ПО, в которой надлежащим образом анализируются требования для создания описания *внутренней структуры ПО*, являющейся основой для конструирования ПО как такового. Программный дизайн (как результат деятельности по проектированию) должен описывать архитектуру программного обеспечения, то есть представлять декомпозицию программной системы в виде организованной структуры компонент и интерфейсов между компонентами. Важнейшей характеристикой готовности дизайна является тот уровень детализации компонентов, который позволяет заняться их конструированием. Термины дизайн и архитектура могут использоваться взаимозаменяемым образом, но чаще говорят о дизайне как о целостном взгляде на архитектуру системы. Проектирование играет важную роль в процессах жизненного цикла создания программного обеспечения (Software Development Life Cycle), например, IEEE и ISO/IEC (ГОСТ Р ИСО.МЭК) 12207 [4].

Проектирование программных систем можно рассматривать как деятельность, результат которой состоит из двух составных частей:

- *Архитектурный или высокоуровневый дизайн* (software architectural design, top-level design) - описание высокоуровневой структуры и организации компонентов системы;

- *Детализированная архитектура* (software detailed design) - описывающая каждый компонент в том объеме, который необходим для конструирования.

Данная область знаний не описывает все сущности или понятия, имеющие в своем названии слово «дизайн» или «архитектура». В 1999 году Том ДеМарко (Tom DeMarco), один из известных специалистов в программной инженерии, предложил терминологическое разделение различных видов дизайна:

- **D-дизайн** (D-design, decomposition design) - декомпозиция структуры программного обеспечения в виде набора фрагментов или компонент;
- **FP-дизайн** (FP-design, family pattern design) - семейство архитектурных представлений, базирующихся на шаблонах;
- **I-дизайн** (I-design, invention) - создание высоко-уровневой концепции, видения того, что из себя будет представлять программная система; данный вид дизайна является результатом процесса анализа требований и их трансформации в подходы к реализации.

Область знаний «Проектирование ПО (Software Design)» состоит из шести следующих разделов (см. рисунок 2.4):

- 2) базовые концепции проектирования ПО (Software Design Basic Concepts);
- 3) ключевые вопросы проектирования ПО (Key Issue in Software Design);
- 4) структура и архитектура ПО (Software Structure and Architecture);
- 5) анализ и оценка качества проектирования ПО (Software Design Quality Analysis and Evaluation);
- 6) нотации проектирования ПО (Software Design Notations);
- 7) стратегия и методы проектирования ПО (Software Design Strategies and Methods).

*Базовая концепция проектирования ПО* - это методология проектирования архитектуры с помощью разных методов (объектного, компонентного и др.), процессы ЖЦ (стандарт ISO/IEC 12207 [4]) и техники - декомпозиция, абстракция, инкапсуляция и др. На начальных стадиях проектирования предметная область декомпозируется на отдельные объекты (при объектно-ориентированном проектировании) или на компоненты (при компонентном проектировании). Для представления архитектуры программного обеспечения выбираются соответствующие артефакты (нотации, диаграммы, блок-схемы и методы).

При проектировании ПО необходимо в первую очередь сформулировать цели архитектуры. Например, архитектурный фреймворк TOGAF, разработанный и

развиваемый консорциумом The Open Group [15], предлагает следующие <возможные> цели:

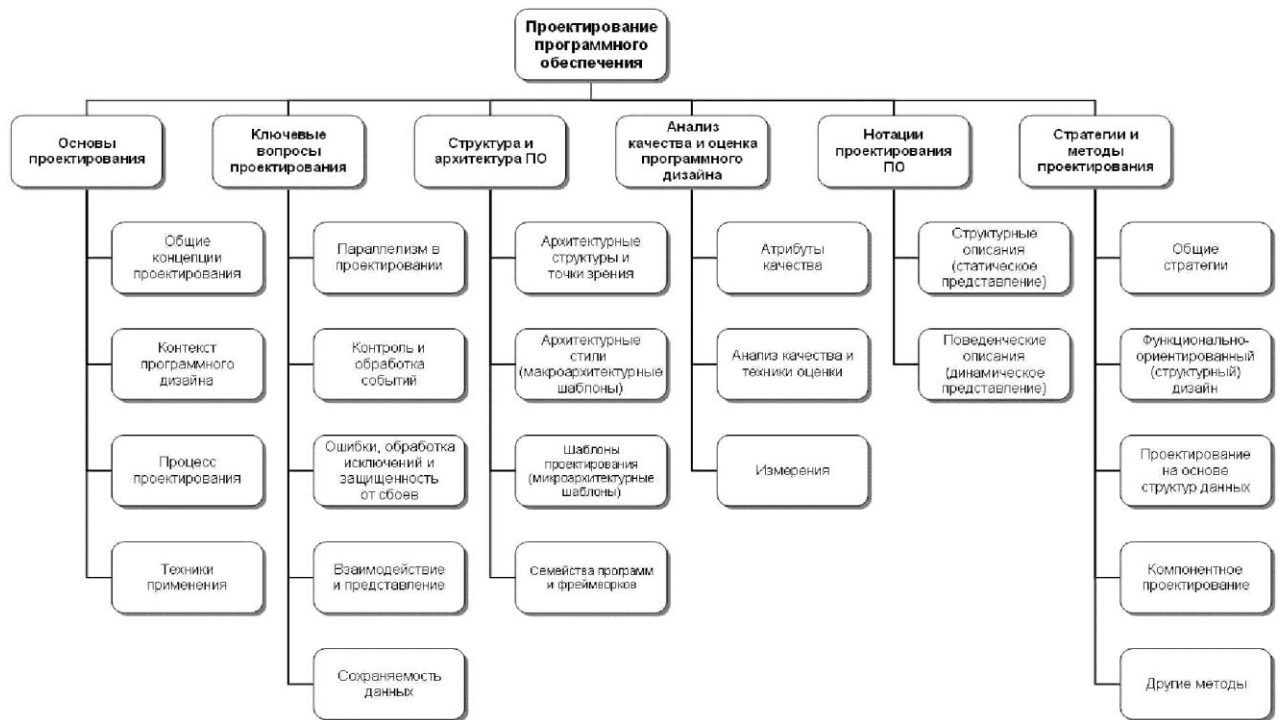


Рисунок 2.4 – Область знаний «Проектирование программного обеспечения» [12]

- ✓ Улучшение и повышение продуктивности бизнес-процессов;
- ✓ Уменьшение затрат;
- ✓ Улучшение операционной бизнес-деятельности;
- ✓ Повышение эффективности управления;
- ✓ Уменьшение рисков;
- ✓ Повышение эффективности ИТ-организации;
- ✓ Повышение продуктивности работы пользователей;
- ✓ Повышение интероперабельности (возможности и прозрачности взаимодействия);
- ✓ Уменьшение стоимости <поддержки> жизненного цикла;
- ✓ Улучшение характеристик безопасности;
- ✓ Повышение управляемости.

Стандарты жизненного цикла, например, IEEE и ISO/IEC (ГОСТ Р) 12207 уделяют специальное внимание вопросам проектирования и детализируют их, описывая контекст проектирования - от требований до тестов.

Отдельно необходимо остановиться на техниках проектирования, которые являются ключевыми идеями и концепциями, рассматриваемыми на фундаментальном уровне в различных методах и подходах к проектированию программного обеспечения.

#### *Абстракция (Abstraction)*

В контексте проектирования программных систем существует два механизма абстракции – параметризация и специфицирование (может интерпретироваться как детализация). При этом, абстракция через специфицирование бывает трех видов: *процедурная абстракция* (динамическая, то есть в отношении поведения), *абстракция данных* (статическая, то есть в отношении информации) и *абстракция контроля* (то есть управления системой и обрабатываемой ею информацией).

#### *Связанность и соединение (Coupling and Cohesion)*

*Связанность (Coupling)* – определяет силу связи (часто, взаимного влияния) между модулями. *Соединение (Cohesion)* – определяет как тот или иной элемент обеспечивает связь внутри модуля, внутреннюю связь.

#### *Декомпозиция и разбиение на модули (Decomposition and Modularization)*

Декомпозиция и разбиение на модули сложных программных систем производится с целью получения более мелких и относительно независимых программных компонентов, каждый из которых несет различную функциональность (логически связанные группы функциональности).

#### *Инкапсуляция/сокрытие информации (Encapsulation/information hiding)*

Данная концепция предполагает группировку и упаковку (с точки зрения подготовки к развертыванию и эксплуатации) элементов и внутренних деталей абстракции (то есть модели) в отношении реализации с тем, чтобы эти детали (как малозначимые для использования компонента или по другим причинам) были недоступны пользователям элементов (компонент). При этом, в качестве «пользователя» одного компонента может выступать другой компонент. Более того, при использовании объектно-ориентированного подхода, наследники компонентов могут не иметь доступа ко внутренним деталям реализации компонента, который является их предком (зависит от объектно-ориентированной модели конкретного языка программирования или платформы).



*Разделение интерфейса и реализации (Separation of interface and implementation)*

Данная техника предполагает определение компонента через специфицирование интерфейса, известного (описанного) и доступного клиентам (или другим компонентам), от непосредственных деталей реализации.

*Достаточность, полнота и простота (Sufficiency, completeness and primitiveness)*

Этот подход подразумевает, что создаваемые программные компоненты обладают всеми необходимыми характеристиками, определенными абстракцией (моделью), но не более того. То есть не включают функциональность, отсутствующую в модели.

*Сохраняемость данных (Data Persistence)*

Именно сохраняемость, а не сохранность, так как тема касается не доступа к базам данных, как такового, а также не гарантий сохранности информации. Суть вопроса – как должны обрабатываться «долгоживущие» данные.

К ключевым вопросам проектирования ПО относятся: декомпозиция программ на функциональные компоненты для независимого и параллельного их выполнения, принципы распределения компонентов в среде выполнения и их взаимодействия между собой, механизмы обеспечения качества и живучести системы и др.

При проектировании архитектуры ПО используется *архитектурный стиль проектирования*, основанный на определении основных элементов структуры - подсистем, компонентов и связей между ними.

В строгом значении *архитектура ПО* (software architecture) - высокоуровневое представление структуры системы и спецификация ее компонентов. Архитектура определяет логику отдельных компонентов системы настолько детально, насколько это необходимо для написания кода, а также определяет связи между компонентами. Существуют и другие виды представления структур, основанные на проектировании образцов, шаблонов, семейств программ и каркасов программных сред.

Вопросы организации архитектуры ПО стали складываться в самостоятельную и достаточно обширную дисциплину в середине 90-х, на волне распространения клиент-серверного подхода и начала его трансформации в «многозвенный клиент-сервер», призванный обеспечить централизованное развертывание и управление общей (для клиентских приложений) бизнес-логикой.

В итоге, уже на сегодняшний день накоплен целый комплекс подходов и созданы различные архитектурные «фреймворки», то есть систематизированные комплексы

методов, практик и инструментов, призванные в той или иной степени формализовать имеющийся в индустрии опыт (как положительный - например, design patterns, так и отрицательный - например, anti-patterns).

Приведем примеры систематизации в форме фреймворков:

- TOGAF - The Open Group Architecture Framework [15];
- Модель Захмана - Zachman Framework [16];
- Руководство по архитектуре электронного правительства E-Gov Enterprise Architecture Guidance [17].

Любая система может рассматриваться с разных точек зрения, поэтому вводится такое понятие, как *архитектурное представление (view)*. Оно может быть структурным, включающим типовые композиции структур из объектов и классов, объектов, связей и др.; поведенческим, определяющим схемы взаимодействия классов объектов и их поведение диаграммами активностей, взаимодействия, потоков управления и др.; порождающим, отображающим типовые схемы распределения ролей экземпляров объектов и способы динамической генерации структур объектов и классов.

Одним из важнейших инструментов проектирования архитектуры является *паттерн* - типовой конструктивный элемент ПО, который задает взаимодействие объектов (компонентов) проектируемой системы, а также роли и ответственности исполнителей<sup>5</sup>.

Проектирование можно рассматривать на разных уровнях:

- ✓ *Макро-уровень* – тогда говорят об архитектурном стиле, проектирование рассматривается на уровне модулей, «крупноблочного» взгляда. Например, архитектура распределенной сервисно-ориентированной системы может строиться в стиле обмена сообщениями через соответствующие очереди сообщений, может проектироваться на основе идеи взаимодействия между компонентами и приложениями через общую объектную шину, а может использовать концепцию брокера как единого узла пересылки запросов. В то же время, на более концептуальном уровне, мы можем говорить о выборе клиент-серверного стиля или распределенного стиля архитектуры системы. Таким образом, архитектурный стиль - набор ограничений, определяющих семейство архитектур, которые удовлетворяют этим ограничениям.

---

<sup>5</sup> Наиболее краткая формулировка того, что такое шаблон проектирования, может звучать так – «общее

---

решение общей проблемы в заданном контексте».

- ✓ *Микро-уровень* – в этом случае определяют частные аспекты деталей архитектуры, т.е. применяют шаблоны проектирования. Чаще всего говорят о следующих группах шаблонов проектирования:
  - *Шаблоны создания* (Creational patterns) - builder, factory, prototype, singleton;
  - *Структурные шаблоны* (Structural patterns) - adapter, bridge, composite, decorator, façade, flyweight, proxy;
  - *Шаблоны поведения* (Behavioral patterns) - command, interpreter, iterator, mediator, memento, observer, state, strategy, template, visitor.

*Анализ и оценка качества проектирования ПО* включает мероприятия по анализу сформулированных в требованиях атрибутов качества, оценки различных аспектов ПО - количества функций, структура ПО, качества проектирования с помощью формальных метрик (функционально-ориентированных, структурных и объектно-ориентированных), а также проведения качественного анализа результатов проектирования путем статического анализа, моделирования и прототипирования.

Существует целый спектр различных атрибутов, помогающих оценить и добиться качественного дизайна. Эти атрибуты могут описывать многие характеристики системы и элементов дизайна как такового: тестируемость, переносимость, модифицируемость, производительность, безопасность и т.п. Важно понимать, что обсуждаемые атрибуты касаются только дизайна (как результата), но не проектирования (как процесса). В принципе, все эти атрибуты можно разбить на несколько групп:

- *применимые к run-time*, то есть ко времени выполнения системы; например, среднее время отклика системы позволяющий оценить качество дизайна с точки зрения производительности;
- *ориентированные на design-time*, то есть позволяющие оценивать качество получаемого дизайна еще на этапе проектирования или, в общем случае, вплоть до тестирования, включительно; например, средняя нагруженность классов бизнес-методами (предположим бизнес-методов в каждом классе в среднем 30 - интересно, насколько легко можно поддерживать, модифицировать и развивать систему с такой внутренней структурой);
- *атрибуты качества архитектурного дизайна как такового*, например, концептуальная целостность дизайна, непротиворечивость, полнота, завершенность; например, любой определенный бизнес-метод является вызываемым, т.е. создан не просто потому, что может понадобиться в будущем, а определен в соответствии с требованиями

или необходим для реализации дизайна в выбранном архитектурном стиле.

*Нотации проектирования* позволяют представить описание объекта (элемента) ПО и его структуру, а также поведение системы. Существует два типа нотаций: структурные, поведенческие и множество различных их представлений.

Структурные нотации - это структурное, блок-схемное или текстовое представление аспектов проектирования структуры ПО из объектов, компонентов, их интерфейсов и взаимосвязей. К нотациям относятся формальные языки спецификаций и проектирования: ADL (Architecture Description Language), UML (Unified Modeling Language), ERD (Entity-Relation Diagrams), IDL (Interface Description Language), Use Case Driven и др. Нотации включают языки описания архитектуры и интерфейса, диаграммы классов и объектов, диаграммы сущность-связь, конфигурации компонентов, схем развертывания, а также структурные диаграммы, задающие в наглядном виде операторы цикла, ветвления, выбора и последовательности.

Поведенческие нотации отражают динамический аспект поведения систем и их компонентов. Таким нотациям соответствуют диаграммы потоков данных (Data Flow), таблиц принятия решений (Decision Tables), деятельности (Activity), кооперации (Colloboration), последовательности (Sequence), пред- и постусловия (Pre-Post Conditions), формальные языки спецификации (Z, VDM, RAISE), языки проектирования PDL и др.

#### *Стратегия и методы проектирования ПО.*

К общим стратегиям, помогающим в проведении работ по проектированию, относятся:

- «разделяй-и-властвуй» и пошаговое уточнение;
- проектирование «сверху-вниз» и «снизу-вверх»;
- абстракция данных и сокрытие информации;
- итеративный и инкрементальный подход;
- и другие...

Методы в данном контексте это не просто некие «слабоформализованные» или просто частные практические подходы или техники. Методы здесь являются более общими понятиями, это - *методологии*, сконцентрированные на процессе (в частности, проектирования) и предполагающие следование определенным правилам и соглашениям, в том числе по используемым выразительным средствам. Такие методы полезны как инструмент систематизации (иногда, формализации) и передачи знаний в виде общего фреймворка (то есть комплексного набора понятий, подходов, техник и инструментов) не только для отдельных специалистов, но для команд и проектных групп программных проектов.

Рассмотрим основные методы:

- *функционально-ориентированные* (структурные), которые базируются на структурном анализе, структурных картах, диаграммах потоков данных (Dataflow) и др. Они ориентированы на идентификацию функций и их уточнение «сверху-вниз», после чего уточняются диаграммы потоков данных и проводится описание процессов.
- в *объектно-ориентированном проектировании* ключевую роль играет наследование, полиморфизм и инкапсуляция, а также абстрактные структуры данных и отображение объектов.
- *подходы, ориентированные на структуры данных*, базируются на методе Джексона [1.8] и используются для задания входных и выходных данных структурными диаграммами. Метод UML предназначен для сценарного моделирования проекта [1.19] в наглядном диаграммном виде.
- *Компонентное проектирование* ориентировано на использование готовых компонентов (reusing), их интерфейсов и интеграцию для формирования конфигурации, служащей основой развертывания компонентного ПО и взаимодействия компонентов в операционной среде.
- *Формальные методы* описания программ основываются на спецификациях, аксиомах, описаниях некоторых условий, называемых предварительными (предусловиями), утверждениях и постусловиях, определяющих заключительное условие получения правильного результата программой. Спецификация является формальным описанием функций и данных программы, с которыми эти функции оперируют. Различают входные и выходные параметры функции, а также данные, которые не привязаны к реализации и определяют интерфейс с другими функциями. По формальным спецификациям, условиям и утверждениям выполняется доказательство правильности программы.

### ***Конструирование ПО (Software Construction)***

**Конструирование ПО** - создание работающего ПО с привлечением методов верификации, кодирования и тестирования компонентов.

Процесс конструирования ПО затрагивает важные аспекты деятельности по проектированию и тестированию. Кроме того, конструирование отталкивается от результатов проектирования, а тестирование (в любой своей форме) предполагает работу с результатами конструирования. Достаточно сложно определить границы между

проектированием, конструированием и тестированием, так как все они связаны в единый комплекс процессов ЖЦ ПО и, в зависимости от выбранной модели жизненного цикла и применяемых методов (методологии), такое разделение может выглядеть по-разному.

К инструментам конструирования ПО отнесены языки программирования и конструирования, а также программные методы и инструментальные системы (компиляторы, СУБД, генераторы отчетов, системы управления версиями, конфигурацией, тестированием и др.). К формальным средствам описания процесса конструирования ПО, взаимосвязей между человеком и компьютером и с учетом среды окружения отнесены, например, структурные диаграммы Джексона.

Область знаний «Конструирование ПО (Software Construction)» включает следующие разделы (см. рисунок 2.5):

- Основы конструирования:
  - снижение сложности (Reduction in Complexity);
  - предупреждение отклонений от стиля (Anticipation of Diversity);
  - структуризация проверок (Structuring for Validation);
  - использование внешних стандартов (Use of External Standards).
- Управление конструированием:
  - модели конструирования;
  - планирование конструирования;
  - измерения в конструировании;
- Практические соображения:
  - проектирование в конструировании;
  - языки конструирования;
  - кодирование;
  - тестирование;
  - повторное использование;
  - качество;
  - интеграция.

#### *Основы конструирования*

*Снижение сложности* - это минимизация, уменьшение и локализация сложности конструирования.

Минимизация сложности определяется ограниченными возможностями исполнителей обрабатывать сложные структуры и большие объемы информации на протяжении длительного периода времени. Минимизация сложности достигается, в



частности, использованием в процессе конструирования модулей и других более простых элементов, а также рекомендациями стандартов.

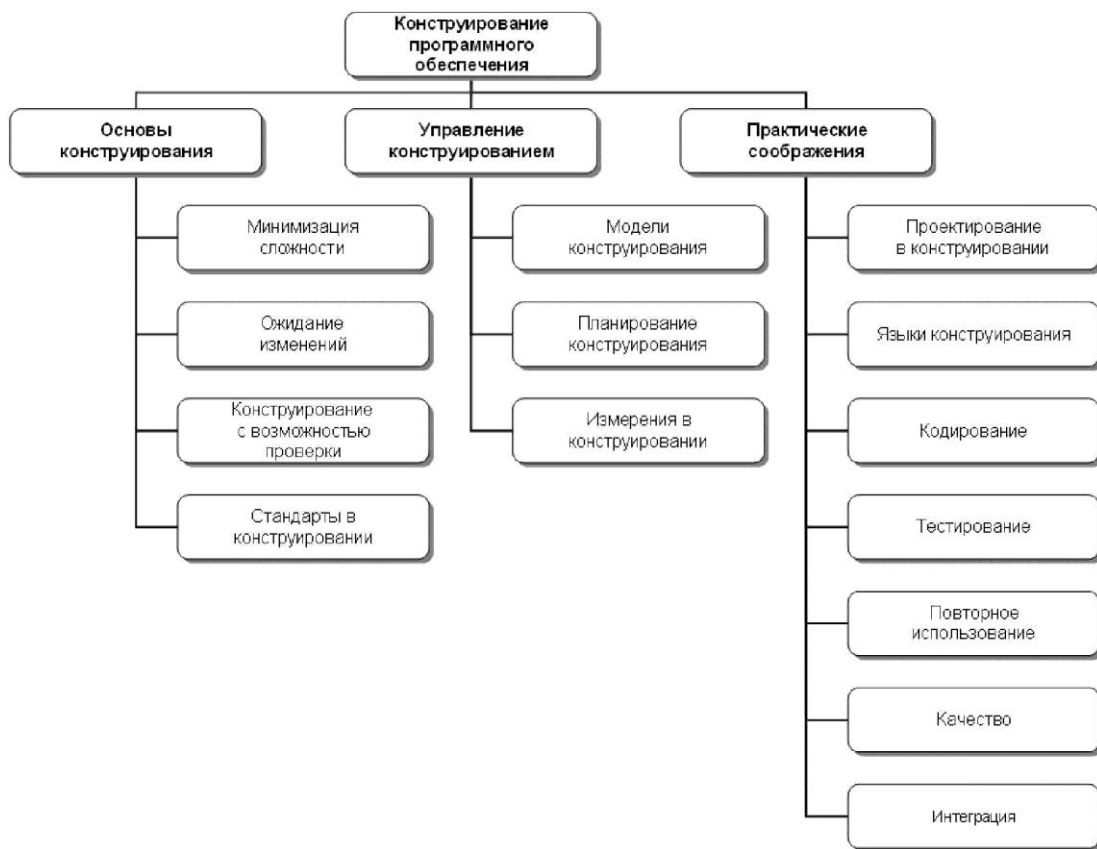


Рисунок 2.5 - Область знаний «Конструирование программного обеспечения» [12]

Уменьшение сложности в конструировании ПО достигается при создании простого и легко читаемого кода для придания большей значимости этому коду, простоте тестирования, производительности и удовлетворению заданных критериев. Это влияет на функциональность, характеристики и ограничения проекта. Потребность в уменьшении сложности влияет на все аспекты конструирования и особенно критична для процессов верификации (проверки) и тестирования результатов конструирования элементов ПС.

Локализация сложности - это способ конструирования ПО с применением объектно-ориентированного подхода, который лимитирует интерфейс объектов, упрощает их взаимодействие, также проверку правильности самих объектов и связей между ними. Локализация упрощает внесение изменений, связанных с обнаруженными ошибками в коде, либо источником ошибок является среда, в которой выполняется код.

*Предупреждение отклонений от стиля.* Для решения различных задач конструирования применяются разные стили конструирования (лингвистический, формальный, визуальный).

Лингвистический стиль основан на использовании словесных инструкций и выражений для представлений отдельных элементов (конструкций) программ. Он используется при конструировании несложных конструкций и приводится к виду традиционных функций и процедур, логическому и функциональному их программированию и др.

Формальный стиль используется для точного, однозначного и формального определения компонентов системы. В результате его применения обеспечивается конструирование сложных систем с минимальным количеством ошибок, которые могут возникнуть в связи с неоднозначностью определений или обобщений при конструировании ПО неформальными методами.

Визуальный стиль является наиболее универсальным стилем конструирования ПО, он позволяет разработчикам проекта представлять конструируемый элемент в наглядном виде. Визуальный язык проектирования UML представляет разработчику набор удобных диаграмм для задания статической и динамической структуры ПО. При применении визуального стиля конструирования создается текстовое и диаграммное описание конструктивных элементов ПО, которое выводится на экран дисплея не только для их наглядного представления, но и корректировки.

*Структуризация проверок* предполагает, что построение ПС должно проводиться таким образом, чтобы сама система помогала вести поиск ошибок, дефектов и причин сбоев при применении различных методов проверки, как на стадии независимого тестирования (например, инженерами-тестировщиками), так и в процессе эксплуатации, когда особенно важно быстрое обнаружение и исправление возникающих ошибок.

Среди техник, направленных на достижение такого результата конструирования:

- обзор, оценка кода (code review);
- модульное тестирование (unit-testing);
- структурирование кода совместно с применением автоматизированных средств тестирования (automated testing);
- ограниченное применение сложных или тяжелых для понимания языковых структур.

*Использование внешних стандартов.* Конструирование ПО зависит от применения внешних стандартов, связанных с языками программирования, инструментальными средствами и интерфейсами. При конструировании должен быть определен достаточный и полный набор стандартов для руководства и обеспечения координации между

определенными видами деятельности, группами операций, минимизации сложности, внесения изменений, анализа рисков и др.

Иными словами, внешние и внутренние стандарты должны определить общие правила работы членов проектной команды с учетом процессов ЖЦ.

Стандарты, которые напрямую применяются при конструировании, включают:

- коммуникационные методы (например, стандарты форматов документов и <оформления> содержания);
- языки программирования и соответствующие стили кодирования (например, Java Language Specification, являющийся частью стандартной документации JDK - Java Development Kit и Java Style Guide, предлагающий общий стиль кодирования для языка программирования Java);
- платформы (например, стандарты программных интерфейсов для вызовов функций операционной среды, такие как прикладные программные интерфейсы платформы Windows - Win32 API, Application Programming Interface или .NET Framework SDK, Software Development Kit);
- инструменты (не в терминах сред разработки, но возможных средств конструирования, например, UML как один из стандартов для определения нотаций для диаграмм, представляющих структура кода и его элементов или некоторых аспектов поведения кода).

Стандарты создаются разными источниками, например, *консорциумом OMG* - Object Management Group (в частности, стандарты CORBA, UML, MDA, ...), *международными организациями по стандартизации* такими, как ISO/IEC, IEEE, TMF, ..., *производителями платформ, операционных сред* и т.д. (например, Microsoft, Sun Microsystems, CISCO, NOKIA, ...), *производителями инструментов, систем управления базами данных* ит.п. (Borland, IBM, Microsoft, Sun, Oracle, ...). Понимание этого факта позволяет определить достаточный и полный набор стандартов, применяемых в проектной команде или организации в целом.

*Использование внутренних стандартов.* Определенные стандарты, соглашения и процедуры могут быть также созданы внутри организации или даже проектной команды. Эти стандарты поддерживают координацию между определенными видами деятельности, группами операций, минимизируют сложность (в том числе при взаимодействии членов проектной группы и за ее пределами), могут быть связаны с вопросами ожидания и обработки изменений, рисков и вопросами конструирования для проверки и дальнейшего тестирования. В сочетании с внешними стандартами, *внутренние стандарты призваны*

определить общие правила игры для всех членов проектной команды, договорившись о терминах, процедурах и других значимых соглашениях, вне зависимости от степени формализации процессов конструирования, в частности, и процессов жизненного цикла, в общем случае.

*Управление конструированием* - это управление процессом конструирования ПО, которое базируется на моделях конструирования, планировании и внесении изменений.

Модели конструирования включают набор операций, последовательность действий и результатов. Виды моделей определяются стандартом ЖЦ, методологиями и практиками.

Создано множество моделей разработки программного обеспечения, ряд из них в большей степени сфокусирован на конструировании ПО, как таковом. Некоторые модели являются более линейными с точки зрения конструирования ПО. К ним относятся, например, водопадная (waterfall) и поэтапная (staged-delivery) модели ЖЦ. Другие модели более итеративны, к ним относятся: эволюционное прототипирование, экстремальное программирование (XP - eXtreme Programming) и Scrum. Эти подходы сходятся к рассмотрению конструирования как деятельности, которая ведется одновременно с другими видами работ по созданию программного обеспечения и пересекаясь с ними, включая определение требований, проектирование и планирование. Эти подходы смешивают проектирование, кодирование и тестирование, часто рассматривая их комбинацию как конструирование. Конструирование с помощью моделирования осуществляется в рациональном унифицированном процессе - RUP (Rational Unified Process) [18].

Более подробно данные модели мы рассмотрим в следующих главах.

Планирование состоит в определении порядка операций и уровня выполнения заданных условий в процессе конструкторской деятельности. Определяется модель ЖЦ, включающая задачи и действия по созданию компонентов, а также по их проверке, включая достижение показателей качества на этапах ЖЦ. Исполнители распределяются по процессам ЖЦ и выполняют соответствующие задачи по реализации промежуточного продукта. Используемый подход к конструированию влияет на возможность уменьшения (в идеале - минимизации) сложности, готовности к изменениям и конструированию с возможностью проверки.

Затем проводится измерение разных аспектов конструирования, например, количественная оценка объема кода, оценка степени повторного использования reuse, вычисление вероятности появления дефектов и оценка количественных показателей

качества ПО. Эти измерения, или как их еще принято называть, – результаты *аудита кода* или метрики кода, несут большую пользу как для оценки рисков и качества (приводящих к соответствующим операциям по снижению рисков и повышению качества), а также, для управления конструированием и программными проектами, в целом.

Код является одним из наиболее четко детерминированных активов проекта (постепенно такими становятся и модели, строящиеся на основе структур метаданных, и тесно связанные с кодом - например, UML). Код является и самим носителем требуемой функциональности. Соответственно, применение измерений в отношении кода становится тем инструментом, который влияет и на управление и на сам код.

Внесение изменений связано с ошибками, выявленными путем разного рода проверок и тестирований, оно проводится с целью сохранения функциональной целостности системы. В случае обнаружения ошибок на этапе сопровождения принимается решение об изменении кода путем исправления обнаруженных ошибок или внесения изменений в требования к ПО.

*Практические соображения.* Приближаясь к ограничениям реального мира, конструирование (в большей степени, чем любая другая область знаний) ведется на основе практических соображений и техник.

Проектирование в конструировании (Construction Design). Некоторые проекты предполагают большой объем работ по проектированию именно на стадии конструирования; другие проекты явно выделяют проектную деятельность в форме фазы дизайна. Вне зависимости от четкости выделения деятельности по проектированию, как таковой, практически всегда на стадии конструирования приходится заниматься и вопросами детального дизайна системы. Такие проектные работы имеют стремление к следованию устойчивым ограничениям, навязываемым конкретными проблемами, решение которых должно быть обеспечено использованием конструируемой программной системы.

Языки конструирования (Construction Languages). Языки конструирования включают все формы коммуникаций, с помощью которых человек может задать решение проблемы, выполняемое на компьютере.

Простейший тип языков конструирования - *конфигурационный язык* (configuration language), позволяющий задавать параметры выполнения программной системы.

*Инструментальный язык* (toolkit language) - язык конструирования из повторно-используемых элементов; обычно строится как сценарный язык (script), выполняемый в соответствующей среде.

*Язык программирования* (programming language) - наиболее гибкий тип языков конструирования. Содержит минимальный объем информации о конкретных областях приложения и процессе разработки, требуя больше всего (по сравнению с другими типами языков конструирования) усилий на изучение и наработку опыта для эффективного применения при решении конкретных задач.

Существует три основных вида нотаций, используемых при определении языков программирования (см. 3.1.2. Стили конструирования):

- лингвистическая;
- формальная;
- визуальная.

*Лингвистические нотации* характеризуются, в частности, использованием строк текста, содержащих специализированные «слова», представляющие сложные программные конструкции и комбинируемые в шаблоны, напоминающие предложения, построенные в соответствии с определенным синтаксисом. В случае корректного использования таких нотаций каждая получаемая строка обладает строгой смысловой нагрузкой (семантикой), обеспечивающей интуитивное понимание того, что будет происходить когда будет выполняться программное обеспечение, построенное с использованием такого языка конструирования.

*Формальные нотации* являются менее интуитивными, чем лингвистические, и часто базируются на точных формальных (математических) определениях. Формальные нотации конструкций и формальные методы являются ядром практически всех форм системного программирования, точнее - поведения систем во времени. Такие нотации обеспечивают наибольшую готовность получаемого кода к тестированию, что намного важнее, чем просто возможность отображения на обычный человеческий язык. Формальные конструкции также используют точный метод определения комбинаций применяемых символов, что позволяет избежать неоднозначностей, присущих конструкциям естественных языков.

*Визуальные нотации* наименее связаны с текстово-ориентированными подходами, предполагая непосредственную интерпретацию визуальных конструкций в процессе исполнения описываемой логики. При этом логика в визуальных нотациях задается расположением соответствующих визуальных сущностей, ответственных за те или иные операции и структуры данных.

Использование визуальных конструкций ограничено сложностью визуального представления сложных выражений и утверждений только за счет перемещения

визуальных сущностей на диаграмме (визуальном представлении). Однако, визуальная нотация может играть роль достаточно мощного инструмента, когда применяется в тех задачах программирования, где необходимо построение пользовательского интерфейса для программ, чья логика, детализированное поведение определено ранее.

Кодирование (Coding). Практика конструирования ПО показывает активное применение следующих соображений и *техник*:

- техники создания легко понимаемого исходного кода на основе использования соглашений об именовании, форматирования и структурирования кода;
- использование классов, перечисляемых типов, переменных, именованных констант и других выразительных сущностей;
- организация исходного текста (в выражения, шаблоны, классы, пакеты/модули и другие структуры);
- использование структур управления;
- обработка ошибочных условий и исключительных ситуаций;
- предотвращение возможных «брешей» в безопасности (например, переполнение буфера или выход за пределы индекса в массиве);
- использование ресурсов на основе применения механизмов исключения (из рассмотрения) и порядка доступа к параллельно используемым ресурсам (например, на основе блокировки данных, использования потоков и их синхронизации и т.п.);
- документирование кода;
- тонкая «настройка» кода.

Тестирование в конструировании (Construction Testing). При конструировании используются две формы тестирования, проводимого инженерами, непосредственно создающими исходный код:

- *модульное тестирование* (unit testing);
- *интеграционное тестирование* (integration testing).

Главная цель тестирования в конструировании уменьшить временной разрыв между моментом проявления ошибок, имеющих в коде, и моментом их обнаружения. Во многих случаях, тестирование в конструировании производится после того, как код написан. В ряде случаев, тесты (например, при использовании XP) пишутся до того, как создается код.

Повторное использование (Reuse). Во введении в стандарт IEEE Std. 1517-99 «IEEE Standard for Information Technology - Software Lifecycle Process - Reuse Processes» дается следующее понимание повторному использованию в программном обеспечении:

«Реализация повторного использования программного обеспечения подразумевает и влечет за собой нечто большее, чем просто создание и использование библиотек активов. Оно требует формализации практики повторного использования на основе интеграции процессов и деятельности по повторному использованию в сам ЖЦ ПО».

Задачи, связанные с повторным использованием в процессе конструирования и тестирования, включают:

- выбор единиц (units), баз данных тестовых процедур, данных <полученных в результате> тестов и самих тестов, подлежащих повторному использованию;
- оценку потенциала повторного использования кода и тестов;
- отслеживание информации и создание отчетности по повторному использованию в новом коде, тестовых процедурах и данных, полученных в результате тестов.

Качество конструирования (Construction Quality). Существует ряд техник, предназначенных для обеспечения качества кода, выполняемых по мере его конструирования. Основные техники обеспечения качества, используемые в процессе конструирования, включают:

- модульное (unit) и интеграционное (integration) тестирование;
- разработка с первичностью тестов (test-first development - тесты пишутся до конструирования кода);
- пошаговое кодирование (деятельность по конструированию кода разбивается на мелкие шаги, только после тестирования результатов которых производится переход к следующему шагу кодирования; известен также как *итеративное кодирование с тестированием*);
- использование процедур утверждений (assertion);
- отладка (в привычном понимании - debugging);
- технические обзоры и оценки (review);
- статический анализ.

Выбор и использование конкретных техник часто диктуется стандартами (внутренними и внешними), используемыми проектной командой, а также зависят от опыта и подготовленности специалистов, занимающихся конструированием кода.

Деятельность по обеспечению качества в конструировании отличается от других операций по обеспечению качества. Основное отличие заключается в фокусе на программном (исходном) коде и других артефактах (активах), тесно связанных с кодом, в частности, детальных моделях.



Интеграция (Integration). Одна из ключевых деятельности, осуществляемых в процессе конструирования, - интеграция отдельно сконструированных операций (процедур), классов, компонентов и подсистем (модулей). В дополнение к этому, некоторые программные системы нуждаются в специальной интеграции с другим программным и аппаратным обеспечением.

Кроме упомянутых аспектов интеграции, к обсуждаемым интеграционным вопросам конструирования относятся:

- планирование последовательности, в которой интегрируются компоненты;
- обеспечение поддержки создания промежуточных версий программного обеспечения;
- задание «глубины» тестирования (в частности, на основе критериев «приемлемого» качества) и других работ по обеспечению качества интегрируемых в дальнейшем компонент;
- наконец, определение этапных точек проекта, когда будут тестироваться промежуточные версии конструируемой программной системы.

### 2.1.5 Тестирование ПО (Software Testing)

Не секрет, что легче предотвратить проблему, чем бороться с ее последствиями. Тестирование, наравне с управлением рисками, является тем инструментом, который позволяет действовать именно в таком ключе. Причем действовать достаточно эффективно. С другой стороны, необходимо осознавать, что даже если приемочные тесты показали положительные результаты, это совсем не означает, что полученный продукт не содержит ошибок. Однако, адекватное внимание вопросам тестирования качественно снижает риск возникновения ошибок на этапе эксплуатации, обеспечивая более высокую удовлетворенность пользователей, что и является, по существу, целью любого проекта.

**Тестирование ПО** – это процесс анализа или эксплуатации программного обеспечения с целью выявления дефектов [Быстрое тестирование].

Несмотря на всю простоту этого определения, в нем содержатся пункты, которые требуют дальнейших пояснений. Слово *процесс* (process) используется для того, чтобы подчеркнуть, что тестирование – плановая, упорядоченная деятельность. Этот момент очень важен, если мы заинтересованы в быстрой разработке, ибо хорошо продуманный, систематический подход быстрее приводит к обнаружению программных ошибок, чем плохо спланированное тестирование, к тому же проводимое в спешке.

Согласно этому определению, тестирование предусматривает «анализ» или «эксплуатацию» программного продукта (ПП). Тестовая деятельность, связанная с анализом результатов разработки программного обеспечения (ПО), называется *статическим тестированием* (static testing). Статическое тестирование предусматривает проверку программных кодов, сквозной контроль и проверку программы без запуска на машине, т.е. проверку за столом (desk checks). В отличие от этого, тестовая деятельность, предусматривающая эксплуатацию программного продукта, носит название *динамического тестирования* (dynamic testing). Статическое и динамическое тестирование дополняют друг друга, и каждый из этих типов тестирования реализует собственный подход к выявлению ошибок.

Исходя из этого, можно дать другое, более развернутое определение тестирования:

**Тестирование ПО** - это процесс проверки готовой программы *в статике* (просмотры, инспекции, отладки исходного кода) и *в динамике* путем ее прогона на *конечном (ограниченном)* наборе тестовых данных (set of test cases), *выбранных* соответствующим образом из обычно выполняемых действий прикладной области, и проверяющих разные пути выполнения программы, и сравнении полученных результатов с заранее запланированными в соответствии *ожидаемому* поведению системы.

Статические техники рассматриваются в области знаний «Инженерия качества ПС» («Software Quality»). Данная область знаний – «Тестирование» - касается динамических техник. В данном определении тестирования выделены слова, определяющие основные вопросы, которым адресуется данная область знаний:

- *Динамичность* (dynamic): этот термин подразумевает выполнение тестируемой программы с заданными *входными данными*. При этом величины, задаваемые на вход тестируемому ПО, не всегда достаточны для определения теста. Сложность и недетерминированность ПС приводит к тому, что система может по разному реагировать на одни и те же входные параметры, в зависимости от состояния системы. В данной области знаний термин «вход» (input) будет использоваться в рамках соглашения о том, что вход может также специфицировать состояние системы, в тех случаях, когда это необходимо.
- *Конечность* (ограниченность, finite): даже для простых программ теоретически возможно столь большое количество тестовых сценариев, что исчерпывающее тестирование может занять многие месяцы и даже годы. Именно поэтому, с практической точки зрения, всестороннее тестирование считается бесконечным. Тестирование всегда предполагает компромисс между ограниченными ресурсами и

заданными сроками, с одной стороны, и практически неограниченными требованиями по тестированию, с другой. То есть мы снова говорим об определении характеристик «приемлемого» качества, на основе которых планируем необходимые объем тестирования.

- *Выбор* (selection): многие предлагаемые техники тестирования отличаются друг от друга в том, как выбираются сценарии тестирования. Инженеры по программному обеспечению должны обладать представлением о том, что различные критерии выбора тестов могут давать разные результаты, с точки зрения эффективности тестирования. Определение подходящего набора тестов для заданных условий является очень сложной проблемой. Обычно, для выбора соответствующих тестов совместно применяют техники анализа рисков, анализ требований и соответствующую экспертизу в области тестирования и заданной прикладной области.
- *Ожидаемое поведение* (expected behavior): Хотя это не всегда легко, все же необходимо решить, какое наблюдаемое поведение программы будет приемлемо, а какое - нет. В противном случае, усилия по тестированию бесполезны. Наблюдаемое поведение может рассматриваться в контексте пользовательских ожиданий (подразумевая «тестирования для проверки» - testing for validation), спецификации («тестирование для аттестации» - testing for verification) или, наконец, в контексте предсказанного поведения на основе неявных требований или обоснованных ожиданий.

Область знаний «Тестирование ПО (Software Testing)» включает следующие разделы (см. рисунок 2.6):

- основные концепции и определение тестирования (Testing Basic Concepts and definitions),
- уровни тестирования (Test Levels);
- техники тестирования (Test Techniques);
- метрики тестирования (Test Related Measures);
- управление процессом тестирования (Managing the Test Process).

Данная область знаний SWEBOOK представляет разработчику *методы проверки правильности ПО: верификация, валидация, тестирование*. Определяются типы, уровни и техники тестирования ПО, методы планирования процесса тестирования и разработки тестовых наборов данных для прогонки ПО в режиме испытания конкретного модуля или системы в целом, с последующим измерением результатов тестирования.

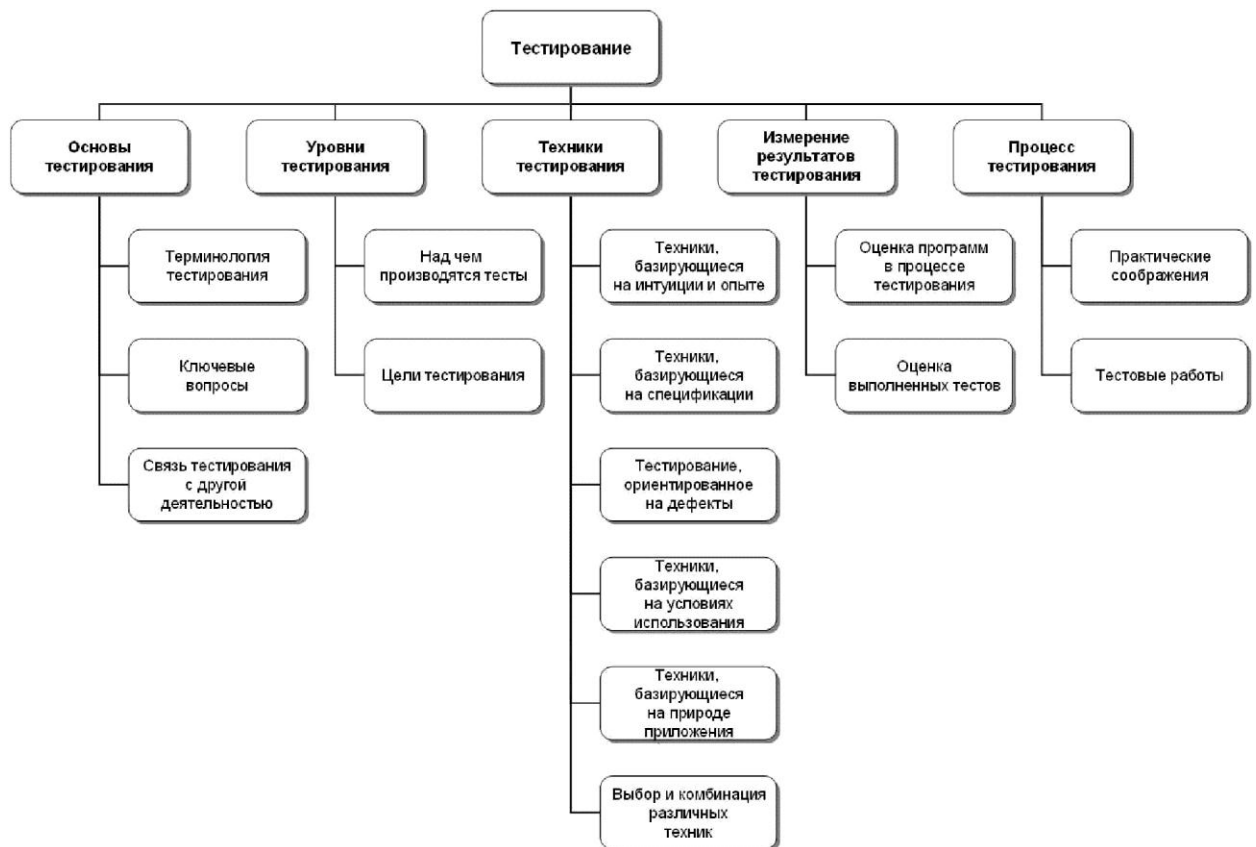


Рисунок 2.6 - Область знаний «Тестирование программного обеспечения» [12]

#### 2.1.4.1. Основные концепции и определение тестирования

Основная концепция тестирования описывает базовые термины, ключевые проблемы и их связь с другими областями знаний. Первое определение, которое требует дополнительных пояснений, – это понятие *дефекта* (bug), т.е. программная ошибка – это ни что иное, как изъян в разработке ПП, который вызывает несоответствие ожидаемых результатов его выполнения и фактически полученных результатов. Дефект может возникнуть:

- на стадии формулирования требований,
- на стадии проектирования,
- на стадии кодирования,
- его причина может крыться в некорректной конфигурации или данных.

Дефектом может быть также что-то другое, что не соответствует ожиданиям заказчика и что может быть, а может и не быть определено в спецификации ПП.

В литературе, посвященной программной инженерии, встречается множество терминов, описывающих нарушение функционирования программных систем – *недостатки* (faults), *дефекты* (defects), *сбои* (failures), *ошибки* (errors) и др. Соответствующая терминология, описанная в IEEE Std. 610-90, также обсуждается в области знаний SWEBOOK «Качество программного обеспечения» (Software Quality).

Важно четко разделять *причину нарушения работы* прикладных систем, обычно описываемую терминами недостаток или дефект, и наблюдаемый *нежелательный эффект*, вызываемый этими причинами – сбой. Термин ошибка, в зависимости от контекста, может описывать и как причину сбоя, и сам сбой. Тестирование позволяет обнаружить дефекты, приводящие к сбоям.

Необходимо понимать, что причина сбоя не всегда может быть однозначно определена. Не существует теоретических критериев, позволяющих гарантированно определить какой именно дефект приводит к наблюдаемому сбою.

*К ключевым вопросам данной области относятся:*

- 1) Критерии отбора тестов/критерии адекватности тестов, правила прекращения тестирования;
- 2) Эффективность тестирования/Цели тестирования;
- 3) Тестирование для идентификации дефектов (данный случай тестирования подразумевает успешность процедуры тестирования, если дефект найден.);
- 4) Проблема оракула («Оракул», в данном контексте, любой агент (человек или программа), оценивающий поведение программы, формулируя вердикт - тест пройден («pass») или нет («fail»));
- 5) Теоретические и практические ограничения тестирования (по словам Дейкстры (Dijkstra), «тестирование программы может использоваться для демонстрации наличия дефектов, но никогда не покажет их отсутствие»);
- 6) Проблема неосуществимых путей (эта сложнейшая проблема автоматизированного тестирования связана с тем, что путь, по которому выполняются потоки работ тестируемой ПС, не могут быть заданы входными параметрами);
- 7) Тестируемость (это понятие может подразумевать две различных идеи: первая описывает степень легкости описания критериев покрытия тестами для заданной программной системы; вторая определяет возможность статистического измерения того, что при тестировании проявится сбой программной системы).

#### *Уровни тестирования*

Тестирование обычно производится на протяжении всей разработки и сопровождения на разных уровнях. Уровень тестирования определяет «над чем» производятся тесты: над отдельным модулем, группой модулей или системой, в целом. При этом ни один из уровней тестирования не может считаться приоритетным. Важны все уровни тестирования, вне зависимости от используемых моделей и методологий.

- *модульное тестирование (тестирование отдельных элементов)*, которое заключается в проверке отдельных, изолированных и независимых частей ПО;
- *интеграционное тестирование*, которое ориентировано на проверку связей и способов взаимодействия (интерфейсов) компонентов друг с другом, включая компоненты, расположенные на разных архитектурных платформах распределенной среды. Классические стратегии интеграционного тестирования – «сверху-вниз» и «снизу-вверх» – используются для традиционных, иерархически структурированных систем, современные стратегии в большей степени зависят от архитектуры тестируемой системы и строятся на основе идентификации функциональных «поток» (например, потоков операций и данных);
- *системное тестирование* предназначено для проверки правильности функционирования системы в целом, с обнаружением отказов и дефектов в системе и их устранение. При этом контролируется выполнение сформулированных нефункциональных требований (безопасность, надежность, производительность, точность и др.) в системе, правильность задания и выполнения внешних интерфейсов системы со средой окружения и др.

Тестирование проводится в соответствии с определенными целями (могут быть заданы явно или неявно) и различным уровнем точности. Определение цели точным образом, выражаемым количественно, позволяет обеспечить контроль результатов тестирования. Можно выделить следующие, наиболее распространенные и обоснованные цели (а, соответственно, виды) тестирования, которые применяются на всех уровнях тестирования:

- ✓ *приёмочное тестирование*, которое проверяет поведение системы на предмет удовлетворения требований заказчика;
- ✓ *установочное тестирование*, которое проводится с целью проверки процедуры инсталляции системы в целевом окружении;
- ✓ *функциональное тестирование*, обеспечивающее проверку реализации функций, которые определены в требованиях, а также правильность их выполнения;
- ✓ *регрессионное тестирование*, ориентированное на повторное выборочное тестирование системы или ее компонентов после внесения в них изменений на тех же тестах, что и до модификации;
- ✓ *тестирование эффективности*, проверяющее производительность, пропускную способность, максимальный объем данных и системные ограничения в соответствии со спецификациями требований;

- ✓ *стресс-тестирование (нагрузочное)*, проверяющее поведение системы при максимально допустимой нагрузке или при ее превышении;
- ✓ *альфа- и бета- тестирование*, выполняющие внутреннее тестирование кодов системы и внешнее тестирование интерфейсов. Альфа - это внутреннее тестирование (функций и алгоритмов), а бета - внешнее (взаимосвязей с другими системами и средой);
- ✓ *конфигурационное тестирование*, проверяющее структуру и идентификацию системы на различных наборах данных, а также работу системы на различных конфигурациях аппаратуры и оборудования;
- ✓ *сравнительное тестирование*, позволяющее сравнить две версии системы;
- ✓ *тестирование простоты и удобства использования*, позволяющее проверить, насколько легко конечный пользователь может освоить работу с системой и документацией на нее, а также застрахована ли система от неправильных действий пользователя.

К методам тестирования относятся также *методы проведения испытаний ПО*, проверки реализации требований и обеспечения параметров настройки и размещения компонентов ПО на заданном количестве и типах компьютеров, среды и ОС.

*Техники тестирования* базируются на некоторых теоретических и практических положениях, например, на природе подхода к проектированию (компонентного, объектно-ориентированного, сервисного и т.п.), а также на следующих данных:

- ✓ информация о структуре ПО или системы в документации («белый ящик»);
- ✓ подбор тестовых наборов данных для проверки правильности работы компонентов и системы в целом без знания их структуры («черный ящик»);
- ✓ анализ граничных значений, таблиц принятия решений, потоков данных, статистики отказов и др.;
- ✓ блок-схемы построения программ и составления наборов тестов для покрытия системы этими тестами;
- ✓ обнаруженные и зафиксированные в таблицах системы дефекты, пред- и постусловия выполнения, структурные характеристики системы (количество модулей, объем данных и др.).

#### *Метрики тестирования*

Для измерения результатов тестирования ПО, а также при проведении анализа качества используются метрики. Измерение как часть планирования и разработки тестов

базируется на размере программ, их структуре и количестве обнаруженных ошибок и дефектов. Метрики тестирования обеспечивают измерение процесса планирования, проектирования и тестирования; а также результатов тестирования на основе таксономии отказов и дефектов, покрытия границ тестирования, проверки потоков данных и др. Процесс тестирования документируется и согласно стандарту IEEE 829-98 включает описание тестовых документов, их связи между собой и с задачами тестирования. Без документации по процессу тестирования невозможно провести сертификацию продукта по модели CMM (Capability Maturity Model - модель зрелости процессов создания ПО). После завершения тестирования рассматриваются вопросы стоимости и оценки рисков, вызванных сбоями или недостаточно надежной работой системы. Стоимость тестирования является одним из ограничений, на основе которого принимается решение о прекращении или его продолжении.

*Управление тестированием:*

- планирование процесса тестирования (составление планов, тестов, наборов данных) и оценивание показателей качества ПО;
- проведение тестирования компонентов повторного использования и паттернов как основных объектов сборки ПО;
- генерация необходимых тестовых сценариев, соответствующих среде выполнения ПО;
- верификация правильности реализации системы и валидация правильности реализации требований к ПО;
- сбор данных об отказах, ошибках и др. непредвиденных ситуациях при выполнении программного продукта;
- подготовка отчетов по результатам тестирования и оценка характеристик системы.

Заметим, что стандарт ISO/IEC 12207 и гармонизированный ГОСТ 12207 не выделяет деятельность по тестированию в качестве самостоятельного процесса, а рассматривает тестирование как неотъемлемую часть всего ЖЦ.

### ***2.1.5 Сопровождение ПО (Software maintenance)***

Фаза сопровождения в жизненном цикле, обычно, начинается сразу после приемки/передачи продукта и действует в течение периода гарантии или, чаще, технической поддержки. Однако, сама деятельность, связанная с сопровождением, начинается намного раньше.



**Сопровождение ПО** - совокупность действий по обеспечению работы ПО, а также по внесению изменений в случае обнаружения ошибок в процессе эксплуатации, по адаптации ПО к новой среде функционирования, а также по повышению производительности или улучшению других характеристик ПО.

В связи с решением проблемы 2000 года сопровождение стало рассматриваться как более важный процесс, который должны осуществлять разработчики. Новая версия системы должна решать те же самые задачи, иметь план переноса информации в другие обновленные БД и учета стоимости сопровождения. Сопровождение (в соответствии со стандартами ISO/IEC 12207 и ISO/IEC 14764) считается модификацией программного продукта в процессе эксплуатации при условии сохранения целостности продукта. *Сопровождение программного обеспечения в SWEBOK определяется как вся совокупность деятельности, необходимой для обеспечения эффективной (с точки зрения затрат) поддержки программных систем.*

Область знаний «Сопровождение ПО (Software maintenance)» состоит из следующих разделов (см. рисунок 2.7):

- основные концепции (Basic Concepts);
- ключевые вопросы сопровождения ПО (key Issue in Software Maintenance);
- процесс сопровождения (Process Maintenance);
- техники сопровождения (Techniques for Maintenance).

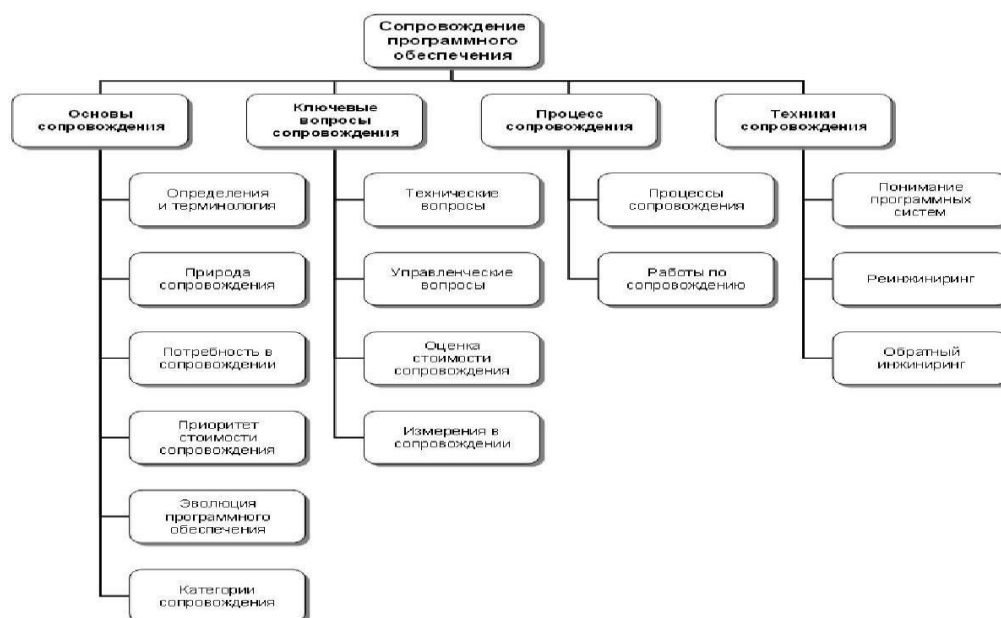


Рисунок 2.7 - Область знаний «Сопровождение программного обеспечения» [12]

Область знаний «Сопровождение программного обеспечения» связана с другими аспектами программной инженерии. По-сути, описание этой области знаний непосредственно пересекается со всеми другими дисциплинами

Сопровождение рассматривается с точки зрения удовлетворения требований к созданному ПО, корректности его выполнения, процессов обучения и оперативного учета процесса сопровождения.

*Основные концепции* описывают базовые определения и терминологию, подходы к эволюции и сопровождению ПО, а также к оценке стоимости сопровождения и др.

К основным концепциям можно отнести ЖЦ ПО (стандарт ISO/IEC 12207) и составление документации. Главное назначение этой области знаний состоит в выполнении готовой программной системы, фиксации возникающих ошибок при выполнении, исследовании причин ошибок, анализа необходимости модификации системы в целях устранения ошибок, оценки стоимости работ по проведению изменений функций и системы в целом. Рассматриваются проблемы, связанные с увеличением сложности продукта при большом количестве изменений и методы ее преодоления.

Процесс сопровождения включает: модели процесса сопровождения и планирование деятельности людей, которые проводят запуск ПО, проверку правильности его выполнения и внесения в него изменений.

В общем случае, работы по сопровождению должны проводиться для решения *следующих задач*:

- устранение сбоев;
- улучшение дизайна;
- реализация расширений функциональных возможностей;
- создание интерфейсов взаимодействия с другими (внешними) системами;
- адаптация (например, портирование) для возможности работы на другой аппаратной платформе (или обновленной платформе), применения новых системных возможностей, функционирования в среде обновленной телекоммуникационной инфраструктуры и т.п.;
- миграции унаследованного (legacy) программного обеспечения;
- вывода программного обеспечения из эксплуатации.

*Деятельность персонала сопровождения* включает четыре ключевых аспекта:

- поддержка контроля (управляемости) программного обеспечения в течение всего цикла эксплуатации;
- поддержка модификаций программных систем;
- совершенствование существующих функций;

- предотвращение падения производительности программной системы до неприемлемого уровня.

Процесс сопровождения согласно стандарту ISO/IEC 14764 определяет четыре *категории сопровождения*:

- Корректирующее сопровождение (corrective maintenance): «реактивная» модификация программного продукта, выполняемая уже после передачи в эксплуатацию для устранения сбоев или нереализованных задач;
- Адаптирующее сопровождение (adaptive maintenance): модификация программного продукта на этапе эксплуатации для обеспечения продолжения его использования с заданной эффективностью (с точки зрения удовлетворения потребностей пользователей) в изменившемся или находящемся в процессе изменения окружении; в первую очередь, подразумевается изменение бизнес-окружения, порождающее новые требования к системе;
- Совершенствующее сопровождение (perfective maintenance): модификация программного продукта на этапе эксплуатации для повышения характеристик производительности и удобства сопровождения;
- Профилактическое сопровождение (preventive maintenance): модификация программного продукта на этапе эксплуатации для идентификации и предотвращения скрытых дефектов до того, когда они приведут к реальным сбоям<sup>6</sup>.

Все категории сопровождения можно представить в виде таблицы.

	Корректирующие работы	Работы по расширению
«Проактивный» подход	Профилактическое сопровождение	Совершенствующее сопровождение
«Реактивный» подход	Корректирующее сопровождение	Адаптирующее сопровождение

<sup>6</sup> Данный вид сопровождения был введен только в данном стандарте, стандарт IEEE 1216 определяет только первые три составляющие. Профилактическое сопровождение (новейшая категория работ по сопровождению) наиболее часто проводится для программных систем, связанных с вопросами безопасности <людей>.

*Ключевые вопросы сопровождения ПО*

Для обеспечения эффективного сопровождения ПС необходимо решать целый комплекс вопросов и проблем, связанных с соответствующими работами. Необходимо понимать, что процесс сопровождения предъявляет уникальные технические и управленческие требования к персоналу, занимающемуся сопровождением и, в первую очередь, специалистам-инженерам. Попытка найти дефект в продукте, содержащем 500 тысяч строк кода, написанных другими инженерами - яркий пример сложностей, с которыми приходится сталкиваться инженерам по сопровождению. Другой пример, уже организационный, постоянная борьба за ресурсы с разработчиками.

Основными из этих вопросов являются технические, управленческие, измерительные и стоимостные.

Сущность технических вопросов состоит в возможности внесения изменений в ПО (исследования показывают, что от 40 до 60 процентов усилий по сопровождению тратится на анализ и понимание сопровождаемого программного обеспечения), правильной организации процесса тестирования (для сопровождения системы особо значимым является выборочное регрессионное тестирование). Инженерам необходимо проводить полный анализ возможных последствий и влияний изменений, вносимых в существующую систему (в частности, с точки зрения эффективности затрат), а также специфицировать, оценивать и контролировать характеристики, влияющие на возможность сопровождения. Если такие работы проводятся регулярно, это облегчает дальнейшее сопровождение, повышая его сопровождаемость (в частности, как характеристику качества).

Сущность управленческих вопросов состоит в контроле за ПО в процессе модификации, совершенствовании функций и недопущении снижения производительности системы. Сопровождение системы преследует цели максимального продления срока эксплуатации программного обеспечения, поэтому необходимо так организовать все работы, чтобы уменьшить срок возврата инвестиций. Часто работа по сопровождению не выглядит привлекательной, инженеры по поддержке воспринимаются как специалисты «второго класса», поэтому вопросы кадрового обеспечения выходят на первый план. Организационные вопросы подразумевают, какая организация будет отвечать и/или какие функции необходимо выполнять для обеспечения деятельности по сопровождению (выбор сопровождающей организации осуществляется исходя из тех соображений, которые выглядят обоснованными для обеспечения адекватной поддержки системы и возможности ее эволюционирования для удовлетворения меняющихся

потребностей пользователей). Крупные корпорации передают в управление другим организациям целые портфели программных систем, а, иногда, и целиком всю ИТ-инфраструктуру (работы по сопровождению передают «в аутсорсинг<sup>7</sup>» только в тех случаях, если убеждены в стратегическом контроле над сопровождением).

Стоимостные вопросы связаны с оценкой затрат на сопровождение ПО в зависимости от его типа, квалификации персонала, платформы и др. Знание этих факторов часто позволяет уменьшить затраты. Стандарт ISO/IEC 14764 определяет, что «существует два наиболее популярных метода оценки стоимости сопровождения - параметрическая модель и использование опыта». Чаще всего, оба этих подхода комбинируются для повышения точности оценки.

Вопросы измерения связаны с оценкой характеристик системы после ее модификации, а также повторного тестирования и оценки показателей качества. Существуют общие (для всего жизненного цикла) *метрики* и, соответственно, их категории, в частности, определяемые Институтом Программной Инженерии университета Карнеги-Меллон (Software Engineering Institute, Carnegie-Mellon University - SEI CMU): размер, усилия, расписание и качество. Применение этих метрик является хорошей отправной точкой для оценки работ со стороны организации, отвечающей за сопровождение. Стандарты IEEE 1219 (Standard for Software Maintenance) и ISO/IEC 9126-01 (Software Engineering - Product Quality - Part 1: Quality Model, 2001 г.) предлагают специализированные метрики, ориентированные именно на вопросы сопровождения и соответствующие программы:

- Анализируемость (Analyzability): оценка (в первую очередь, дополнительных) усилий или ресурсов, необходимых для диагностики недостатков или причин сбоев, а также для идентификации тех фрагментов программной системы, которые должны быть модифицированы.
- Изменяемость (Changeability): оценка усилий, необходимых для проведения заданных модификаций.
- Стабильность (Stability): оценка случаев непредусмотренного поведения системы, включая ситуации, обнаруженные в процессе тестирования.
- Тестируемость (Testability): оценка усилий персонала сопровождения и пользователей по тестированию модифицированного программного обеспечения.

---

<sup>7</sup> Аутсорсинг - передача работ, в первую очередь, вспомогательных (непрофильных для организации) «на сторону».

*Процессы сопровождения*

Одна из наиболее детально проработанных и распространенных (на уровне стандарта de facto) процессных моделей, изначально созданных с ориентацией на программное обеспечение – CMMI (Capability Maturity Model Integration – интегрированная модель зрелости), разработанная в Институте программной инженерии университета Карнеги-Меллон (SEI CMU). CMMI, в частности, уделяет специальное внимание процессам сопровождения. Существуют и другие, менее распространенные, но тем не менее развивающиеся модели. Процессы сопровождения описывают необходимые работы и детальные входы/выходы этих работ. На рисунках 2.8 и 2.9 приведены работы, которые необходимо выполнить в процессе сопровождения ПО, в соответствии с разными стандартами: работы в общем аналогичные, но сгруппированы по-разному.

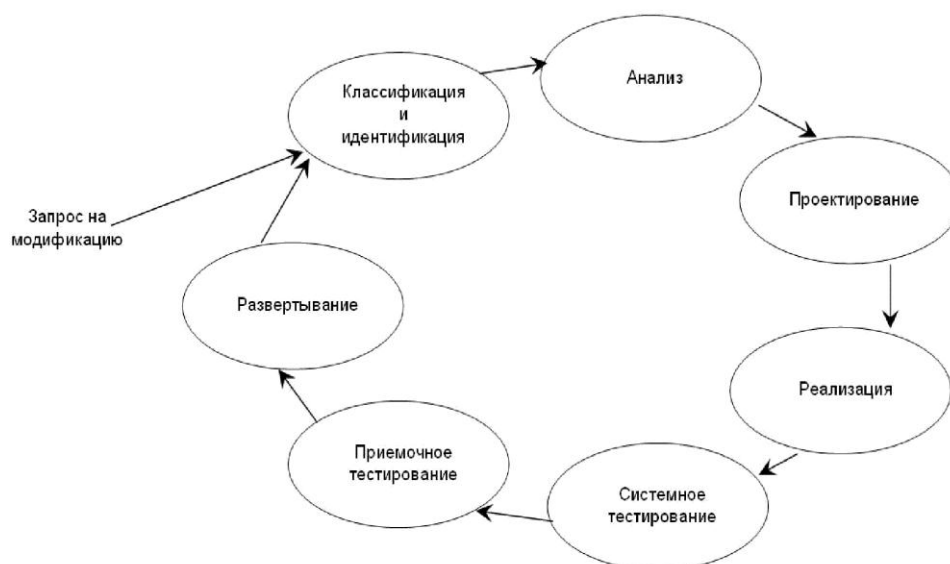


Рисунок 2.8 - Работы в процессе сопровождения по стандарту IEEE 1219



Рисунок 2.9 - Процесс сопровождения по стандарту ISO/IEC 14764

Многие работы по сопровождению похожи на аспекты деятельности по разработке. В обоих случаях необходимо проводить анализ, проектирование, кодирование, тестирование и документирование. Специалисты по сопровождению должны отслеживать требования так же, как и инженеры-разработчики и обновлять документацию по мере разработки и/или выпуска обновленных или новых релизов продукта. В то же время, деятельность по сопровождению обладает и определенными *уникальными чертами*, что приводит к необходимости использования специализированных процессов.

SWEBOK приводит следующие примеры такого рода уникальных характеристик:

- Передача (Transition): контролируемая и координируемая деятельность по передаче программного обеспечения от разработчиков группе, службе или организации, отвечающей за дальнейшую поддержку;
- Принятие/отклонение запросов на модификацию (Modification Request Acceptance/Rejection): запросы на изменения могут как приниматься и передаваться в работу, так и отклоняться по различным обоснованным причинам - объему и/или сложности требуемых изменений, а также необходимых для этого усилий;
- Средства извещения персонала сопровождения и отслеживания статуса запросов на модификацию и отчетов об ошибках (Modification Request and Problem Report Help Desk): функция поддержки конечных пользователей, инициирующая работы по оценке (assessment, подразумевая в том числе оценку необходимости), анализу приоритетности и стоимости модификаций, связанных с поступившим запросом или сообщенной проблемой;
- Анализ влияния (Impact Analysis): анализ возможных последствий изменений, вносимых в существующую систему;
- Поддержка программного обеспечения (Software Support): работы по консультированию пользователей, проводимые в ответ на их информационные запросы (request for information), проверки, содержания данных и специфических (ad hoc) вопросов пользователей и их сообщений о проблемах (ошибках, сбоях, непредусмотренному поведению, непониманию аспектов работы с системой и т.п.);
- Контракты и обязательства: к ним относятся классическое соглашение об уровне предоставляемого сервиса - Service Level Agreement (SLA), а также другие договорные аспекты, на основании которых, группа/служба/организация по сопровождению выполняет соответствующие работы.

*Техники сопровождения (Techniques for Maintenance)*

Известный специалист в области ПО Дж. Леман (1970 г.) предложил рассматривать сопровождение как *эволюционную разработку* программных систем, поскольку сданная в эксплуатацию система не всегда является полностью завершенной, ее надо изменять в течение срока эксплуатации. В результате ПС становится более сложной и плохо управляемой, возникает проблема уменьшения ее сложности. Для решения этих задач в процессе сопровождения программных систем используются общепринятые техники. К ним относятся:

Понимание программных систем (Program Comprehension). Средства работы с кодом являются ключевым инструментом для решения этой задачи. Четкая, однозначная и лаконичная документация обеспечивает адекватное понимание программных систем;

Реинженеринг - это усовершенствование устаревшего ПО путем его реорганизации или реструктуризации, а также перепрограммированием отдельных элементов или настройки параметров на другую платформу или среду выполнения с сохранением удобства его сопровождения.

Обратный (реверсный) инженеринг состоит в восстановлении спецификации (графов вызовов, потоков данных и др.) по полученному коду системы для наблюдения за ней на более высоком уровне. Восстанавливается идентификация программных компонентов и связей между ними для обеспечения перепрограммирования системы к новой форме (уже в процессе реинженеринга). Обратный инжиниринг является *пассивным*, предполагая отсутствие деятельности по изменению или созданию нового программного обеспечения, чаще он всего применяется после того, как в код ПО было внесено много изменений и оно стало неуправляемым. Один из типов обратного инжиниринга - создание новой документации на существующую систему (redocumentation). Другой из распространенных типов - восстановление дизайна системы (design recovery). К вопросам обратного инжиниринга, как и к вопросам реинжиниринга, также относятся работы по рефакторингу. Рефакторинг - это трансформация программного обеспечения, в процессе которой программная система реорганизуется (не переписываясь) с целью улучшения структуры, без изменения поведения. Этот процесс реализуется путем постепенного изменения отдельных операций над текстами, интерфейсами, средой программирования и выполнения ПО, а также настройки или внесения изменений в инструментальные средства поддержки ПО. Сохранение «формы» (платформы, архитектурных и технологических решений) существующей программной системы позволяет рассматривать рефакторинг как один из вариантов обратного инжиниринга.



## Описание организационных областей знаний SWEBOOK

### *Управление конфигурацией ПО (Software Configuration Management -SCM)*

**Управление конфигурацией** состоит в идентификации компонентов системы, определении функциональных и физических характеристик аппаратного и программного обеспечения для контроля за внесением изменений и трассированием конфигурации на протяжении ЖЦ.

Это управление соответствует одному из вспомогательных процессов ЖЦ (ISO/IEC 12207), выполняется техническим и административным руководством проекта; составляются отчеты об изменениях, внесенных в конфигурацию, и степени их реализации, а также проводится проверка соответствия внесенных изменений заданным требованиям.

*Конфигурация системы* - состав функций, программного и технического обеспечения системы, возможные их комбинации в зависимости от наличия оборудования, общесистемных средств, обозначенных в технической документации системы, и требования к продукту.

Конфигурация ПО включает набор функциональных и технических характеристик ПО, заданных в технической документации и реализованных в готовом продукте. Это сочетание разных элементов продукта вместе с заданными процедурами сборки и настройки на среду в соответствии с назначением системы. Примерами элементов конфигурации являются график разработки, проектная документация, исходный и исполняемый код, библиотека компонентов, инструкции по установке и развертыванию системы.

Область знаний «Управление конфигурацией ПО» состоит из следующих разделов:

- управление процессом конфигурации (Management of SMC Process);
- идентификация конфигурации ПО (Software Configuration Identification);
- контроль конфигурации ПО (Software Configuration Control);
- учет статуса (положение конфигурации в ПО или состояние) конфигурации ПО (Software Configuration Status Accounting);
- аудит конфигурации ПО (Software Configuration Auditing);
- управление версиями ПО и доставкой (Software Release Management and Delivery).

*Управление процессом конфигурации.* Это деятельность по контролю эволюции и целостности продукта при идентификации, контроле изменений и обеспечении отчетной информацией, касающейся конфигурации. Включает:

- систематическое отслеживание вносимых изменений в отдельные составные части конфигурации, выполнение аудита изменений и автоматизированного контроля за внесением изменений в конфигурацию системы или ПО;
- поддержку целостности конфигурации, ее аудит и обеспечение внесения изменений в элементы конфигурации;
- ревизию конфигурации в целях проверки наличия разработанных программных или аппаратных средств и согласования версии конфигурации с заданными требованиями;
- трассировка изменений в конфигурацию на этапах сопровождения и эксплуатации ПО.

*Идентификация конфигурации ПО* заключается в документировании функциональных и физических характеристик элементов конфигурации ПО, а также в оформлении технической документация на элементы конфигурации ПО.

*Контроль конфигурации ПО* состоит в проведении работ по координации, утверждению или отбрасыванию реализованных изменений в элементах конфигурации после формальной ее идентификации, а также в анализе входящих компонентов в конфигурацию и соответствия их идентификации.

*Учет статуса или состояния конфигурации ПО* проводится с помощью комплекса мероприятий, позволяющих определить степень изменения конфигурации, полученной от разработчика, а также правильность внесенных изменений в конфигурацию ПО при ее сопровождении. Информация и количественные показатели накапливаются в соответствующей БД и используются при управлении конфигурацией, составлении отчетности, оценивании качества и выполнении других процессов ЖЦ.

*Аудит конфигурации* - это деятельность, которая выполняется для оценки продукта и процессов на соответствие стандартам, инструкциям, планам и процедурам. Аудит определяет степень, в которой элемент конфигурации удовлетворяет заданным функциональным и физическим (аппаратным) характеристикам системы. На основе функционального и физического аудита конфигурации фиксируется базовая линия изготовленного продукта.

*Управление версиями ПО* - это отслеживание имеющейся версии элемента конфигурации; сборка компонентов; создание новых версий системы на основе

существующих путем внесения изменений в конфигурацию; согласование версии продукта с требованиями и проведенными изменениями на этапах ЖЦ; обеспечение оперативного доступа к информации об элементах конфигурации и системы, к которым они относятся. *Управление выпуском* охватывает идентификацию, упаковку и передачу элементов продукта и документации заказчику.

При этом используются следующие основные понятия.

**Базис (baseline)** - формально обозначенный набор элементов ПО, зафиксированный на этапах ЖЦ ПО.

**Библиотека ПО** - контролируемая коллекция объектов ПО и документации, предназначенная для облегчения процесса разработки, использования и сопровождения ПО.

**Сборка ПО** - объединение корректных элементов ПО и конфигурационных данных в единую исполняемую программу.

### *Управление проектами ПО (Software Engineering Management)*

**Управление проектами ПО** - руководство работами команды разработчиков ПО в процессе выполнения плана проекта, определение критериев эффективности работы команды и оценка процессов и продуктов проекта с использованием общих методов управления, планирования и контроля работ.

Как любое управление, менеджмент ПО базируется на планировании, координации, измерении, контроле и учете процесса управления проектом. Координацию людских, финансовых и технических ресурсов при реализации задач программного проекта выполняет менеджер проекта, аналогично тому, как это делается в технических проектах. В его обязанности входит соблюдение запланированных бюджетных и временных характеристик и ограничений, стандартов и сформулированных требований. Общие вопросы управления проектом содержатся в ядре знаний PMBOK [19] в разделе Management Process Activities, а также в стандарте ISO/IEC 12207, где управление проектом рассматривается как дополнительный и организационный процесс ЖЦ.

Область знаний «Управление инженерией ПО (Software Engineering Management)» состоит из следующих разделов:

- организационное управление (Organizational Management);
- управление процессом и проектом (Process/Project Management);
- инженерия измерений ПО (Software Engineering Measurement).

*Организационное управление* - это планирование и составление графика работ, оценка стоимости работ, подбор и

---

управление	персоналом,	контроль	за
------------	-------------	----------	----

выполнением работ согласно принятым стандартам и планам. Главными проблемами организационного управления проектом являются: управление персоналом (обучение, мотивация и др.), коммуникации между сотрудниками (сценарии, встречи, презентации и др.), а также риски (минимизация риска, техники определения риска и др.). Для управления проектом создается определенная структура коллектива, специалисты распределяются по работам и решают задачи проекта под руководством менеджера с учетом заданной стоимости и сроков разработки. Для задач проекта подбираются также необходимые программные, инструментальные и аппаратные средства.

*Управление проектом/процессом* включает: составление плана проекта, построение графиков работ (сетевых или временных диаграмм) с учетом имеющихся ресурсов, распределение персонала по работам проекта, исходя из заданных сроков и стоимости их выполнения. Для эффективного управления проектом проводится анализ финансовой, технической, операционной и социальной политики организации разработчика для выбора правильной стратегии выполнения плана, контроля процесса управления планами и выпуском промежуточных продуктов (проектных решений, диаграмм UML, алгоритмов и др.).

В задачи управления проектом входят также уточнение требований, проверка их на соответствие заданным спецификациям характеристик качества, а также верификация функций отдельных продуктов проекта. Процесс управления проектом базируется на плановых сроках выполнения работ. Результаты планирования отображаются в сетевых диаграммах PERT (Program Evaluation and Review Technique), CPM (Critical Path Method) и др., предназначенных для отображения всех аспектов работ, в частности, времени их выполнения и связей между разными работами в проекте.

На сегодняшний день наиболее распространенным представлением сети для управления разными видами работ является сетевая диаграмма PERT-граф, в вершинах которого располагаются работы, а дуги задают взаимные связи между этими работами. Другой тип сетевой диаграммы, CPM, является событийным. В вершинах такой диаграммы указываются события, а работы задаются линиями между двумя узлами событиями. Ожидаемое время выполнения работы для сетевых диаграмм оценивается с помощью среднего весового значения трех оценок: оптимистической, пессимистической и ожидаемой, т.е. вероятностной. Эти оценки берутся из заданного времени на разработку и заключений экспертов, оценивающих как отдельные работы, так и весь комплекс работ. Есть и другие методы оценок.

После составления плана решается вопрос управления проектом и контроля работ в соответствии с планом, выбранным процессом и сущностью проекта. Корректно составленный план обеспечивает выполнение требований и целей проекта. Контроль осуществляется при внесении изменений в проект, направлен на оценку риска и принимаемых решений по минимизации рисков.

Важной проблемой выполнения проекта является процесс определения рисков и разработки мероприятий по уменьшению их влияния на ход выполнения проекта. Под **риском** понимается вероятность проявления неблагоприятных обстоятельств, которые могут повлиять негативно на управление разработкой (например, увольнение сотрудника и отсутствие замены для продолжения работ и др.). При составлении плана проекта проводится идентификация и анализ риска, планирование непредвиденных ситуаций, касающихся рисков. Предотвращение риска заключается в выполнении действий, которые снимают риск (например, увеличение времени разработки и др.), что уменьшает вероятность появления нового риска при реорганизации проекта, БД или транзакций, а также при выполнении ПО.

*Инженерия измерений ПО* проводится в целях определения отдельных характеристик продуктов и процессов, инженерии планирования и измерения этих характеристик (например, количество строк в продукте, ошибок в спецификациях и т.п.). Предварительно проводятся работы по выбору метрик процессов и продуктов с учетом обстоятельств и зависимостей, влияющих на измерение их характеристик. К аспектам инженерии измерений относятся совершенствование процессов управления проектом; оценки временных затрат и стоимости ПО, их регулирование; определение категорий рисков и отслеживание факторов для регулярного расчета вероятностей их возникновения; проверка заданных в требованиях показателей качества отдельных продуктов и проекта в целом.

Проведение разного рода измерений является важным принципом любой инженерной деятельности. В программном проекте результаты измерений необходимы заказчику и потребителям, чтобы установить, действительно ли проект был реализован правильно. Без измерений в инженерии ПО процесс управления становится неэффективным и превращается в самоцель.

### *Процесс инженерии ПО (Software Engineering Process)*

В некотором смысле это **метауровень**, который связан с определением, реализацией, оценкой, измерением, управлением изменениями и совершенствованием самого процесса.

Однако такой процесс не является единственно правильным способом выполнения задач программной инженерии. На самом деле стандарты о процессах (ГОСТ 12207) говорят о том, что процессов много, например, основной процесс разработки (Development Process), процесс управления конфигурацией (Configuration Management Process) и т.п.

Для оценивания и совершенствования процессов программной инженерии используется **модель зрелости** (Capability Maturity Models - CMM), эту концепцию разработал институт программной инженерии SEI (Software Engineering Institute) США. *Модель описывает существенные атрибуты, которыми должен обладать процесс, находящийся на определенном уровне зрелости, а также указывает практические приемы обеспечения уровня абстракции без ограничений на способы реализации процесса в конкретном проекте.* Разновидностями этой модели являются: CMM - SW (software) для оценки ПО, CMMI - вариант CMM для учета потребностей крупных государственных структур США, Bootstrap - вариант CMM для малых и средних коммерческих компаний, ISO-15504 (Software Process Improvement and Capability - SPICE), ISO 9000-3 как приложение к общей модели качества ISO 9001 и др.

Концепция зрелости процесса ПО основывается на интеграции концепции процесса ПО (software process), широты возможностей процесса ПО (software process capability), результативности процесса ПО (software process performance) и зрелости процесса ПО (software process maturity). Процесс ПО в модели CMM - это множество деятельности (activities), методов (methods), практических приемов и процедур (practices), используемых при разработке ПО и связанных с ним продуктов (например, планов проекта, проектных документов, кода, тестов, руководства пользователя и др.).

**Зрелость процесса** - это степень его четкости определения, управления, измерения, контроля.

CMM предоставляет шаблон для улучшения процесса в виде пяти уровней зрелости, которые создают основу непрерывного улучшения процесса. Эти пять уровней зрелости определяют шкалу упорядочения для измерения зрелости процесса ПО организации и оценивании возможностей этого процесса. **Уровень зрелости** - это определенные

средства для достижения зрелого процесса, а именно, множество целей процесса, которые, в случае их достижения, стабилизируют процесс в проектной организации.

Область знаний «Процесс программной инженерии (Software Engineering Process)» состоит из следующих разделов:

- концепции процесса инженерии ПО (Software Engineering Process Concepts),
- инфраструктура процесса (Process Infrastructure),
- определение процесса (Process Definition),
- оценки процесса (Process Assessments),
- количественный анализ процесса (Qualitative Process Analysis),
- выполнение и изменение процесса (Process Implementation and Change).

*Концепции процесса инженерии ПО* - задачи и действия, которые связаны с управлением, реализацией, оценкой, изменениями и совершенствованием процесса и/или ПО. Цель управления процессом - это создание инфраструктуры процесса, выделение необходимых ресурсов, планирование реализации и изменения процесса в целях внедрения его в практику и, наконец, оценка преимущества от его внедрения при реальной практике проектирования конкретного проекта.

*Инфраструктура проекта* - это его ресурсы (человеческие, технические, информационные и программные), стандарты, службы управления качеством и риском, а также форма организации производства проектов: бригада, экспериментальная фабрика (Experimental Factory), линейка программных продуктов (Framework for Product Line Practice) и др. Она также включает управление и коммуникации в коллективе, инженерные методы организации и производства программного продукта. Главная задача инфраструктуры - совершенствование процесса с учетом опыта разработки ПО.

*Определение процесса* основывается на:

- типах процессов и моделей (водопадная, спиральная, итерационная и др.);
- моделях ЖЦ процессов и средств, стандартах ЖЦ ПО ISO/IEC 12207 и 15504, IEEE std. 1074-91 и 1219-92;
- методах и нотациях задания процессов и автоматизированных средствах их поддержки.

Основной целью процесса является повышение качества получаемого продукта, улучшение различных аспектов программной инженерии, автоматизация процессов и др.

*Оценка процесса* проводится с использованием соответствующих моделей



---

и методов оценки. Например, оценка потенциальной способности специалиста к выполнению соответствующей работы. SWEBOK обращает внимание на процедуру

оценки потенциальной возможности заключения контракта на разработку, организации разработки ПО на основе модели оценки зрелости (СММ) и процессов, согласно которым проводится разработка ПО.

Оценки относятся также и к техническим работам в сфере программной инженерии, к управлению персоналом и качеству ПО. Для этого проводятся экспериментальные исследования среды, наблюдение за выполнением процесса, сбор информации, моделирование, классификация полученных ошибок и дефектов, а также статический анализ недостатков процесса в сравнении с существующими стандартами (например, ГОСТ 12207) и потенциальных аспектов совершенствования процесса.

*Качественный анализ процесса* состоит в идентификации и поиске слабых мест в процессе создания ПО до начала его функционирования. Рассматриваются две техники анализа: обзор данных и сравнение процесса с основными положениями стандарта ISO/IEC 12207; сбор данных о качестве процессов; анализ главных причин отказов в функционировании ПО, откат назад от точки возникновения отклонения до точки нормальной работы ПО для выяснения причин изменения процесса. На качество результатов проекта и процесса влияют применяемые инструменты и опыт специалистов.

*Выполнение и изменение процесса.* Существует ряд фундаментальных аспектов измерений в программной инженерии, лежащих в основе детальных измерений процесса. Оценка совершенствования процессов проводится для установления количественных характеристик процесса и продуктов. После процесса развертывания ПО, выполняются вычисления функций и инспекция результатов выполнения. Результаты процесса могут касаться оценки качества продукта, продуктивности, трудозатрат и др. Если результаты не удовлетворяют пользователя ПО, то проводится анализ и принятие решений о необходимости изменения или усовершенствования процесса, организационной структуры проекта и некоторых инструментов управления измерениями.

### *Методы и инструменты инженерии ПО (Software Engineering Tools and Methods)*

**Методы** обеспечивают проектирование, реализацию и выполнение ПО. Они накладывают некоторые ограничения на инженерию ПО в связи с особенностями применения их нотаций и процедур, а также обеспечивают оценку и проверку процессов и продуктов. **Инструменты** являются программной поддержкой отдельных методов инженерии ПО и обеспечивают автоматизированное выполнение задач процессов ЖЦ.

Область знаний «Методы и инструменты инженерии ПО (Software Engineering Tools and Methods)» состоит из разделов:

- инструменты инженерии ПО (Software Engineering Tools),
- методы инженерии ПО (Software Engineering Methods).

*Методы инженерии ПО* - это эвристические методы (heuristic methods), формальные методы (formal methods) и методы прототипирования (prototyping methods),.

Эвристические методы включают:

- *структурные методы* (structured methods), основанные на функциональной парадигме. При таком подходе системы строятся с функциональной точки зрения, начиная с высокоуровневого понимания поведения системы с постепенным уточнением низко-уровневых деталей. (такой подход, иногда, также называют «проектированием сверху-вниз»);
- *методы, ориентированные на структуры данных* (data-oriented methods), которыми манипулирует ПО. Отправной точкой такого подхода являются структуры данных, которыми манипулирует создаваемое программное обеспечение. Функции в этом случае являются вторичными;
- *объектно-ориентированные методы* (object-oriented methods), которые рассматривают предметную область как коллекцию объектов, а не функций;
- *методы, ориентированные на конкретную область применения* (domain-specific methods). Такие специализированные методы разрабатываются с учетом специфики решаемых задач, например, на системы реального времени, безопасности <жизнедеятельности> (safety) и защищенности <от несанкционированного доступа> (security).

Формальные методы. «Термин формальные методы подразумевает ряд операций, в состав которых входит создание формальной спецификации системы, анализ и доказательство спецификаций, реализация системы на основе преобразования формальной спецификации в программы и верификация программ. Все эти действия зависят от формальной спецификации программного обеспечения. *Формальная спецификация* – это системная спецификация, записанная на языке, словарь, синтаксис и семантика которого определены формально. Необходимость формального определения языка предполагает, что этот язык основывается на математических концепциях. Здесь используется область математики, которая называется дискретной математикой и основывается на алгебре, теории множеств и алгебре логики» [5]<sup>8</sup>.

Различаются следующие категории формальных методов:

<sup>8</sup> Соммервиль, И. Инженерия программного обеспечения. - С.188.

- *языки и нотации спецификации* (specification languages and notations), ориентированные на модель, свойства и поведение. Ярким примером такого рода методов являются формальные методы описания требований, интерес к которым периодически возникает на протяжении всей истории программной инженерии;
- *уточнение спецификации* (refinement specification) путем трансформации в конечный результат, близкий к конечному исполняемому программному продукту. В качестве результата применения таких методов рассматривается конечный - исполнимый программный продукт;
- *методы доказательства/верификации* (verification/proving properties), использующие утверждения (теоремы), пред- и постусловия, которые формально описываются и применяются для установления правильности спецификации программ.

Эти методы применялись в основном в теоретических экспериментах и более 25 лет их практическое применение было ограничено из-за трудоемкости и экономической невыгодности. В 2005 г. проблема верификации приобрела вновь актуальность в связи с разработкой нового международного проекта по верификационному ПО «Целостный автоматизированный набор инструментов для проверки корректности ПС», ставящим следующие перспективные задачи:

- разработка единой теории построения и анализа программ;
- построение многостороннего интегрированного набора инструментов верификации на всех производственных процессах - разработка формальных спецификаций, их доказательство и проверка правильности, генерация программ и тестовых примеров, уточнение, анализ и оценка;
- создание репозитория формальных спецификаций, верифицированных программных объектов разных типов и видов.

Предполагается, что формальные методы верификации будут охватывать все аспекты создания и проверки правильности программ. Это приведет к созданию мощной верификационной производственной основы и значительному сокращению ошибок в ПО (доказательству и верификации будет посвящена тема 7.)

Методы прототипирования (Prototyping Methods) основаны на прототипировании ПО и подразделяются на:

- *стили прототипирования*, включающие в себя создание временно используемых прототипов (throwaway), эволюционное прототипирование (превращение

прототипа в конечный продукт) и разработка исполняемых спецификаций (часто основывается на формальных методах);

- *цели прототипирования*. Примерами таких целей служат требования, архитектурный дизайн или пользовательский интерфейс;
- *техники оценки/исследования* (evaluation) результатов прототипирования. Эти аспекты касаются того, как именно будут использованы результаты создания прототипа (например, будет ли он трансформирован в продукт, создается он для оценки нагрузочных способностей и других аспектов масштабируемости и т.п.).

*Инструменты инженерии ПО* обеспечивают автоматизированную поддержку процессов разработки ПО и включают множество разных инструментов, охватывающих все процессы ЖЦ.

Инструменты работы с требованиями (Software Requirements Tools) - это:

- инструменты разработки (Requirement Development) управления требованиями (Requirement Management) для анализа, сбора, специфицирования и проверки требований. Например, в модели CMMI Staged на 2-м уровне зрелости находится управление требованиями, а на 3-м уровне - разработка требований;
- инструменты трассировки требований (Requirement traceability tools) являются неотъемлемой частью работы с требованиями, их функциональное содержание зависит от сложности проектов и уровня зрелости процессов.

Инструменты проектирования (Software Design Tools) - это инструменты для создания ПО с применением базовых нотаций (SADT/IDEF, UML, Microsoft DSL, Oracle и т.п.).

Инструменты конструирования ПО (Software Construction Tools) - это инструменты для производства, трансляции программ и машинного выполнения. К ним относятся:

- *редакторы* (program editors) для создания и модификации программ, и редакторы «общего назначения» (UNIX и UNIX-подобные среды);
- *компиляторы и генераторы кода* (compilers and code generators) как самостоятельные средства объединения в интегрированной среде программных компонентов для получения выходного продукта с использованием препроцессоров, сборщиков, загрузчиков и др.;
- *интерпретаторы* (interpreters) обеспечивают исполнение программ путем эмуляции, предоставляя для исполнения программ контролируемое и наблюдаемое окружение. Наметились тенденотладчики (debuggers) для проверки правильности описания исходных программ и устранения ошибок;

- *интегрированные среды разработки* (IDE - integrated developers environment), библиотеки компонент (libraries components), без которых не может проводиться процесс разработки ПС, программные платформы (Java, J2EE и Microsoft .NET) и платформа распределенных вычислений (CORBA и WebServices).

Инструменты тестирования (Software Testing Tools) это:

- *генераторы тестов* (test generators), помогающие в разработке сценариев тестирования;
- *средства выполнения тестов* (test execution frameworks) обеспечивают выполнение тестовых сценариев и отслеживают поведение объектов тестирования;
- *инструменты оценки тестов* (test evaluation tools) поддерживают оценку результатов выполнения тестов и степени соответствия поведения тестируемого объекта ожидаемому поведению;
- *средства управления тестами* (test management tools) обеспечивают инженерию процесса тестирования ПО;
- *инструменты анализа производительности* (performance analysis tools), количественной ее оценки и оценки поведения программ в процессе выполнения.

Инструменты сопровождения (Software Maintenance Tools) включают в себя:

- *инструменты облегчения понимания* (comprehension tools) программ, например, различные средства визуализации;
- *инструменты реинжиниринга* (reengineering tools) поддерживают деятельность по реинжинирингу и обратной инженерии (reverse engineering) для восстановления (артефактов, спецификация, архитектуры) стареющего ПО и генерации нового продукта.

Инструменты конфигурационного управления (Software Configuration Management Tools) - это:

- *инструменты отслеживания* (tracking) дефектов;
- *инструменты управления версиями*;
- *инструменты управления сборкой*, выпуском версии (конфигурации) продукта его инсталляции.

Инструменты управления инженерной деятельностью (Software Engineering Management Tools) состоят из:

- *инструментов планирования и отслеживания проектов*, количественной оценки усилий и стоимости работ проекта (Microsoft Project 2003);



- *инструментов управления рисками* используются для идентификации, мониторинга рисков и оценки нанесенного вреда;
- *инструментов количественной оценки свойств ПО* путем ведения измерений и расчета окончательного значения надежности и качества.

Инструменты поддержки процессов (Software Engineering Process Tools) разделены на:

- *инструменты моделирования* и описания моделей ПО (например, UML и его инструменты);
- *инструменты управления программными проектами* (Microsoft Project 2003);
- *инструменты управления конфигурацией* для поддержки версий и всех артефактов проекта.

Инструменты обеспечения качества (Software Quality Tools) делятся на две категории:

- *инструменты инспектирования* для поддержки просмотра (review) и аудита;
- *инструменты статического анализа* программных артефактов, данных, потоков работ и проверки свойств или артефактов на соответствие заданным характеристикам.

Дополнительные аспекты инструментального обеспечения (Miscellaneous Tool Issues) соответствуют таким аспектам:

- *техники интеграции инструментов* (платформ, представлений, процессов, данных и управления) для естественного их сочетания в интегрированной среде
- *метаинструменты* для генерации других инструментов;
- *оценка инструментов при их эволюции*.

### ***Качество ПО (Software Quality)***

**Качество ПО** - набор свойств продукта (сервис или службы), которые характеризуют его способность удовлетворить установленные или предполагаемые потребности заказчика.

В общем случае под **качеством (quality) программ** понимается то, насколько они соответствуют установленным для них требованиям – спецификациям и сколько высоко установлена планка этих требований, это совокупность черт и характеристик ПС, которые влияют на ее способность удовлетворять заданные потребности пользователей [20, 21].

Понятие качества имеет разные интерпретации в зависимости от конкретной системы и требований к программному продукту. Кроме того, в разных источниках таксономия (классификация) характеристик в модели качества отличается.

Каждая модель имеет различное число уровней, и все известные модели качества имеют полное или частичное совпадение набора характеристик качества.

#### *Качество по МакКолу*

Модель качества МакКолла на самом высоком уровне имеет три характеристики: функциональность, модифицируемость и переносимость, а на более нижних уровнях модели, 11 подхарактеристик качества и 18 критериев (атрибутов) качества.

Первой широко известной моделью качества ПО стала предложенная в 1977 МакКолом и др. [22] модель. В ней характеристики качества разделены на три группы:

1. *Факторы* (factors), описывающие ПО с позиций пользователя и задаваемые требованиями.
2. *Критерии* (criteria), описывающие ПО с позиций разработчика и задаваемые как цели.
3. *Метрики* (metrics), используемые для количественного описания и измерения качества.

Факторы качества, которых было выделено 11, группируются в три группы по различным способам работы людей с ПО. Полученная структура изображается в виде треугольника МакКола (рисунок 2.10).



Рисунок 2.10 – Треугольник МакКола

Критерии качества — это числовые уровни факторов, поставленные в качестве целей при разработке. Объективно оценить или измерить факторы качества непосредственно довольно трудно. Поэтому, МакКол ввел метрики качества, которые с его точки зрения легче измерять и оценивать. Оценки в его шкале принимают значения от 0 до 10.

МакКол рассматривал следующие метрики:

1. Удобство проверки на соответствие стандартам (auditability).
2. Точность управления и вычислений (accuracy).
3. Степень стандартности интерфейсов (communication commonality).
4. Функциональная полнота (completeness).
5. Однородность используемых правил проектирования и документации (consistency).
6. Степень стандартности форматов данных (data commonality).
7. Устойчивость к ошибкам (error tolerance).
8. Эффективность работы (execution efficiency).
9. Расширяемость (expandability).
10. Широта области потенциального использования (generality).
11. Независимость от аппаратной платформы (hardware independence).
12. Полнота протоколирования ошибок и других событий (instrumentation).
13. Модульность (modularity).
14. Удобство работы (operability).
15. Защищенность (security).
16. Самодокументированность (selfdocumentation).
17. Простота работы (simplicity).
18. Независимость от программной платформы (software system independence).
19. Возможность соотнесения проекта с требованиями (traceability).
20. Удобство обучения (training).

Каждая метрика влияет на оценку нескольких факторов качества. Числовое выражение фактора представляет собой линейную комбинацию значений влияющих на него метрик. Коэффициенты этого выражения определяются по-разному для разных организаций, команд разработки, видов ПО, используемых процессов и т.п.

#### *Качество по Боему*

В 1978 Боем [23, 24] предложил свою модель, по существу представляющую собой расширение модели МакКола, в которой атрибуты качества подразделяются по способу использования ПО (primary use). ИМ определено 19 промежуточных атрибутов (intermediate construct), включающих все 11 факторов качества по МакКолу. Промежуточные атрибуты разделяются на примитивные (primitive construct), которые, в свою очередь, могут быть оценены на основе метрик.

В дополнение к факторам МакКола атрибуты качества по Боему включают следующие:

1. ясность (clarity),
2. удобство внесения изменений (modifiability),
3. документированность (documentation),
4. способность к восстановлению функций (resilience),
5. понятность (understandability),
6. адекватность (validity),
7. функциональность (functionality),
8. универсальность (generality),
9. экономическая эффективность (economy).

#### *Качество по стандарту ISO 9126-01*

Стандарт ISO 9126-01 рассматривает внешние и внутренние характеристики качества. Внешние характеристики качества отображают требования к функционирующему программному продукту. Для количественного задания критериев качества, по которым будет осуществляться проверка и подтверждение соответствия ПО заданным требованиям, определяются соответствующие внешние измеряемые свойства (внешние атрибуты) ПО и метрики. Ими могут быть конкретные значения (например, время выполнения отдельных компонентов), диапазоны изменения значений для некоторых внешних атрибутов и модели их оценивания. Метрики, применение которых возможно только для ПО, функционирующего на компьютере, используются на стадии тестирования или функционирования. Они называются внешними метриками и представляют собой модели оценивания атрибутов.

Внутренние характеристики качества и внутренние атрибуты ПО используются при составлении плана достижения необходимых внешних характеристик качества для конечного программного продукта. Для определения внутренних характеристик качества специфицируются внутренние метрики, используемые при проверке соответствия промежуточных продуктов спецификациям внутренних требований к качеству, которые формулируются на этапах, предшествующих тестированию (определение требований, проектирование, кодирование).

Внешние и внутренние характеристики качества отображают свойства самого ПО (работающего или не работающего), а также взгляд заказчика и разработчика на это ПО. Непосредственного конечного пользователя ПО интересует **эксплуатационное качество ПО** - совокупный эффект от достижения характеристик качества, который измеряется в сроках использования результата, а не свойств самого ПО. Это понятие шире, чем любая отдельная характеристика (например, удобство использования или надежность).

Окончательная оценка качества проводится в соответствии со стандартом ISO 15504-98 [25]. Качество может повышаться за счет постоянного улучшения используемого продукта в связи с процессами обнаружения, устранения и предотвращения сбоев/дефектов в ПО.

Область знаний «Качество ПО (Software Quality)» состоит из следующих разделов:

- концепция качества ПО (Software Quality Concepts);
- определение и планирование качества (Definition & Planning for Quality);
- техники и активности, обеспечивающие гарантию качества, валидацию и верификацию (Activities and Techniques for Software Quality Assurance, Validation - V & Verification - V);
- измерение при анализе качества ПО (Measurement in Software Quality Analysis).

*Концепция качества ПО* - это внешние и внутренние характеристики качества, их метрики, а также модели качества, определенные на множестве этих характеристик, которые представлены в стандартах качества [26] - это шесть характеристик и каждая из них имеет несколько атрибутов. К характеристикам качества относятся:

- **функциональность** (functionality) – степень соответствия системы требованиям заказчика, ПС должна выполнять те функции и в таком виде, как они были определены в функциональной спецификации;
- **надежность** (realibility) - это способность ПС выполнять возложенные на нее функции при поступлении требований на их выполнение в течение заданного интервала времени. Надежность относится к *динамическим требованиям*, предъявляемым к системе, и включает в себя такие элементы как:
  - ✓ *Отказоустойчивость* – возможность восстановления программы и данных в случае сбоев в работе;
  - ✓ *Безопасность* – сбои в работе программы не должны приводить к опасным последствиям (авариям);
  - ✓ *Защищенность* от случайных или преднамеренных внешних воздействий («защита от дурака», вирусов, спама);
- **удобство применения** (usability) - ПО должно быть легким в использовании, причем именно тем типом пользователей, на которых рассчитано приложение. Это включает в себя интерфейс пользователя и адекватную документацию. Причем, пользовательский интерфейс должен быть не интуитивно, а профессионально понятным пользователю.;

- **эффективность** (efficiency) - это отношение уровня услуг, предоставляемых ПО пользователю при заданных условиях, к объему используемых ресурсов. Эффективность ПО оценивается следующими показателями:

- ✓ *время выполнения кода,*
- ✓ *загруженность процессора,*
- ✓ *объем требуемой памяти,*
- ✓ *время отклика и т.п.*

Вычислительная машина традиционно являлась критическим ресурсом, увеличение быстродействия должно достигаться не только за счет применения новой более производительной аппаратуры, но и за счет применения новых более эффективных алгоритмов, «распараллеливания» вычислений и т.д., то есть *эффективность программирования* становится все более важным фактором.;

- **сопровождаемость** (maintainability) - это характеристики ПС, которые позволяют минимизировать усилия по внесению изменений для устранения в нем ошибок и по его модификации в соответствии с изменяющимися потребностями пользователей. Это критическое свойство системы, т.к. изменения ПО неизбежны вследствие изменения бизнеса. Сопровождаемость включает такие элементы как наличие и понятность проектной документации, соответствие проектной документации исходному коду, понятность исходного кода, простота изменений исходного кода, простота добавления новых функций (см. п. 2.1.5);
- **переносимость** (portability) - машины и технические средства развиваются и дешевеют быстрее, чем программы, поэтому ПО должно быть легко переносимым на новые и более дешевые машины и с одной платформы на другую.

Базовая модель качества включает эти характеристики и относится к любому типу программных продуктов. При разработке требований заказчик формулирует те требования к качеству, которые наиболее подходят для заказываемого программного продукта. К дополнительным характеристикам качества относятся:

- **Корректность** - в первую очередь программа должна правильно работать, ошибочную программу невозможно использовать, даже если у нее будут другие полезные свойства;
- **Тестируемость (testability) и корректируемость**. Тестирование разделяется на *автономное*, когда проверяется работа отдельных модулей (компонентов), и *комплексное*, когда проверяется правильность совместной работы составных частей системы, при этом особое внимание уделяется взаимодействию

компонентов. Тестирование программ и проверка их корректности – часто наиболее объемная работа, чем их написание. Плохую программу нередко легче сделать заново, чем выполнить тестирование и коррекцию.

- **Отлаживаемость (debuggability).** Почти во всех уже оттестированных программах позже находятся ошибки или возникают потребности внести исправления. В отличие от тестирования, которое служит лишь для установления факта существования ошибок, отладка необходима их для локализации и устранения. Программы нужно создавать так, чтобы позднее было просто локализовать и исправить ошибки.
- **Удобочитаемость (readability) и понятность программ.** Помимо разработчиков и другие должны при необходимости в состоянии понять смысл программы и ознакомиться с текстом работающих функций (и прочей документацией). С увеличением среднего размера программ все большее их количество приходится создавать коллективными усилиями, чем еще больше подчеркивается значимость удобочитаемости.

*Определение и планирование качества ПО* основывается на положениях стандартов в этой области, составлении планов и графиков работ, процедурах проверки и др. План обеспечения качества включает набор действий для проверки процессов обеспечения качества (*верификация, валидация* и др.) и формирование документа по управлению качеством.

*Планирование качества* обеспечивает управление процессами обеспечения качества продуктов проекта (в частности промежуточных рабочих продуктов) и ресурсов - программных, технических, исполнительских и др., а также включает управление требованиями к процессам и продуктам. Планирование качества включает:

- определение продукта в терминах заданных характеристик качества;
- планирование процессов для гарантии получения требуемого качества;
- выбор методов оценки планируемых характеристик качества и установления соответствия продукта сформулированным требованиям.

В стандарте 12207 определены специальные процессы: обеспечения качества, верификации, аттестации (валидации), совместного анализа и аудита.

*Деятельности и техники гарантии качества* включают: инспекцию, верификацию и валидацию ПО.

*Инспекция ПО* - анализ и проверка различных представлений системы и ПО (спецификаций, архитектурных схем, диаграмм, исходного кода и др.). Выполняется на всех этапах ЖЦ разработки ПО.

*Целью инспекций* является обнаружение различных аномальных состояний в ПО

---

независимыми специалистами команды экспертов и с привлечением авторов



промежуточного или конечного продукта. Эксперты инспектирует выполнение требований, интерфейсы, входные данные и т.п., а затем документируют обнаруженные отклонения в проекте. *Назначением аудита* является независимая оценка продуктов и процессов на соответствие регулирующим и регламентирующим документам (планам, стандартам и др.), формулирование отчета о случаях несоответствия и предложений для их корректировки.

Верификация ПО - процесс обеспечения правильной реализации ПО, которое соответствует спецификациям, выполняется на протяжении всего жизненного цикла. Верификация дает ответ на вопрос, правильно ли создана система.

Валидация - процесс проверки соответствия ПО функциональным и нефункциональным требованиям и ожидаемым потребностям заказчика.

Верификация и валидация начинают выполняться на ранних стадиях ЖЦ и ориентированы на качество. Они планируются и обеспечиваются определенными ресурсами с четким распределением ролей. Проверка основывается на использовании соответствующих техник тестирования для обнаружения тех или иных дефектов и сбора статистики. В результате собранных данных проводится оценка правильности реализации требований и работы ПО в заданных условиях.

Измерение в анализе качества ПО основывается на метриках продукта и данных, собранных в процессе создания продукта при заданных ресурсах, на оценках процессов, ПО и его моделях, а также на документировании измерений. Оценка качества продукта - это измерение и оценивание качественных показателей с помощью данных о разных типах ошибок и отказах во время тестирования ПО и исполнения кода на тестовых данных. Эти данные анализируются, проверяются и используются при качественной и количественной оценке ПО.

Для имитации работы системы в режиме тестирования разрабатываются тесты с реальными входными данными для проверки правильности работы ПО на разных фрагментах программы и путях следования в них операторов. В процессе тестирования ПО обнаруживаются разного рода ошибки (отказы, дефекты, ошибки и т.п.), количество которых в значительной степени может повлиять на получение правильного результата.

С учетом типов обнаруженных ошибок можно установить наличие (или отсутствие) соответствия реализованных и нереализованных функций, заданных в требованиях к системе, а также оценить принципы реализации нефункциональных требований (производительность, надежность и др.). Проводятся также оценки процессов управления планами, инспекциями, прогонами и т.п. По этим оценкам принимаются решения о

завершении разработки продукта проекта и передаче его заказчику в опытную эксплуатацию или о необходимости внесения изменений для устранения ошибок, определения адекватности планов и требований, оценки рисков на переделку ПО и др.

### 3 ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ПРОДУКТА

Одним из ключевых понятий управления проектами, в том числе в приложении к индустрии программного обеспечения, является жизненный цикл проекта (Project Life Cycle Management - PLCM).

**Немного истории.** Появление понятия жизненного цикла ПО было связано с кризисом программирования, который наметился в конце 60-х – начале 70-х годов прошлого века. Суть кризиса состояла в том, что программные проекты все чаще стали выходить из-под контроля: нарушались сроки, превышались запланированные объемы финансирования, результаты не соответствовали требуемым. Многие проекты вообще не доводились до завершения. Кроме того, оказалось, что недостаточно разработать программу, а надо ее еще сопровождать и этап сопровождения часто требует больше средств, чем разработка.

Ситуация была вызвана ростом сложности проектов. Масштабы ее нарастали. Необходимо было принимать меры для радикального усовершенствования принципов и методов разработки ПО с учетом его развития и сопровождения. Заговорили о том, что надо обратиться к опыту промышленного проектирования и производства, где был накоплен опыт успешной разработки не менее сложных проектов.

Методологическую основу промышленной инженерии составляет понятие **жизненного цикла (ЖЦ) изделия (продукта)** как совокупности всех действий, которые надо выполнить на протяжении всей «жизни» изделия. Смысл жизненного цикла состоит во взаимосвязанности всех этих действий. Например, вы решили построить садовый домик, имея некоторый опыт строительных работ. Ориентируясь на здравый смысл, вы купили материалы, инструменты и построили дом. Через год он начал оседать одним углом – вы забыли его спроектировать и рассчитать фундамент. Через два года жена сказала, что лестница на мансарду должна быть не на веранде, где и так мало места, а с улицы. Лестница у вас была сработана фундаментально, а с улицы надо вырубать и перекрывать козырьки – вы не предусмотрели модификацию конструкции. Через три года у вас сгорела проводка, которая была «надежно» спрятана и вам пришлось снимать всю облицовку, которую вы «надежно» прибивали 100 мм гвоздями – вы не предусмотрели ремонтпригодность. Если бы вы изначально рассматривали строительство с позиции жизненного цикла, то на этапах проектирования и строительства вы подумали бы о надежности, модифицируемости и ремонтпригодности.

Итак, жизненный цикл промышленного изделия:

- последовательность этапов (фаз, стадий): проектирования, изготовления образца, организация производства, серийное производство, эксплуатация, ремонт, вывод из эксплуатации;
- состоящих из технологических процессов, действий и операций.

Организация промышленного производства с позиции жизненного цикла позволяет рассматривать все его этапы во взаимосвязи, что ведет к сокращению сроков, стоимости и трудозатрат.

Впервые о жизненном цикле ПО заговорили в 1968 г. в Лондоне, где состоялась встреча 22-х руководителей проектов по разработке ПО. На встрече анализировались проблемы и перспективы проектирования, разработки, распространения и поддержки программ. Применяющиеся принципы и методы разработки ПО требовали постоянного усовершенствования. Именно на этой встрече была предложена концепция жизненного цикла ПО (SLC – Software Lifetime Cycle) как последовательности шагов-стадий, которые необходимо выполнить в процессе создания и эксплуатации ПО.

Вокруг этой концепции было много споров. В 1970 г. У.У. Ройс (W.W. Royce) произвел идентификацию нескольких стадий в типичном цикле и было высказано предположение, что контроль выполнения стадий приведет к повышению качества ПО и сокращению стоимости разработки.

Основным стандартом, регламентирующим концепцию ЖЦ ПС, стал стандарт ISO/IEC 12207 - Information Technology - Software Life Cycle Processes, разработанный в 1995 (в России он известен как ГОСТ 12207. Процессы жизненного цикла программных средств).

Стандарт ISO 12207 разрабатывался с учетом лучшего мирового опыта на основе вышеперечисленных стандартов. Основными результатами стандарта ISO 12207 являются:

- введение единой терминологии по разработке и применению ПО (предназначен не только для разработчиков, но и для заказчиков, пользователей, всех заинтересованных лиц);
- разделение понятий ЖЦ ПО и модели ЖЦ ПО. ЖЦ ПО в стандарте вводится как полная совокупность всех процессов и действий по созданию и применению ПО, а модель ЖЦ – конкретный вариант организации ЖЦ, обоснованно (разумно) выбранный для каждого конкретного случая;
- описание организации ЖЦ и его структуры (процессов);

- выделение процесса адаптации стандарта для построения конкретных моделей ЖЦ.

### Стандарт ISO 12207. Основные определения

**Программный продукт (software product)** - набор машинных программ, процедур и, возможно, связанных с ними документации и данных.

**Жизненный цикл ПП (software life cycle)** – это непрерывный процесс, который начинается с момента принятия решения о необходимости его создания и заканчивается в момент его полного изъятия из эксплуатации.

**Процесс (process)** - набор взаимосвязанных работ, которые преобразуют исходные данные в выходные результаты.

**Организация ЖЦ ПП** - совокупность процессов, каждый из которых разбит на действия, состоящие из отдельных задач.

**Структура (архитектура) ЖЦ ПП** - перечень процессов, действий и задач.

В общем случае, жизненный цикл определяется *моделью* и описывается в форме *методологии* (метода). **Модель** или парадигма жизненного цикла определяет концептуальный взгляд на организацию жизненного цикла и, часто, основные фазы жизненного цикла и принципы перехода между ними. **Методология (метод)** задает комплекс работ, их детальное содержание и ролевую ответственность специалистов на всех этапах выбранной модели жизненного цикла, обычно определяет и саму модель, а также рекомендует *практики (best practices)*, позволяющие максимально эффективно воспользоваться соответствующей методологией и ее моделью.

**Модель жизненного цикла** - структура, состоящая из процессов, работ и задач, включающих в себя разработку, эксплуатацию и сопровождение ПП, охватывающая жизнь системы от установления требований к ней до прекращения ее использования.

Говоря «жизненный цикл» мы, в первую очередь, подразумеваем «модель жизненного цикла». Несмотря на данное в стандартах 12207 определение модели жизненного цикла, все же, *модель* чаще подразумевает именно *общий принцип* организации ЖЦ, чем детализацию соответствующих работ. Соответственно, определение и выбор модели, в первую очередь, касается вопросов определенности и стабильности требований, жесткости и детализированности плана работ, а также частоты сборки работающих версий создаваемой программной системы.

Скотт Амблер (Scott W. Ambler) [Ambler, 2005], автор концепций и практик гибкого моделирования (Agile Modeling) и Enterprise Unified Process (расширение Rational Unified

Process), предлагает следующие уровни жизненного цикла, определяемые соответствующим содержанием работ (см. рисунок 3.1):

- Жизненный цикл разработки ПО - проектная деятельность по разработке и развертыванию программных систем.
- Жизненный цикл ПС - включает разработку, развертывание, поддержку и сопровождение.
- Жизненный цикл ИТ - включает всю деятельность ИТ-департамента.
- Жизненный цикл организации/бизнеса - охватывает всю деятельность организации в целом.

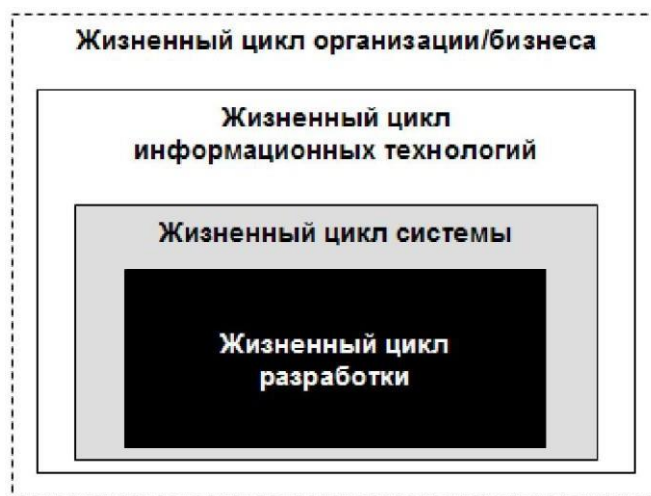


Рисунок 3.1 - Содержание четырех категорий жизненного цикла по Амблеру

В данном контексте, SWEBOOK описывает области знаний *жизненного цикла системы* и *жизненного цикла разработки ПО*. В свою очередь, как упоминается в SWEBOOK, одним из фундаментальных взглядов на жизненный цикл является стандарт процессов жизненного цикла ISO/IEC, IEEE, ГОСТ Р ИСО/МЭК 12207.

### Стандарт ISO 12207. Процессы ЖЦ

Цель разработки данного стандарта была определена как создание общего фреймворка по организации жизненного цикла программного обеспечения для формирования общего понимания ЖЦ ПО всеми заинтересованными сторонами и участниками процесса разработки приобретения, поставки, эксплуатации, поддержки и сопровождения программных систем, а также возможности управления, контроля и совершенствования процессов жизненного цикла.

Данный стандарт определяет ЖЦ как *структуру декомпозиции работ*. Детализация, техники и метрики проведения работ – вопрос программной инженерии. Организация

последовательности работ – модель жизненного цикла. Совокупность моделей, процессов, техник и организации проектной группы задаются методологией. В частности, выбор и применение метрик оценки качества программной системы и процессов находятся за рамками стандарта 12207, а концепция совершенствования процессов рассматривается в стандарте ISO/IEC 15504 «Information Technology - Software Process Assessment» («Оценка процессов <в области> программного обеспечения»).

Стандарт ISO 12207 определяет организацию ЖЦ программного продукта как совокупность 17 процессов, каждый из которых разбит на действия, состоящие из отдельных задач. В соответствии с ним все процессы ЖЦ делятся на три группы (рисунок 3.2):

- *основные* (Primary Processes);
- *вспомогательные* (Supporting Processes);
- *организационные* (Organizational Processes).

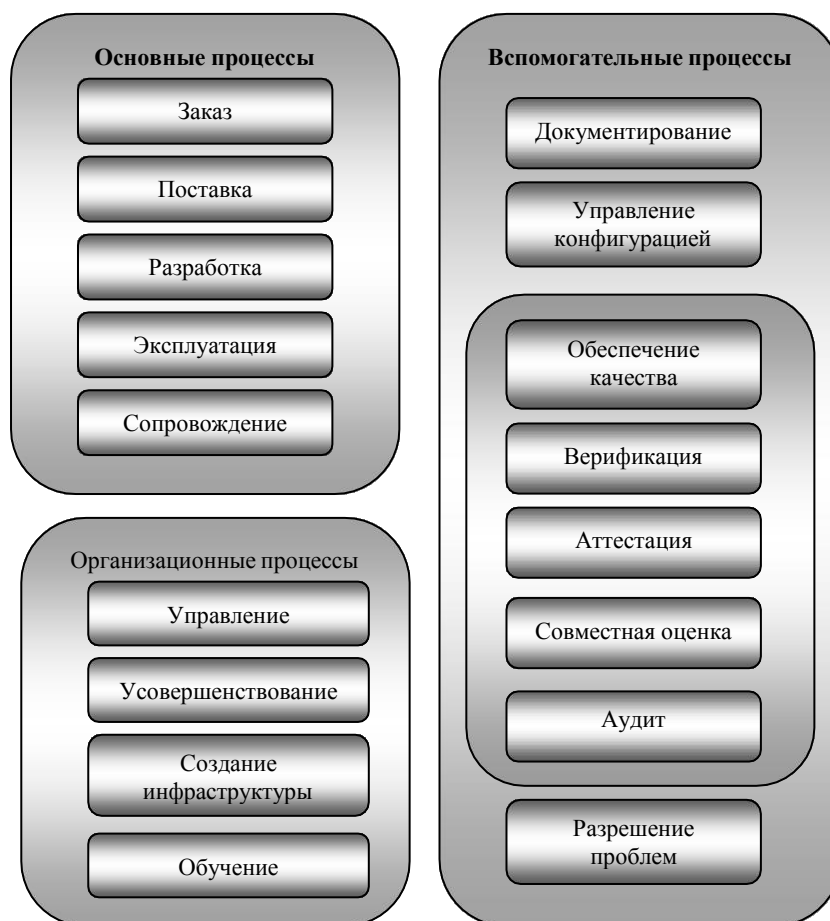


Рисунок 3.2 – Структура процессов ЖЦ ПО

Отдельно описан процесс адаптации стандарта, содержащий основные работы, которые должны быть выполнены при адаптации настоящего стандарта к условиям конкретного программного проекта.

Стандарт отмечает, что все работы проводятся с использованием **проектного подхода** и *могут пересекаться по времени*, т.е. проводиться одновременно или с наложением, а также могут предполагать *рекурсию и разбиение на итерации*.

К числу **основных процессов** относятся:

- **Заказ.** Определяет работы и задачи заказчика, то есть организации, которая приобретает систему, программный продукт или программную услугу, связанную с ПО, на основе контрактных отношений: инициирование (подготовка); подготовка заявки на подряд; подготовка и корректировка договора; мониторинг поставщика; приемка и закрытие договора.
- **Поставка.** Определяет работы поставщика, то есть организации, которая поставляет систему, программный продукт или программную услугу заказчику. Работы также проводятся с использованием проектного подхода (см. выше, дополнительные работы: выполнение и контроль; проверка и оценка).
- **Разработка.** Определяет работы разработчика, то есть организации, которая проектирует и разрабатывает программный продукт: определение процесса; анализ системных требований; проектирование системы; анализ требований к программным средствам; проектирование программной архитектуры; детальное проектирование программной системы (техническое проектирование программных средств); кодирование и тестирование (программирование и тестирование программных средств); интеграция программной системы (сборка программных средств); квалификационные испытания программных средств; интеграция системы в целом (сборка системы); квалификационные испытания системы; установка (ввод в действие); обеспечение приемки программных средств.
- **Эксплуатация.** Определяет работы оператора, то есть организации, которая обеспечивает эксплуатационное обслуживание вычислительной системы в заданных условиях в интересах пользователей: определение процесса; операционное тестирование (эксплуатационные испытания); эксплуатация системы; поддержка пользователя.
- **Сопровождение.** Определяет работы персонала сопровождения, то есть организации, которая предоставляет услуги по сопровождению программного



продукта, состоящие в контролируемом изменении программного продукта с целью сохранения его исходного состояния и функциональных возможностей: определение процесса; анализ проблем и изменений; внесение изменений; проверка и приемка при сопровождении; миграция (перенос); вывод программной системы из эксплуатации (снятие с эксплуатации).

**Вспомогательными процессами** являются:

- *Документирование.* Определяет работы по описанию информации, выдаваемой в процессе жизненного цикла.
- *Управление конфигурацией.* Определяет работы по управлению конфигурацией.
- *Обеспечение качества.* Определяет работы по объективному обеспечению того, чтобы программные продукты и процессы соответствовали требованиям, установленным для них, и реализовывались в рамках утвержденных планов. Совместные анализы, аудиторские проверки, верификация и аттестация могут использоваться в качестве методов обеспечения качества.
- *Верификация.* Определяет работы (заказчика, поставщика или независимой стороны) по верификации ПП по мере реализации программного проекта.
- *Аттестация.* Определяет работы (заказчика, поставщика или независимой стороны) по аттестации программных продуктов программного проекта.
- *Совместный анализ.* Определяет работы по оценке состояния и результатов какой-либо работы. Данный процесс может использоваться двумя любыми сторонами, когда одна из сторон (проверяющая) проверяет другую сторону (проверяемую) на совместном совещании.
- *Аудит.* Определяет работы по определению соответствия требованиям, планам и договору. Данный процесс может использоваться двумя сторонами, когда одна из сторон (проверяющая) контролирует программные продукты или работы другой стороны (проверяемой).
- *Решение проблем.* Определяет процесс анализа и устранения проблем (включая несоответствия), независимо от их характера и источника, которые были обнаружены во время осуществления разработки, эксплуатации, сопровождения или других процессов.

**Организационными процессами** являются:

- *Управление.* Определяет основные работы по управлению, включая управление проектом, при реализации процессов жизненного цикла.

- *Создание инфраструктуры.* Определяет основные работы по созданию основной структуры процесса жизненного цикла.
- *Усовершенствование.* Определяет основные работы, которые организация (заказчика, поставщика, разработчика, оператора, персонала сопровождения или администратора другого процесса) выполняет при создании, оценке, контроле и усовершенствовании выбранных процессов жизненного цикла.
- *Обучение.* Определяет работы по соответствующему обучению персонала.

Дерево процессов жизненного цикла представляет собой структуру декомпозиции жизненного цикла на соответствующие процессы (группы процессов). *Декомпозиция процессов строится на основе двух важнейших принципов, определяющих правила разбиения (partitioning) жизненного цикла на составляющие процессы. Эти принципы:*

#### **Модульность**

- задачи в процессе являются функционально связанными;
- связь между процессами - минимальна;
- если функция используется более, чем одним процессом, она сама является процессом;
- если Процесс Y используется Процессом X и только им, значит Процесс Y принадлежит (является его частью или его задачей) Процессу X, за исключением случаев потенциального использования Процесса Y в других процессах в будущем.

#### **Ответственность**

- каждый процесс находится под ответственностью конкретного лица (управляется и/или контролируется им), определенного для заданного жизненного цикла, например, в виде роли в проектной команде;
- функция, чьи части находятся в компетенции различных лиц, не может рассматриваться как самостоятельный процесс.

*Модель жизненного цикла разработки ПО* является единственным видом процесса, в котором представлен порядок его осуществления. Модель жизненного цикла разработки ПО (Software Life Cycle Model, SLCM) схематически объясняет, каким образом будут выполняться действия по разработке программного продукта, посредством описания «последовательности» этих действий. Такая последовательность может быть или не быть линейной, поскольку фазы могут следовать друг за другом, повторяться или происходить одновременно. На рисунке 3.3 представлена простая обобщенная схема процесса.

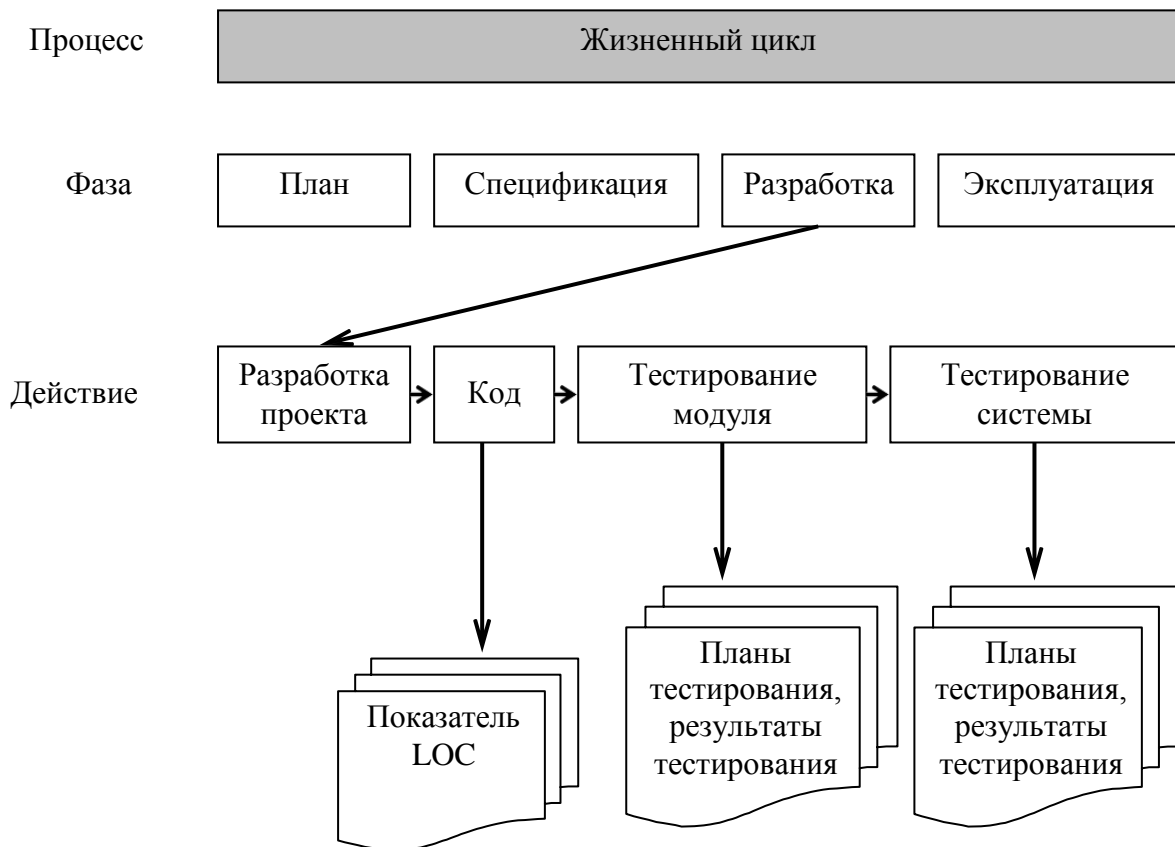


Рисунок 3.3 - . Обобщенная схема процесса

Общая иерархия (декомпозиция) составных элементов жизненного цикла выглядит следующим образом (рисунок 3.4):

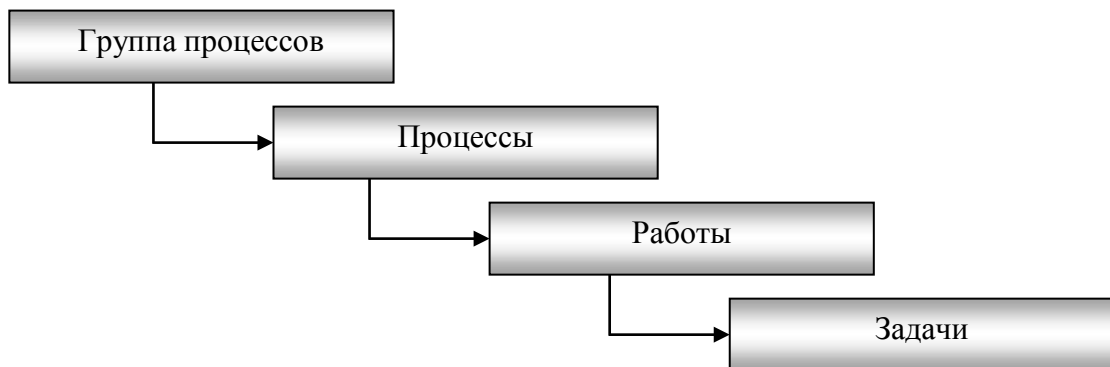


Рисунок 3.4 - Иерархия составных элементов жизненного цикла ПП

В общем случае, разбиение процесса базируется на широко распространенном **PDCA-цикле**:

- «P» - *Plan* – Планирование;
- «D» - *Do* - Выполнение;
- «C» - *Check* - Проверка;
- «A» - *Act* - Действие (реакция).

### Модели жизненного цикла ПП

Типовая модель процессов жизненного цикла сложной системы начинается с *концепции идеи системы* или потребности в ней, охватывает *проектирование, разработку, применение и сопровождение* системы, и заканчивается снятием системы с эксплуатации. Программные средства служат для выполнения определенных функций систем на компьютерах. Модель жизненного цикла системы обычно разделяют на последовательные периоды реализации - *стадии* или *этапы*.

Каждый подобный период включает основные реализуемые в нем процессы, работы и задачи, при завершении которых может потребоваться переход к следующему периоду реализации. *Общую модель жизненного цикла* сложной системы обычно разделяют на следующие основные этапы с последующей адаптацией каждого из них в модели жизненного цикла конкретной системы [2]:

- определение потребностей;
- исследование и описание основных концепций;
- проектирование и разработка;
- испытания системы;
- создание и производство;
- распространение и продажа;
- эксплуатация;
- сопровождение и мониторинг;
- снятие с эксплуатации (утилизация).

Крупномасштабные комплексы программ являются компонентами систем, реализующими обычно их основные, функциональные свойства, увеличивающими сложность и создающими предпосылки для последующих изменений их жизненного цикла. Реализация ЖЦ, методологии управления и изменения ПС зависит от многих факторов, от персонала, технических, организационных и договорных требований и сложности проекта. Множество текущих состояний и модификаций компонентов сложных ПС менеджерам необходимо упорядочивать, контролировать их развитие и применение участниками проекта. Организованное, контролируемое и методичное отслеживание динамики изменений в жизненном цикле программ и данных, их слаженная разработка при строгом учете и контроле каждого изменения, является основой эффективного, поступательного развития каждой крупной системы *методами программной инженерии*.

Существует множество моделей процессов жизненного цикла систем и программных средств, но три из них в международных стандартах обычно квалифицируются как фундаментальные: *каскадная*; *инкрементная*; *эволюционная*. Каждая из указанных моделей может быть использована самостоятельно или скомбинирована с другими для создания гибридной модели жизненного цикла конкретного проекта. При этом конкретную модель жизненного цикла системы или ПС следует выбирать так, чтобы процессы и задачи были связаны между собой, и определены их взаимосвязи с предшествующими процессами, видами деятельности и задачами [2].

### ***Каскадная модель (1970-1985 г.г.)***

Классическая каскадная модель, несмотря на полученную в последнее время негативную оценку, исправно служила специалистам по программному инжинирингу многие годы. Понимание ее сильных сторон и недостатков улучшает оценочный анализ других, зачастую более эффективных моделей жизненного цикла, основанных на данной модели [Карпенко].

В первые годы практики программирования сначала записывался программный код, а затем происходила его отладка. Общепринятым считалось правило начинать работу не с разработки плана, а с общего ознакомления с продуктом. Без лишних формальностей можно было спроектировать, закодировать, отладить и протестировать ПО еще до того, как оно будет готово к выпуску. Это напоминало процесс, изображенный на рисунке 3.5.

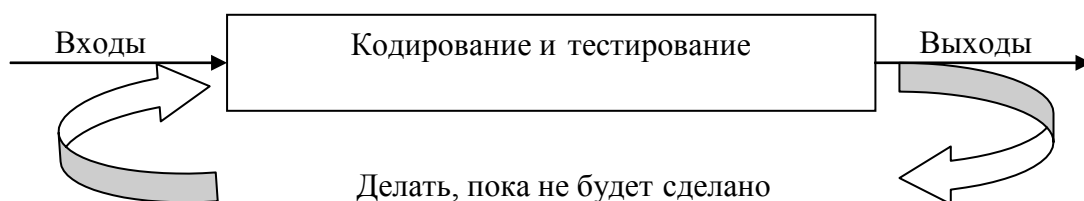


Рисунок 3.5 – Модель процесса «делать, пока не будет сделано»

В структуре такого процесса есть несколько недостатков. Во-первых, поскольку изначально не существовало официального проекта или анализа, невозможно было узнать о моменте завершения процесса. Также отсутствовал способ определения соответствия требованиям относительно достижения качества.

В 1970 году каскадная модель была впервые определена как альтернативный вариант метода разработки ПО по принципу *кодирование-устранение ошибок*. Это была первая модель, которая формализовала структуру этапов разработки ПО, придавая особое значение исходным требованиям и проектированию, а также созданию документации на ранних этапах процесса разработки.

Основной характеристикой каскадного способа является разбиение всей разработки на этапы, причем переход с одного этапа на следующий происходит только после того, как будет полностью завершена работа на текущем. Каждый этап завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков [27]. *Основа модели* – сформулированные требования в техническом задании (ТЗ), которые меняться не должны. *Критерий качества результата* – соответствие продукта установленным требованиям.

Каскадная модель (водопад – waterfall) включает выполнение следующих фаз (рисунок 3.6):

- **исследование концепции** — происходит исследование требований, разрабатывается видение продукта и оценивается возможность его реализации;
- **определение требований** — определяются программные требования для предметной области системы, предназначение, линии поведения, производительность и интерфейсы;
- **разработка проекта** — разрабатывается и формулируется логически последовательная техническая характеристика программной системы, включая структуры данных, архитектуру ПО, интерфейсные представления и процессуальную (алгоритмическую) детализацию;
- **реализация** — эскизное описание ПО превращается в полноценный программный продукт. Результат: исходный код, база данных и документация. В реализации обычно выделяют два этапа: реализацию компонентов ПО и интеграцию компонентов в готовый продукт. На обоих этапах выполняется кодирование и тестирование, которые тоже иногда рассматривают как два подэтапа;
- **эксплуатация и поддержка** – подразумевает запуск и текущее обеспечение, включая предоставление технической помощи, обсуждение возникших вопросов с пользователем, регистрацию запросов пользователя на модернизацию и внесение изменений, а также корректирование или устранение ошибок;
- **сопровождение** — устранение программных ошибок, неисправностей, сбоев, модернизация и внесение изменений.

Основными *принципами* каскадной модели являются:

- строго последовательное выполнение фаз;
- каждая последующая фаза начинается лишь тогда, когда полностью завершено выполнение предыдущей фазы;

- каждая фаза имеет определенные критерии входа и выхода: входные и выходные данные;
- каждая фаза полностью документируется;
- переход от одной фазы к другой осуществляется посредством формального обзора с участием заказчика.

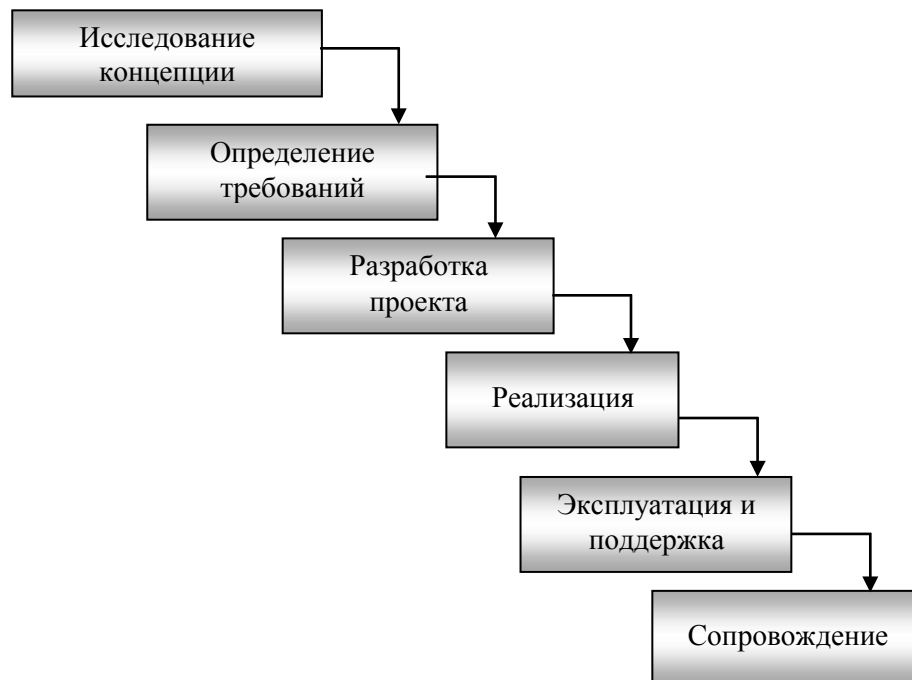


Рисунок 3.6– Классическая каскадная схема разработки ПО

**Преимущества и недостатки.** Каскадная модель имеет следующие *преимущества*:

- *проста и понятна* заказчикам, так как часто используется другими организациями для отслеживания проектов, не связанных с разработкой ПО;
- проста и удобна в применении;
- процесс *разработки* выполняется *поэтапно*;
- ее структурой может руководствоваться даже слабо подготовленный в техническом плане или неопытный персонал;
- она способствует осуществлению строгого контроля менеджмента проекта;
- каждую стадию могут выполнять независимые команды (*все документировано*);
- позволяет достаточно точно планировать сроки и затраты.

При использовании каскадной модели для «неподходящего» проекта могут проявляться следующие ее *недостатки*:

- *невозможно вернуться* на одну или две фазы назад, чтобы исправить какую-либо проблему или недостаток, это приведет к значительному увеличению затрат и сбою в графике;
- *интеграция* компонентов, на которой обычно выявляется большая часть ошибок, выполняется *в конце разработки*, что сильно увеличивает стоимость устранения ошибок;
- *запаздывание результатов*, если в процессе выполнения проекта требования изменились, то получится устаревший результат.

Недостатки каскадной модели особо остро проявляются в случае, когда трудно (или невозможно) сформулировать требования или требования могут меняться в процессе выполнения проекта. В этом случае разработка ПО имеет принципиально циклический характер.

**Применимость.** Каскадная модель впервые четко сформулирована в 1970 году Ройсом [28], на начальном периоде она сыграла ведущую роль как метод регулярной разработки сложного ПО. В семидесятых-восемидесятых годах XX века модель была принята как стандарт министерства обороны США. Со временем недостатки каскадной модели стали проявляться все чаще, и возникло мнение, что она безнадежно устарела.

Между тем, каскадная модель не утратила своей *актуальности* при решении следующих типов задач:

- ✓ требования и их реализация максимально четко определены и понятны; используется неизменяемое определение продукта и вполне понятные технические методики (задачи научно-вычислительного характера, операционные системы и компиляторы, системы реального времени управления конкретными объектами);
- ✓ повторная разработка типового продукта (автоматизированного бухгалтерского учета, начисления зарплаты и т.п.);
- ✓ выпуск новой версии уже существующего продукта, если вносимые изменения вполне определены и управляемы (перенос уже существующего продукта на новую платформу).

В настоящее время используется усовершенствованная каскадная модель, которая позволяет при необходимости вернуться на любой предыдущий этап и внести необходимые изменения (рисунок 3.7).



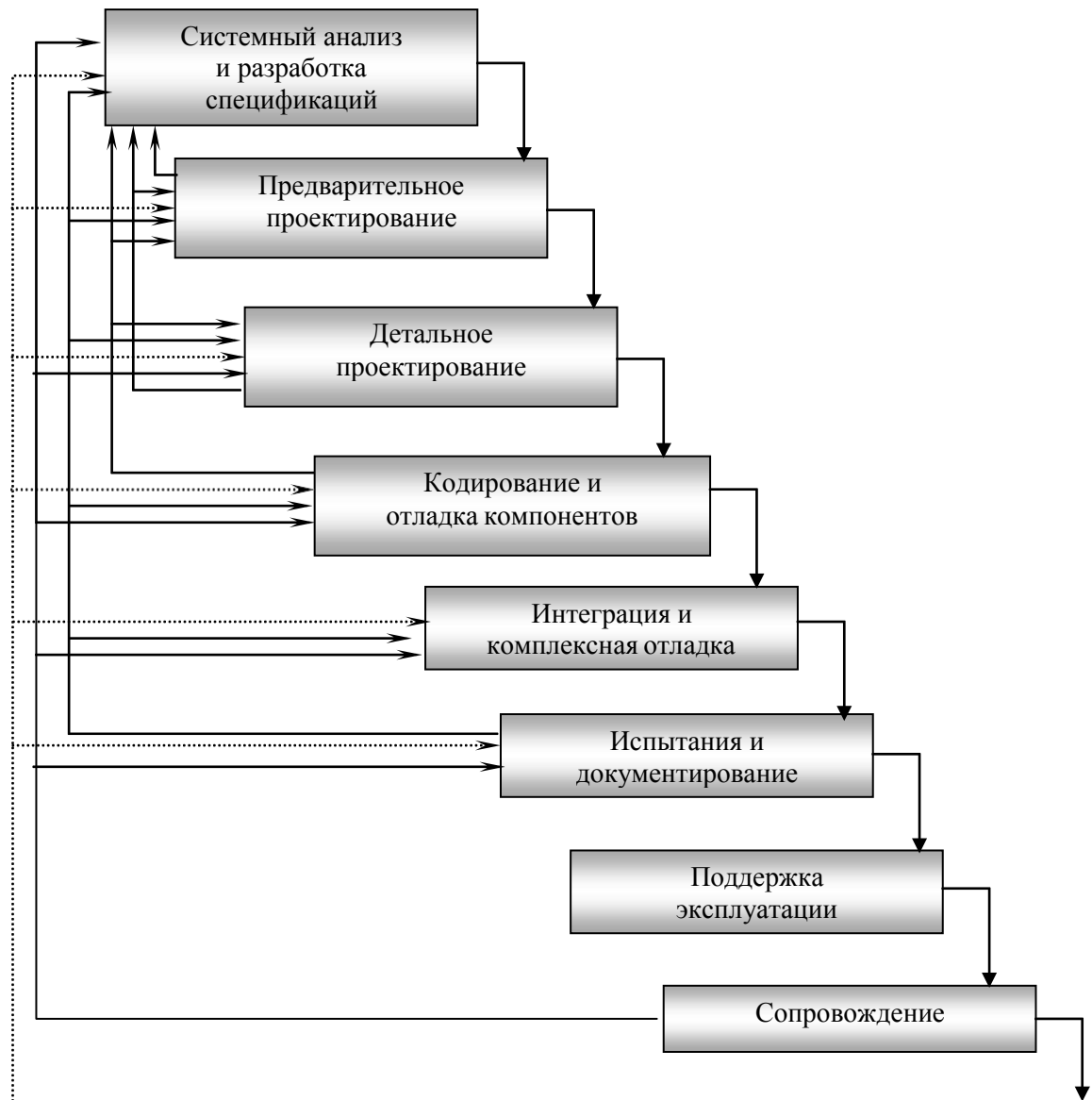


Рисунок 3.7 - Обобщенная каскадная модель ЖЦ ПО

### ***Итеративная и инкрементальная модель – эволюционный подход***

Итеративная модель предполагает разбиение жизненного цикла проекта на последовательность итераций, каждая из которых напоминает «мини-проект», включая все фазы жизненного цикла в применении к созданию меньших фрагментов функциональности, по сравнению с проектом, в целом. Цель каждой итерации – получение работающей версии программной системы, включающей функциональность, определенную интегрированным содержанием всех предыдущих и текущей итерации. Результата финальной итерации содержит всю требуемую функциональность продукта. Таким образом, с завершением каждой итерации, продукт развивается инкрементально.

С точки зрения структуры жизненного цикла такую модель называют *итеративной* (iterative). С точки зрения развития продукта – *инкрементальной* (incremental). Опыт индустрии показывает, что невозможно рассматривать каждый из этих взглядов изолированно. Чаще всего такую смешанную эволюционную модель называют просто итеративной (говоря о процессе) и/или инкрементальной (говоря о наращивании функциональности продукта).

Эволюционная модель подразумевает не только сборку работающей (с точки зрения результатов тестирования) версии системы, но и ее развертывание в реальных операционных условиях с анализом откликов пользователей для определения содержания и планирования следующей итерации. «Чистая» инкрементальная модель не предполагает развертывания промежуточных сборок (релизов) системы и все итерации проводятся по заранее определенному плану наращивания функциональности, а пользователи (заказчик) получают только результат финальной итерации как полную версию системы. С другой стороны, Скотт Амблер, например, определяет эволюционную модель как сочетание итеративного и инкрементального подходов. В свою очередь, Мартин Фаулер [29, с.47] пишет: «Итеративную разработку называют по-разному: инкрементальной, спиральной, эволюционной и постепенной. Разные люди вкладывают в эти термины разный смысл, но эти различия не имеют широкого признания и не так важны, как противостояние итеративного метода и метода водопада».

Брукс пишет [30, с. 246-247], что, «в идеале, поскольку на каждом шаге мы имеем работающую систему:

- можно очень рано начать тестирование пользователями;
- можно принять стратегию разработки в соответствии с бюджетом, полностью защищающую от перерасхода времени или средств (в частности, за счет сокращения второстепенной функциональности)».

Итерационная модель является более жизненной, чем классическая каскадная модель, т.к. создание ПО всегда связано с устранением ошибок. Следует отметить, что уже в первых работах, посвященных каскадной модели, отмечалось это обстоятельство и предлагался итерационный вариант каскадной модели. Практически все применяемые модели жизненного цикла имеют итерационный характер, но цели итераций могут быть разными. Основная опасность использования такой схемы связана с тем, что разработка никогда не будет завершена, постоянно находясь в состоянии уточнения и усовершенствования.

На рисунке 3.8 приведено соотношение между неопределенностью проекта и его функциональностью при итеративной организации жизненного цикла.

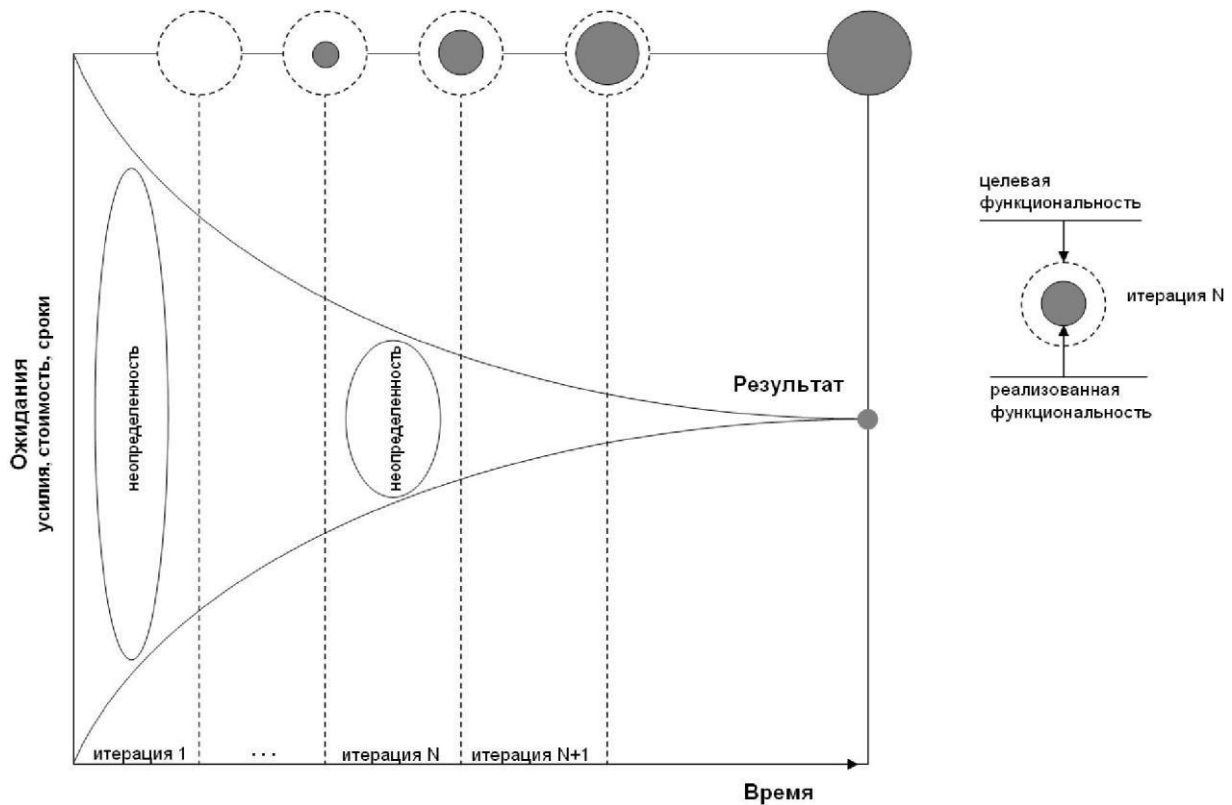


Рисунок 3.8 - Снижение неопределенности и инкрементальное расширение функциональности при итеративной организации жизненного цикла

Наиболее известным и распространенным вариантом эволюционной модели является спиральная модель.

### ***Спиральная модель***

Спиральная модель (представлена на рисунке 3.9) была впервые сформулирована Барри Бозмом (Barry Boehm) в 1988 году. Отличительной особенностью этой модели является специальное внимание рискам, влияющим на организацию жизненного цикла. Большая часть этих рисков связана с организационными взаимодействиями специалистов в проектной команде.

На практике, при решении достаточно большого количества задач, разработка ПО имеет циклический характер, когда после выполнения некоторых стадий приходится возвращаться на предыдущие. Это может происходить по следующим основным причинам:

- из-за *ошибок разработчиков*, допущенных на ранних стадиях и выявленных на поздних стадиях (это ошибки анализа, проектирования, кодирования, выявляемые, как правило, на стадии тестирования);

– из-за *изменения требований в процессе разработки* – «ошибки» заказчиков (это или неготовность заказчиков сформулировать требования, или изменения требований, вызванные изменениями ситуации в процессе разработки (изменения рынка, новые технологии и т.п.)).

Бозм формулирует «top-10» наиболее распространенных (по приоритетам) рисков:

- 1) Дефицит специалистов.
- 2) Нереалистичные сроки и бюджет.
- 3) Реализация несоответствующей функциональности.
- 4) Разработка неправильного пользовательского интерфейса.
- 5) «Золотая сервировка», ненужная оптимизация и оттачивание деталей.
- 6) Непрерывающийся поток изменений.
- 7) Нехватка информации о внешних компонентах, определяющих окружение системы или вовлеченных в интеграцию.
- 8) Недостатки в работах, выполняемых внешними (по отношению к проекту) ресурсами.
- 9) Недостаточная производительность получаемой системы.
- 10) «Разрыв» в квалификации специалистов разных областей знаний.

Спиральная модель была предложена как альтернатива каскадной модели для преодоления перечисленных выше проблем и учитывает повторяющийся характер разработки программного обеспечения.

Основными **принципами** спиральной модели являются:

- разработка вариантов продукта, соответствующих различным вариантам требований с возможностью вернуться к более ранним вариантам;
- создание прототипов ПО как средства общения с заказчиком для уточнения и выявления требований;

**Прототипом** называют действующий программный продукт, реализующий отдельные функции и внешние интерфейсы разрабатываемой ПС.

- планирование следующих вариантов с оценкой альтернатив и анализом рисков, связанных с переходом к следующему варианту;
- переход к разработке следующего варианта до завершения предыдущего в случае, когда риск завершения очередного варианта (прототипа) становится неоправданно высок;
- использование каскадной модели как схемы разработки очередного варианта;

– активное привлечение заказчика к работе над проектом (заказчик участвует в оценке очередного прототипа ПО, в уточнении требований при переходе к следующему, в оценке предложенных альтернатив очередного варианта и оценке рисков).

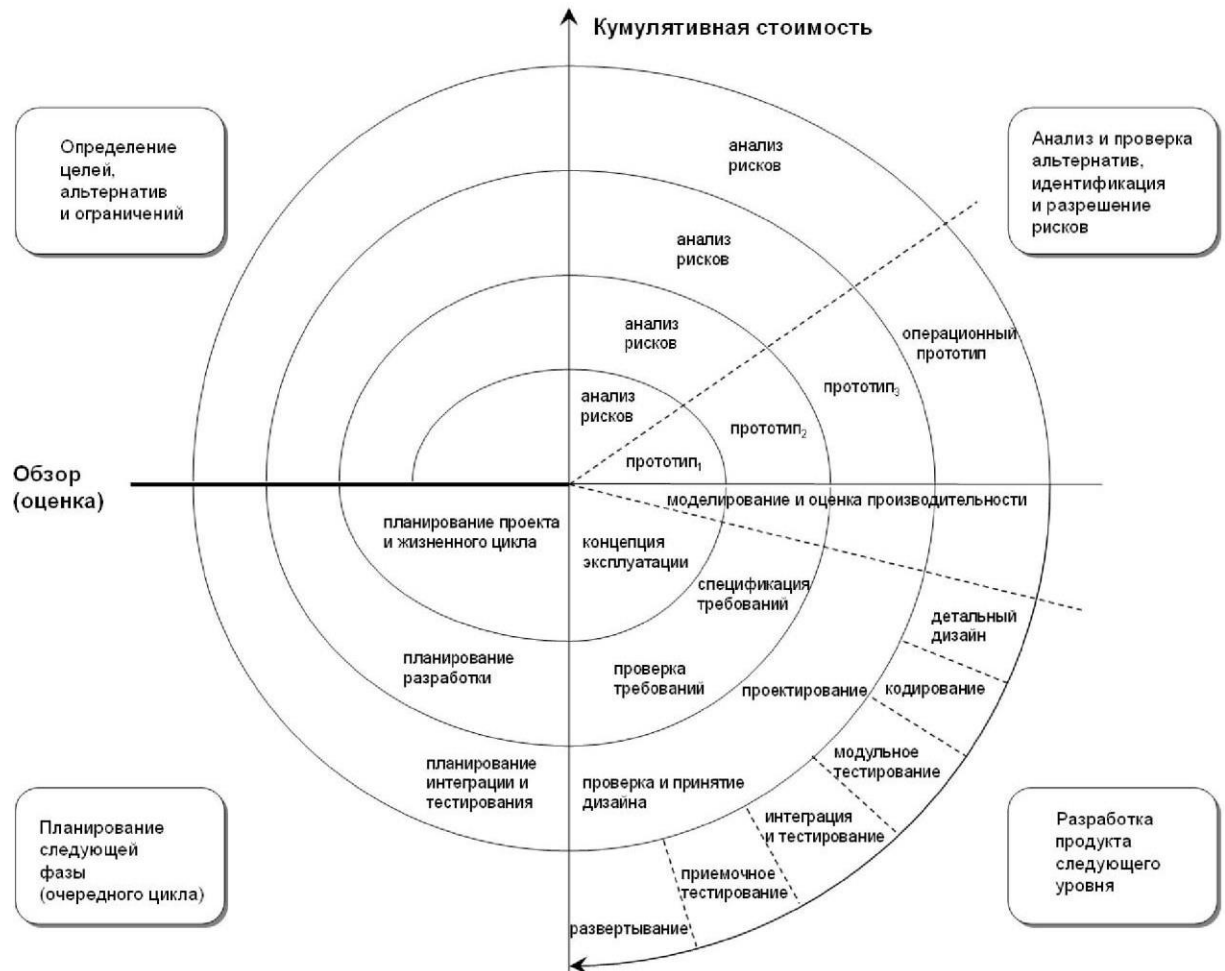


Рисунок 3.9 - Оригинальная спиральная модель жизненного цикла разработки по Бозму

***Спиральная модель обладает рядом преимуществ:***

- 1) Модель уделяет специальное внимание *раннему анализу возможностей повторного использования*. Это обеспечивается, в первую очередь, в процессе идентификации и оценки альтернатив.
- 2) Модель предполагает *возможность эволюции жизненного цикла, развитие и изменение программного продукта*. Главные источники изменений заключены в целях, для достижения которых создается продукт. Подход, предусматривающий скрытие информации о деталях на определенном уровне дизайна, позволяет рассматривать различные архитектурные альтернативы так, как если бы мы говорили о единственном проектном решении, что уменьшает риск невозможности согласования функционала продукта и изменяющихся целей (требований).

- 3) Модель предоставляет *механизмы достижения необходимых параметров качества* как составную часть процесса разработки программного продукта. Эти механизмы строятся на основе идентификации всех типов целей (требований) и ограничений на всех “циклах” спирали разработки. Например, ограничения по безопасности могут рассматриваться как риски на этапе специфицирования требований.
- 4) Модель уделяет *специальное внимание предотвращению ошибок* и отбрасыванию ненужных, необоснованных или неудовлетворительных альтернатив на ранних этапах проекта. Это достигается явно определенными работами по анализу рисков, проверке различных характеристик создаваемого продукта (включая архитектуру, соответствие требованиям и т.п.) и подтверждение возможности двигаться дальше на каждом «цикле» процесса разработки.
- 5) Модель позволяет *контролировать источники проектных работ и соответствующих затрат*. По-сути речь идет об ответе на вопрос – как много усилий необходимо затратить на анализ требований, планирование, конфигурационное управление, обеспечение качества, тестирование, формальную верификацию и т.д. Модель, ориентированная на риски, позволяет в контексте конкретного проекта решить задачу приложения адекватного уровня усилий, определяемого уровнем рисков, связанных с недостаточным выполнением тех или иных работ.
- 6) Модель не проводит *различий между разработкой нового продукта и расширением (или сопровождением) существующего*. Этот аспект позволяет избежать часто встречающегося отношения к поддержке и сопровождению как ко “второсортной” деятельности. Такой подход предупреждает большого количество проблем, возникающих в результате одинакового уделения внимания как обычному сопровождению, так и критичным вопросам, связанным с расширением функциональности продукта, всегда ассоциированным с повышенными рисками.
- 7) Модель позволяет *решать интегрированные задачи системной разработки*, охватывающей и программную и аппаратную составляющие создаваемого продукта. Подход, основанный на управлении рисками и возможности своевременного отбрасывания непривлекательных альтернатив (на ранних стадиях проекта) сокращает расходы и одинаково применим и к аппаратной части, и к программному обеспечению.

В 2000 году [Boehm, 2000], представляя анализ использования спиральной модели и, в частности, построенного на его основе подхода MBASE - Model-Based (System) Architecting and Software Engineering (MBASE), Боэм формулирует **6 ключевых практик** или характеристик, обеспечивающих успешное применение спиральной модели:

1. Параллельное, а не последовательное определение артефактов (активов) проекта
2. Согласие в том, что на каждом цикле уделяется внимание:
  - целям и ограничениям, важным для заказчика;
  - альтернативам организации процесса и технологических решений, закладываемых в продукт;
  - идентификации и разрешению рисков;
  - оценки со стороны заинтересованных лиц (в первую очередь заказчика);
  - достижению согласия в том, что можно и необходимо двигаться дальше.
3. Использование соображений, связанных с рисками, для определения уровня усилий, необходимого для каждой работы на всех циклах спирали.
4. Использование соображений, связанных с рисками, для определения уровня детализации каждого артефакта, создаваемого на всех циклах спирали.
5. Управление жизненным циклом в контексте обязательств всех заинтересованных лиц на основе трех контрольных точек:
  - Life Cycle Objectives (LCO);
  - Life Cycle Architecture (LCA);
  - Initial Operational Capability (IOC).
6. Уделение специального внимания проектным работам и артефактам создаваемой системы (включая непосредственно разрабатываемое программное обеспечение, ее окружение, а также эксплуатационные характеристики) и жизненного цикла (разработки и использования).

Эволюционирование спиральной модели, таким образом, связано с вопросами детализации работ. Особенно стоит выделить акцент на большем внимании вопросам уточнения - требований, дизайна и кода, т.е. придание большей важности вопросам итеративности, в том числе, увеличения их количества при сокращении длительности каждой итерации. В результате, можно определить общий набор контрольных точек в сегодняшней спиральной модели:

- Concept of Operations (COO) - концепция <использования> системы;
- Life Cycle Objectives (LCO) - цели и содержание жизненного цикла;
- Life Cycle Architecture (LCA) - архитектура жизненного цикла; здесь же возможно говорить о готовности концептуальной архитектуры целевой программной системы;
- Initial Operational Capability (IOC) - первая версия создаваемого продукта, пригодная для опытной эксплуатации;

– Final Operational Capability (FOC) - готовый продукт, развернутый (установленный и настроенный) для реальной эксплуатации.

Таким образом, мы приходим к возможному современному взгляду (см., например, представление спиральной модели в на *итеративный и инкрементальный* - эволюционный жизненный цикл в форме спиральной модели, изображенной на рисунке 3.10.

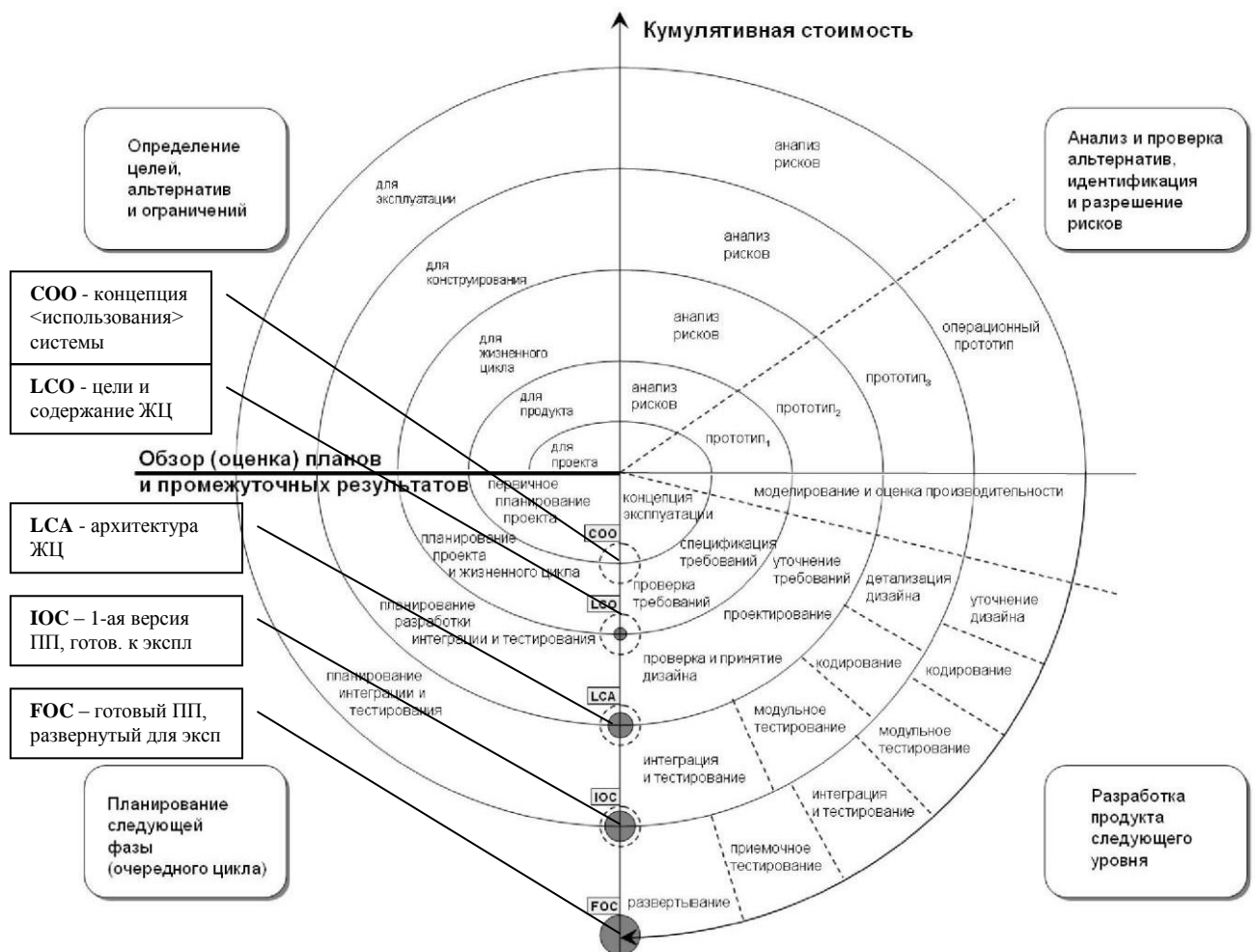


Рисунок 3.10 - Обновленная спиральная модель с контрольными точками проекта

**Основные недостатки спиральной модели связаны с ее сложностью:**

- Сложность анализа и оценки рисков при выборе вариантов;
- Сложность поддержания версий продукта (хранение версий, возврат к ранним версиям, комбинация версий);
- Сложность оценки точки перехода на следующий цикл;
- Бесконечность модели – на каждом витке заказчик может выдвигать новые требования, которые приводят к необходимости следующего цикла разработки.



### *V-образная модель*

V-образная модель была создана как итерационная разновидность каскадной модели. Целями итераций в этой модели является обеспечение процесса тестирования. Тестирование продукта обсуждается, проектируется и планируется на ранних этапах жизненного цикла разработки. План испытания приемки заказчиком разрабатывается на этапе планирования, а компоновочного испытания системы - на фазах анализа, разработки проекта и т.д. Этот процесс разработки планов испытания обозначен пунктирной линией между прямоугольниками V-образной модели. Помимо планов, на ранних этапах разрабатываются также и тесты, которые будут выполняться при завершении параллельных этапов. На рисунке 3.11 приведена V-образная модель разработки ЖЦ ПП.

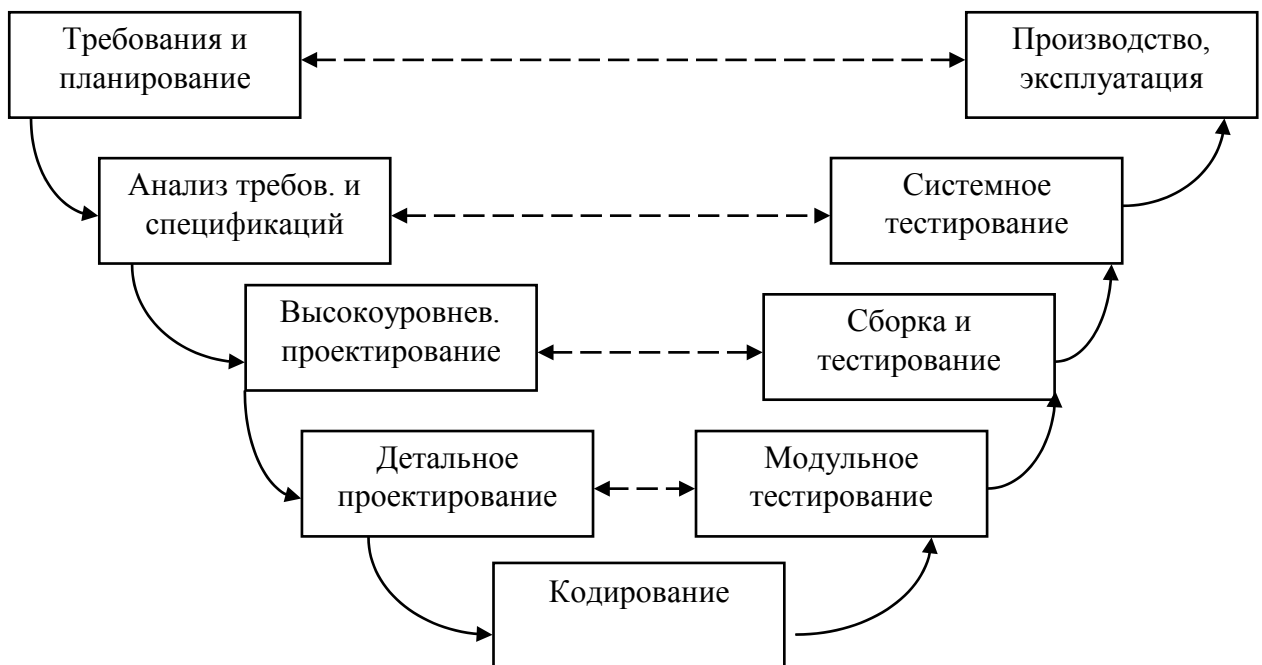


Рисунок 3.11 – V-образная модель ЖЦ ПП

### *Инкрементная (пошаговая) модель*

Инкрементная (пошаговая) модель представляет собой процесс поэтапной реализации всей системы и поэтапного наращивания (приращения) функциональных возможностей. На первом шаге необходим полный заранее сформулированный набор требований, которые делятся по некоторому признаку на части. Далее выбирается первая группа требований и выполняется полный проход по каскадной модели. После того как первый вариант системы, выполняющий первую группу требований, сдан заказчику (см.

рисунок 3.12), разработчики переходят к следующему шагу (второму инкременту) по разработке варианта, выполняющего вторую группу требований т.д.

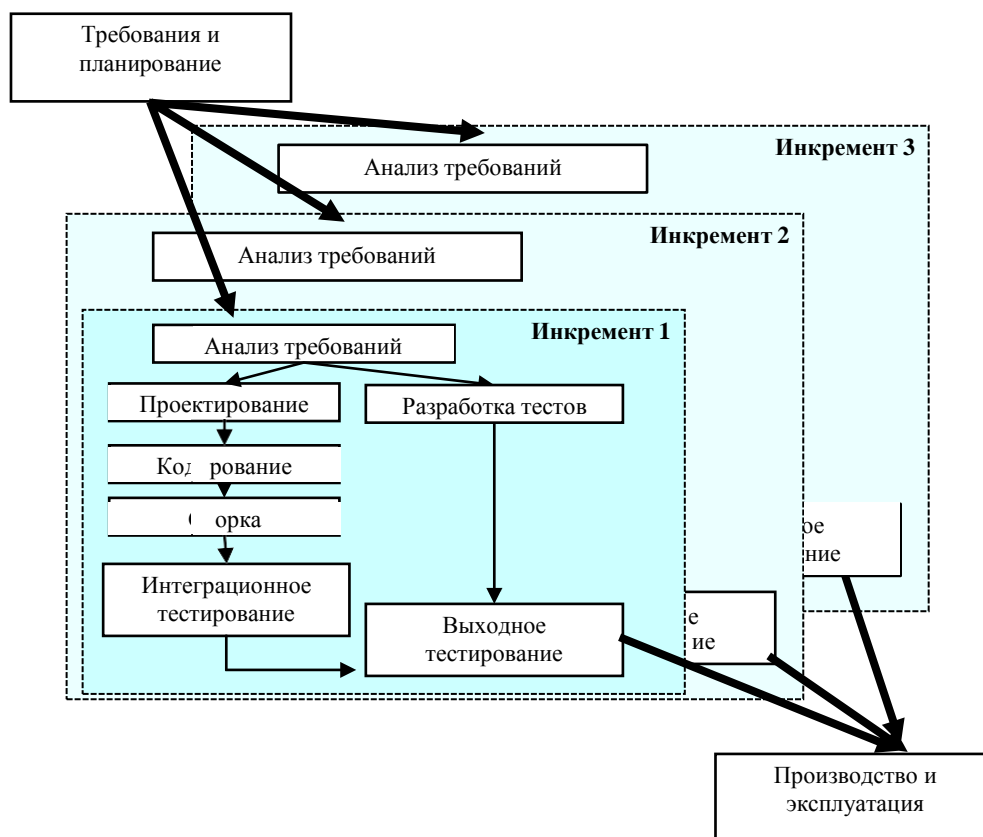


Рисунок 3.12 – Инкрементная модель ЖЦ ПС

Особенностью инкрементной модели является разработка приемочных тестов на этапе анализа требований, что упрощает приемку варианта заказчиком и устанавливает четкие цели разработки очередного варианта системы. Инкрементная модель особенно эффективна в случае, когда задача разбивается на несколько относительно независимых подзадач (разработка подсистем «Зарплата», «Бухгалтерия», «Склад», «Поставщики») в рамках единого проекта, для внутренней итерации можно использовать не только каскадную, но и другие типы моделей.

### ***Модель быстрого прототипирования***

Модель быстрого прототипирования предназначена для быстрого создания прототипов ПС с целью уточнения требований заказчика и поэтапного развития системы в конечный продукт. Скорость (высокая производительность) выполнения проекта обеспечивается планированием разработки прототипов и участием заказчика в процессе разработки.

Начало жизненного цикла разработки помещено в центре эллипса (рисунок 3.13). Совместно с пользователем разрабатывается предварительный план проекта на основе его

предварительных требований и формируется документ, описывающий в общих чертах примерные графики и результирующие данные.

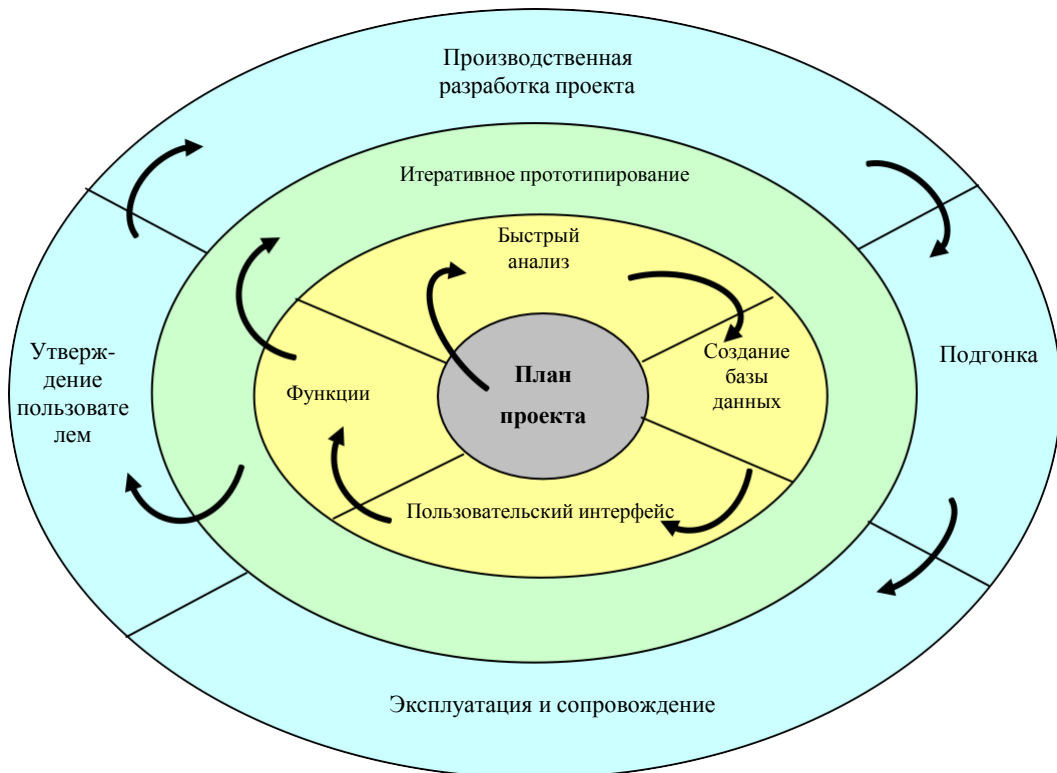


Рисунок 3.13 – Модель быстрого прототипирования

После этого переходят к созданию исходного прототипа на основе быстрого анализа, проекта базы данных, пользовательского интерфейса и некоторых функций. Затем начинается итерационный цикл быстрого прототипирования: разработчик проекта демонстрирует очередной прототип, пользователь (заказчик) оценивает его функционирование, совместно определяются проблемы и пути их преодоления для перехода к следующему прототипу. Этот процесс продолжается до тех пор, пока пользователь не согласится, что очередной прототип в точности отображает все его требования.

Получив одобрение пользователя, быстрый прототип преобразуют в детальный проект, настраивают на производственное использование, и он становится полностью действующей системой.

При разработке производственной версии программы, может понадобиться более высокий уровень функциональных возможностей, могут измениться системные ресурсы, необходимые для обеспечения полной рабочей нагрузки, или появятся ограничения во времени. После этого следуют тестирование в предельных режимах, определение измерительных критериев и настройка, а затем, как обычно, функциональное сопровождение.

### ***Модели жизненного цикла MSF, RUP, XP***

В настоящее время широкое применение получают так называемые *промышленные технологии создания программного продукта*. Эти технологии были разработаны фирмами, накопившими большой опыт создания ПО. Технологии представлены описаниями принципов, методов, применяемых процессов и операций. Такие технологии, как правило, поддерживаются набором CASE-средств, охватывают все этапы жизненного цикла продукта и успешно применяются для решения практических задач.

Рассмотрим особенности моделей жизненного цикла трех наиболее известных промышленных технологий:

- ***Microsoft Solution Framework (MSF)*** – разработка фирмы Microsoft, предназначенная для решения широкого круга задач. Технология масштабируема, т.е. настраиваема на решение задач любой сложности коллективом любой численности.
- ***Rational Unified Process (RUP)*** – разработка фирмы Rational, долгое время успешно занимавшейся созданием CASE-средств, применяемых на различных этапах жизненного цикла продукта от анализа до тестирования и документирования. Аналогично MSF, RUP универсальна, масштабируема и настраиваема на применение в конкретных условиях.
- ***Extreme Programming (XP)*** – активно развивающаяся в последнее время технология, предназначенная для решения относительно небольших задач, относительно небольшими коллективами профессиональных разработчиков в условиях жестко ограниченного времени.

Каждая из этих технологий имеет свои особенности организации модели жизненного цикла создания продукта.

#### **Модель Microsoft Solution Framework (MSF)**

Одна из особенностей технологии MSF состоит в том, что *она ориентирована не просто на создание ПП, удовлетворяющего перечисленным требованиям, а на поиск решения проблем, стоящих перед заказчиком*. Различие состоит в том, что перечисляемые заказчиком требования являются проявлениями некоторых более глубоких проблем и неточность, неполнота, изменение требований в процессе разработки – следствие недопонимания проблем. Поэтому, в технологии MSF большое внимание уделяется анализу проблем заказчика и разработке вариантов системы для поиска решения этих проблем [31].

Модель жизненного цикла MSF является некоторым гибридом каскадной и спиральной моделей, сочетая простоту управления каскадной модели с гибкостью спиральной. Схема модели жизненного цикла MSF (модели процессов) представлена на слайде.

Модель жизненного цикла MSF ориентирована на **«вехи»** (milestones) – ключевые точки проекта, характеризующие достижение какого-либо существенного результата. Этот результат может быть оценен и проанализирован, что подразумевает ответ на вопрос: «А достигли ли мы целей, поставленных на этом шаге?». В модели предусматривается наличие основных вех (завершение главных фаз модели) и промежуточных, отражающих внутренние этапы главных фаз (см. рисунок 3.12).

*Основными фазами модели MSF являются:*

- **Создание общей картины приложения (*Envisioning*)**. На этом этапе решаются следующие основные задачи: оценка существующей ситуации; определение состава команды, структуры проекта, бизнес-целей, требований и профилей пользователей; разработка концепции решения и оценка риска. Устанавливаются две промежуточные вехи: «Организован костяк команды» и «Создана общая картина решения».
- **Планирование (*Planning*)**. Включает планирование и проектирование продукта. На основе анализа требований разрабатывается проект и основные архитектурные решения, функциональные спецификации системы, планы и календарные графики, среды разработки, тестирования и пилотной эксплуатации. Этап состоит из трех стадий: концептуальное, логическое и физическое проектирование. На стадии концептуального проектирования задача рассматривается с точки зрения пользовательских и бизнес-требований и заканчивается определением набора сценариев использования системы. При логическом проектировании задача рассматривается с точки зрения проектной команды, решение представляется в виде набора сервисов. И уже на стадии физического проектирования задача рассматривается с точки зрения программистов, уточняются используемые технологии и интерфейсы.
- **Разработка (*Developing*)**. Создается вариант решения проблемы, в виде кода и документации очередного прототипа, включая спецификации и сценарии тестирования. Основная веха этапа – «Окончательное утверждение области действия проекта». Продукт готов к внешнему тестированию и стабилизации. Кроме того, заказчики, пользователи, сотрудники службы поддержки и

сопровождения, а также ключевые участники проекта могут предварительно оценить продукт и указать все недостатки, которые нужно устранить до его поставки.

- **Стабилизация (*Stabilizing*)**. Подготовка к выпуску окончательной версии продукта, доводка его до заданного уровня качества. Здесь выполняется комплекс работ по тестированию (обнаружение и устранение дефектов), проверяется сценарий развертывания продукта. Когда решение становится достаточно устойчивым, проводится его пилотная эксплуатация в тестовой среде с привлечением пользователей и применением реальных сценариев работы.
- **Развертывание (*Deploying*)**. Выполняется установка решения и необходимых компонентов окружения, проводится его стабилизация в промышленных условиях и передача проекта в руки группы сопровождения. Кроме того, анализируется проект в целом на предмет уровня удовлетворенности заказчика.



Рисунок 3.14 – Модель ЖЦ MSF

В *точке конвергенции* (bug convergence) становится заметен существенный прогресс в устранении ошибок, то есть скорость устранения ошибок начинает превосходить скорость их обнаружения (рисунок 3.15).

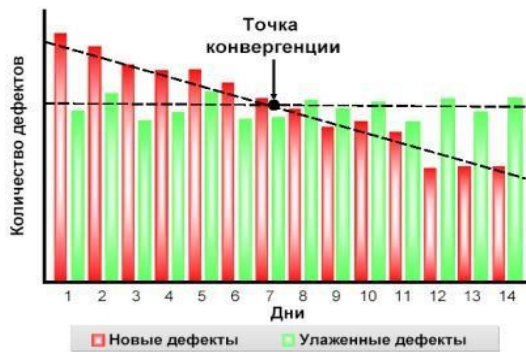


Рисунок 3.15 – Точка конвергенции

проектной группе возможность понять, что процесс тестирования близится к концу.

*Точка достижения нуля* (zero-bug bounce) - это момент, когда впервые все выявленные ошибки оказываются устраненными. Вслед за ней пики количества активных

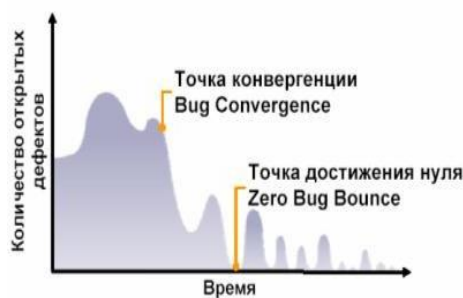


Рисунок 3.16 – Точка достижения нуля

ошибок должны становиться все меньше, вплоть до полного угасания в момент, когда решение уже достаточно стабильно для выпуска первой версии кандидата (рисунок 3.16).

Поскольку количество найденных, но не устраненных ошибок может колебаться даже после того, как оно начало убывать, *конвергенция может рассматриваться скорее как тенденция*, нежели как фиксированный момент во времени. Вслед за этой вехой количество активных ошибок должно продолжать убывать, вплоть до точки достижения нуля. Точка конвергенции дает

Существенную роль играет тщательная приоритезация ошибок, поскольку устранение всякой из них содержит риск внесения новых ошибок. Точка достижения нуля ясно показывает, что проектная группа приближается к созданию

стабильной версии кандидата (release candidate).

### Модель процессов MSF for Agile Software Development

Модель процессов MSF представляет общую методологию разработки и внедрения ИТ- решений. Особенность этой модели состоит в том, что благодаря своей гибкости и отсутствию жестко навязываемых процедур она может быть применена при разработке весьма широкого круга ИТ проектов. Эта модель сочетает в себе свойства двух стандартных производственных моделей: каскадной (*waterfall* ) и спиральной (*spiral* ) (см. п.3.3.1 и 3.3.3).

Процесс MSF ориентирован на «вехи» (*milestones* ) – ключевые точки проекта, характеризующие достижение в его рамках какого-либо существенного (промежуточного либо конечного) результата. Модель процессов MSF учитывает постоянные изменения проектных требований. Она исходит из того, что разработка решения должна состоять из

коротких циклов, создающих поступательное движение от простейших версий решения к его окончательному виду. Рассмотрим основные принципы модели процессов.

*Взаимодействуйте с «заказчиками»*

MSF настаивает на непрерывном взаимодействии с заказчиком в ходе всей работы над проектом. Удовлетворенный заказчик – главный приоритет проектной группы. Понимание бизнес-отдач заказчика от проекта, потребностей его будущих пользователей требует максимальной полной вовлеченности заказчика в процесс создания решения.

*Поощряйте свободный обмен информацией в проекте*

Модель процессов MSF считает очень важным открытый обмен информацией как внутри команды, так и с ключевыми заинтересованными лицами. Свободный обмен информацией не только сокращает риск возникновения недоразумений, недопонимания и неоправданных затрат, но и обеспечивает максимальный вклад всех участников проектной группы в снижение существующей в проекте неопределенности. Естественно речь здесь не идет об информации финансовой или имеющей статус коммерческой или иной тайны.

*Создавайте «единое видение проекта»*

Успех коллективной работы над проектом немыслим без наличия у членов проектной группы и заказчика единого видения (*shared vision*), т.е. четкого, и, самое главное, одинакового, понимания целей и задач проекта. Изначально понимание того, что должно быть достигнуто в ходе работы над проектом, у заказчика может (и нередко) не совпадать с пониманием проектной группы. Лишь наличие единого видения способно внести ясность и обеспечить движение всех заинтересованных в проекте сторон к общей цели.

Формирование единого видения и последующее следование ему являются столь важными, что модель процессов MSF выделяет для этой цели специальную фазу (фаза «Выработка концепции»), которая заканчивается соответствующей вехой.

*Следите за качеством продукта*

MSF настаивает на том, что каждый участник проектной группы должен ощущать ответственность за качество разрабатываемого решения. Она не может быть делегирована одним членом команды другому или же от одной ролевой группы другой. Несмотря на наличие в команде ролевых групп “Удовлетворение потребителя” и “Тестирование”, прямая задача которых – следить за качеством решения и повышать его, все остальные ролевые группы также постоянно должны иметь в виду нужды конечного пользователя.

*Проявляйте гибкость – будьте готовы к изменениям*



MSF основывается на принципе непрерывной изменяемости условий проекта при неизменной эффективности управленческой деятельности. Проектная группа должна быть готова к переменам, и методология MSF предоставляет эффективный инструментарий для адекватной и своевременной реакции на изменения в проектной среде.

*Ставьте «вехи»*

Каждая итерация, каждая фаза процесса создания решения должна заканчиваться некоторым зримым результатом, некоторой вехой (*milestone*). Наличие вех позволяет проектной группе и заказчику видеть движение проекта вперед.

*Будьте готовы к внедрению сегодня*

Работа проектной группы в идеале должна быть построена так, чтобы при возникновении такой потребности у заказчика текущее состояние разрабатываемого решения могло быть немедленно внедрено (естественно с той функциональностью, которая в данный момент реализована). Это означает, что итерации работы над проектом должны быть максимально короткими, и в каждый момент времени должна существовать текущая работоспособная версия решения. Такой подход дает возможность раннего обнаружения рисков, ошибок, пропущенных или недопонятых требований, дает возможность своевременно получать обратную реакцию от заказчика, а, значит, сокращает затраты.

*Управление компромиссами*

В силу свойственной IT-проектам неопределенности и рискованности, одним из ключевых факторов их успеха являются эффективные компромиссные решения (*trade-offs*).

Хорошо известна взаимозависимость между ресурсами проекта (людскими и финансовыми), его календарным графиком (временем) и реализуемыми возможностями (рамками). Эти три переменные образуют *треугольник компромиссов* (*tradeoff triangle*), приведенный на рисунке 3.17 [19].



Рисунок 3.17 - Треугольник компромиссов

После достижения равновесия в этом треугольнике изменение на любой из его сторон для поддержания баланса требует модификаций на другой (двух других) сторонах и/или на изначально измененной стороне.

Нахождение верного баланса между ресурсами, временем разработки и возможностями — ключевой момент в построении решения, должным образом отвечающего нуждам заказчика.

### Матрица компромиссов проекта

Другое полезное средство управления проектными компромиссами — матрица компромиссов проекта (*project tradeoff matrix*). Она отражает достигнутое на ранних этапах проекта соглашение между проектной группой и заказчиком о выборе приоритетов в возможных в будущем компромиссных решениях. Возможный вариант такой матрицы представлен на рисунке 3.18.

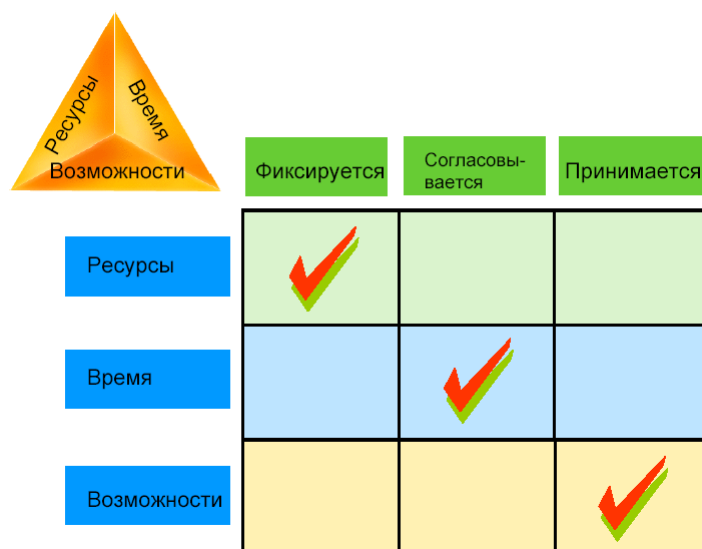


Рисунок 3.18 - Матрица компромиссов (возможный вариант)

Матрица компромиссов помогает обозначить проектное ограничение, воздействие на которое практически невозможно (столбец «Фиксируется»), фактор, являющийся в проекте приоритетным (столбец «Согласовывается»), и третий параметр, значение которого должно быть принято в соответствии с установленными значениями первых двух величин (столбец «Принимается»).

### Модель Rational Unified Process (RUP)

Модель жизненного цикла RUP является довольно сложной, детально проработанной итеративно-инкрементной моделью с элементами каскадной модели [32]. В модели RUP выделяются 4 основные фазы, 9 видов деятельности (процессов). Кроме того, в модели описывается ряд практик, которые следует применять или

руководствоваться для успешного выполнения проекта. RUP ориентирован на поэтапное моделирование создаваемого продукта с помощью языка UML (рисунок 3.19).

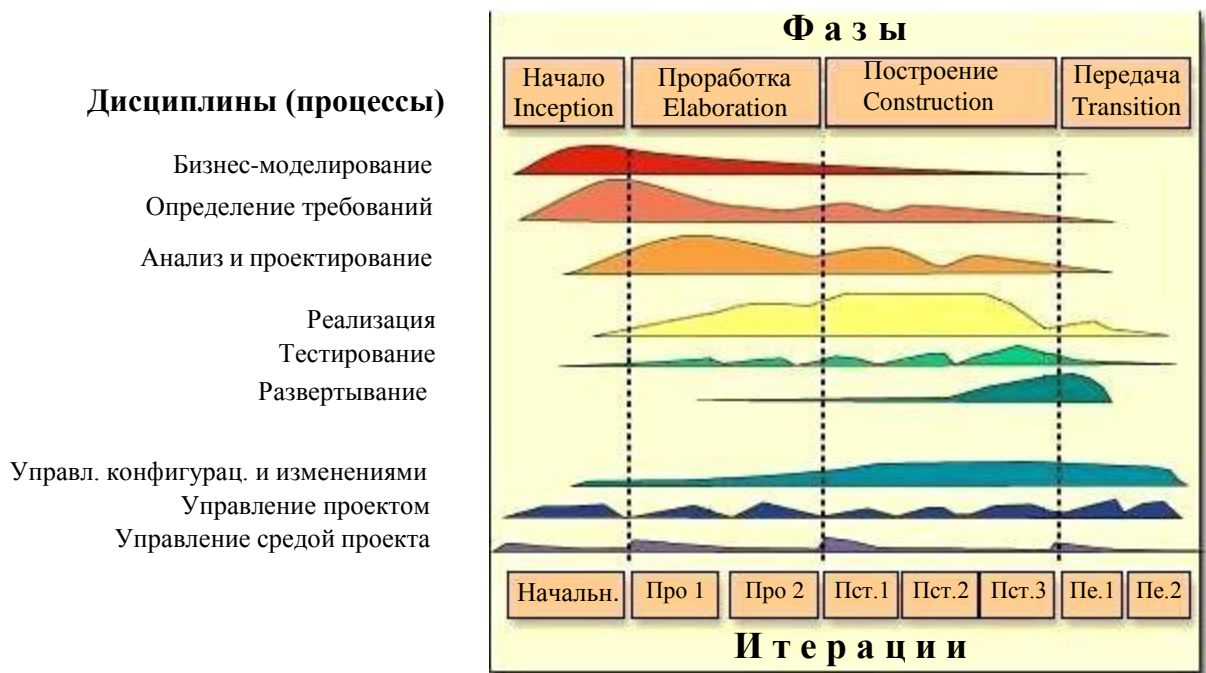


Рисунок 3.19 – Модель RUP

Основными фазами RUP являются:

- **Фаза начала проекта (Inception)**. Определяются основные цели проекта, бюджет проекта, основные средства его выполнения - технологии, инструменты, ключевой персонал, составляются предварительные планы проекта. Основная цель этой фазы - достичь компромисса между всеми заинтересованными лицами относительно задач проекта.
- **Фаза проработки (Elaboration)**. Основная цель этой фазы - на базе основных, наиболее существенных требований разработать стабильную базовую архитектуру продукта, которая позволяет решать поставленные перед системой задачи и в дальнейшем используется как основа разработки системы.
- **Фаза построения (Construction)**. Основная цель этой фазы - детальное прояснение требований и разработка системы, удовлетворяющей им, на основе спроектированной ранее архитектуры.
- **Фаза передачи (Transition)**. Цель фазы - сделать систему полностью доступной конечным пользователям. Здесь происходит окончательное развертывание системы в ее рабочей среде, подгонка мелких деталей под нужды пользователей.

В рамках каждой фазы возможно проведение нескольких итераций, количество которых определяется сложностью выполняемого проекта.

*Деятельности* (основные процессы) RUP делятся на пять рабочих и четыре поддерживающие. **К рабочим деятельности относятся:**

- *Моделирование предметной области* (бизнес-моделирование, Business Modeling). Цели этой деятельности - понять бизнес-контекст, в котором должна будет работать система (и убедиться, что все заинтересованные лица понимают его одинаково), понять возможные проблемы, оценить возможные их решения и их последствия для бизнеса организации, в которой будет работать система.
- *Определение требований* (Requirements). Цели - понять, что должна делать система, определить границы системы и основу для планирования проекта и оценок ресурсозатрат в нем.
- *Анализ и проектирование* (Analysis and Design). Выработка архитектуры системы на основе ключевых требований, создание проектной модели, представленной в виде диаграмм UML, описывающих продукт с различных точек зрения.
- *Реализация* (Implementation). Разработка исходного кода, компонент системы, тестирование и интегрирование компонент.
- *Тестирование* (Test). Общая оценка дефектов продукта, его качество в целом; оценка степени соответствия исходным требованиям.

**Поддерживающими деятельности являются:**

- *Развертывание* (Deployment). Цели - развернуть систему в ее рабочем окружении и оценить ее работоспособность.
- *Управление конфигурациями и изменениями* (Configuration and Change Management). Определение элементов, подлежащих хранению и правил построения из них согласованных конфигураций, поддержание целостности текущего состояния системы, проверка согласованности вносимых изменений.
- *Управление проектом* (Project Management). Включает планирование, управление персоналом, обеспечения связей с другими заинтересованными лицами, управление рисками, отслеживание текущего состояния проекта.
- *Управление средой проекта* (Environment). Настройка процесса под конкретный проект, выбор и смена технологий и инструментов, используемых в проекте.

## Модель Extreme Programming (XP)

Экстремальное программирование является примером так называемого метода «живой» разработки ([Agile Development Method](#)). В группу «живых» входят, помимо экстремального программирования, входит еще ряд методов, о чем подробнее можно прочитать [33].

Модель жизненного цикла XP является итерационно-инкрементной моделью быстрого создания (и модификации) прототипов продукта, удовлетворяющих очередному требованию (user story). Особенности этой модели представлены на рисунке 3.20.

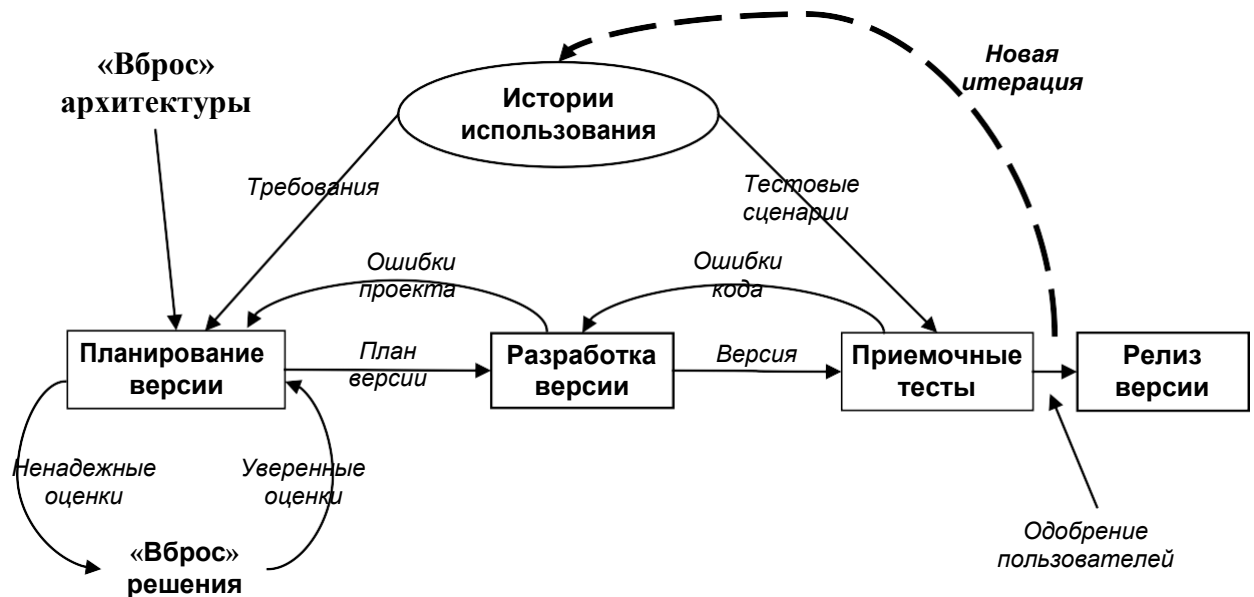


Рисунок 3.20 – Модель экстремального программирования  
(для одной версии – итерации системы)

Основными фазами модели можно считать:

- **«Вброс» архитектуры** – начальный этап проекта, на котором создается видение продукта, принимаются основные решения по архитектуре и применяемым технологиям. Результатом начального этапа является *метафора (metaphor)* системы, которая в достаточно простом и понятном команде виде должна описывать основной механизм работы системы.
- **Истории использования (User Story)** – этап сбора требований, записываемых на специальных карточках в виде *сценариев выполнения отдельных функций*. Истории использования являются требованиями для планирования очередной версии и одновременной разработки приемочных тестов (Acceptance tests) для ее проверки.
- **Планирование версии (релиза)**. Проводится на собрании с участием заказчика путем выбора User Stories, которые войдут в следующую версию. Одновременно

принимаются решения, связанные с реализацией версии. Цель планирования - получение оценок того, что и как можно сделать за 1-3 недели создания следующей версии продукта.

- **Разработка** проводится в соответствии с планом и включает только те функции, которые были отобраны на этапе планирования.
- **Тестирование** проводится с участием заказчика, который участвует в составлении тестов.
- **Выпуск релиза** – разработанная версия передается заказчику для использования или бета-тестирования.

По завершению цикла делается переход на следующую итерацию разработки.

Особенности модели жизненного цикла ХР проясняют следующие **принципы этого метода**. Прежде всего, это принципы «живой» разработки ПО, зафиксированные в манифесте «живой» разработки:

- Люди их общение более важны, чем процессы и инструменты.
- Работающая программа более важна, чем исчерпывающая документация.
- Сотрудничество с заказчиком более важно, чем обсуждение деталей контракта.
- Отработка изменений более важна, чем следование планам.

Кроме того, в ХР есть несколько *правил (техник)*, характеризующих особенности модели его жизненного цикла:

- *Живое планирование (planning game)* - как можно быстрее определить объем работ, который нужно сделать до следующей версии ПО. Решение принимается на основе, в первую очередь, бизнес-приоритетов заказчика и, во-вторую, технических оценок. Планы изменяются, как только они начинают расходиться с действительностью или пожеланиями заказчика.
- *Частая смена версий (small releases)* - первая работающая версия должна появиться как можно быстрее, и тут же должна начать использоваться. Следующие версии подготавливаются через достаточно короткие промежутки времени.
- *Простые проектные решения (simple design)* - в каждый момент времени система должна быть сконструирована так просто, насколько это возможно. Новые функции добавляются только после ясной просьбы об этом. Вся лишняя сложность удаляется, как только обнаруживается.
- *Разработка на основе тестирования (test-driven development)* - сначала пишутся тесты, потом реализуются модули так, чтобы тесты срабатывали. Заказчики

- заранее пишут тесты, демонстрирующие основные возможности системы, чтобы можно было увидеть, что система действительно заработала.
- *Постоянная переработка (refactoring)* - системы для устранения излишней сложности, увеличения понятности кода, повышения его гибкости. При этом предпочтение отдается более элегантным и гибким решениям, по сравнению с просто дающими нужный результат.
  - *Программирование парами (pair programming)* - весь код пишется двумя программистами на одном компьютере, что повышает его качество (отсутствие ошибок, понятность, читаемость, ...).
  - *Постоянная интеграция (continuous integration)* - система собирается и проходит интеграционное тестирование как можно чаще, по несколько раз в день, каждый раз, когда пара программистов оканчивает реализацию очередной функции.
  - *40-часовая рабочая неделя* - сверхурочная работа рассматривается как признак больших проблем в проекте. Не допускается сверхурочная работа 2 недели подряд - это истощает программистов и делает их работу значительно менее продуктивной.

## 4 УПРАВЛЕНИЕ КОМАНДОЙ ПРОЕКТА

*Успех проекта напрямую связан с используемыми талантами, и, что более важно, способом, в соответствии с которым руководство использует эти таланты в проекте.*

Джон Макдоналд

В спиральной модели удалось более четко и естественно определить контрольные точки проекта, в определенной степени, подчеркнув эволюционную природу жизненного цикла. Теперь же пора взглянуть на жизненный цикл в контексте методологий, не просто детализирующих ту или иную модель, но добавляющих к ним *ключевой элемент - людей*. Роли, как представление различных функциональных групп работ, связывает создание, модификацию и использование активов проектов с конкретными участниками проектных команд. В совокупности с процессами и активами (артефактами) они позволяют нам создать целостную и подробную картину жизненного цикла (рисунок 4.1).



Рисунок 4.1 – Модель производственной архитектуры

Управление программным проектом включает решение трех основных задач:

1. Подбор и управление командой (персонала).
2. Выбор процесса.
3. Выбор инструментальных средств.

Хотя все три задачи одинаково важны для успеха проекта, ведущую роль играет правильный подбор и управление командой. Успех проекта во многом зависит от того, насколько состав участников проекта сможет быть преобразован в команду единомышленников, насколько эта команда будет активной и инициативной с одной стороны и управляемой с другой. Из множества вопросов управления командой проекта мы рассмотрим три:

- Ролевая модель команды;
- Модели организации команд;
- Общение в команде.



### *Ролевая модель команды*

Состав команды определяется опытом и уровнем коллектива, особенностями проекта, применяемыми технологиями и уровнем этих технологий. На рисунке 4.1 представлен один из вариантов состава команды, описанный в [34]. Выделенные позиции не обязательно представлены конкретными людьми. Это список основных функциональных ролей в команде (ролевая модель команды). В малых командах роли могут совмещаться. В больших – выделяться группы или отделы (отдел проектирования, отдел тестирования, отдел контроля качества, отдел подготовки документации, ...).

Состав команды определяется также типом выполняемых работ: под заказ или коробочное производство (продукт на рынок). *Инженерный психолог и инженер по маркетингу* нужны в последнем случае.

В представленной модели выделены следующие основные роли (рисунок 4.2):

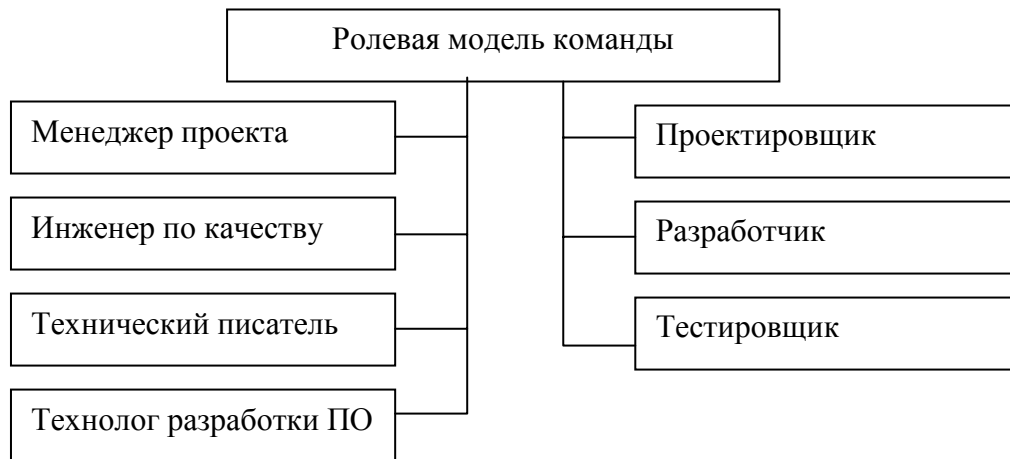


Рисунок 4.1 – Ролевая модель команды

- **Менеджер проекта** - главное действующее лицо, обладающее знаниями и навыками, необходимыми для успешного управления проектом. Его основные функции:
  - ✓ Подбор и управление кадрами;
  - ✓ Подготовка и исполнение плана проекта;
  - ✓ Руководство командой;
  - ✓ Обеспечение связи между подразделениями;
  - ✓ Обеспечение готовности продукта.
- **Проектировщик** - это функция проектирования архитектуры высокого уровня и контроля ее выполнения. В небольших командах функция распределяется между менеджером и разработчиками. В больших проектах это может быть целый отдел. Основными функциями проектирования являются:

- ✓ Анализ требований.
  - ✓ Разработка архитектуры и основных интерфейсов.
  - ✓ Участие в планировании проекта.
  - ✓ Контроль выполнения проекта.
  - ✓ Участие в подборе кадров.
- **Разработчик** – роль, ответственная за непосредственное создание конечного продукта. Помимо собственно программирования (кодирования) в его функции входит:
- ✓ Контроль архитектурных и технических спецификаций продукта.
  - ✓ Подбор технологических инструментов и стандартов.
  - ✓ Диагностика и разрешение всех технических проблем.
  - ✓ Контроль за работой разработчиков документации, тестирования, технологов.
  - ✓ Мониторинг состояния продукта (ведение списка обнаруженных ошибок).
  - ✓ Подбор инструментов разработки, метрик и стандартов. Контроль их использования.
- **Тестировщик** – роль, ответственная за удовлетворение требований к продукту (функциональных и нефункциональных). В функции тестировщика входит:
- ✓ Составление плана тестирования. План тестирования составляет один из элементов проекта и составляется до начала реализации (разработки) проекта. Время, отводимое в плане на тестирование может быть сопоставимо с временем разработки.
  - ✓ Контроль выполнения плана. Важнейшая функция контроля – поддержка целостности базы данных зарегистрированных ошибок. В этой базе регистрируется:
    - кто, когда и где обнаружил, описание ошибки, описание состояния среды;
    - статус ошибки: приоритет, кто разрешает
    - состояние ошибки: висит, в разработке, разрешена, проблемы
- Эта база должна быть доступна всем, т.к. в тестировании принимают участие все члены команды.
- ✓ Разработка тестов. Самая трудоемкая часть в работе тестировщика. Тестирование должно обеспечить полную проверку функциональности при всех режимах работы продукта.

- ✓ Автоматизация тестирования включает автоматизацию составления тестов, автоматизацию пропуска тестов и автоматизацию обработки результатов тестирования. В виду важности автоматизации тестирования, иногда вводят нового участника – инженера по автоматизации.
  - ✓ Выбор инструментов, метрик, стандартов для организации процесса тестирования.
  - ✓ Организация Бета тестирования - тестирования почти готового продукта внешними тестерами (пользователями). Эту важную процедуру надо продумать и организовать в случае разработки коробочного продукта.
- **Инженер по качеству.** В современном представлении рассматривается три аспекта (уровня) качества:
- ✓ –качество конечного продукта – обеспечивается тестированием,
  - ✓ –качество процесса разработки (тезис: для повышения качества продукта надо повысит качество процесса разработки),
  - ✓ –качество (уровень) организации (тезис: для повышения качества процесса надо повысить качество организации работ).

В некоторых случаях функции инженера по качеству возлагаются на тестировщика. На самом деле они шире – два следующих уровня качества. Здесь приведены функции, отличные от функции тестировщика:

- ✓ Составление плана качества. План качества включает все мероприятия по повышению качества (на всех уровнях). Имеет долговременный характер. План тестирования – его оперативная составляющая.
- ✓ Описание процессов. Описание процессов является их формализацией. При описании вводятся метрики процесса, влияющие на качество продукта.
- ✓ Оценка процессов включает регистрацию хода выполнения процессов и оценку значений установленных метрик процессов. Выявление «слабых» мест и выработка рекомендаций по улучшению процессов.
- ✓ Улучшение процессов - переопределение процесса, автоматизация части работ, обучение персонала.

Повышение качества процессов требует участия всех действующих лиц. Принятое решение должно быть обосновано, всем понятно и всеми принято. При повышении качества организации работа по улучшению процессов проводится по определенной схеме. На каждом шаге повышения уровня организации работ выделяются ключевые процессы и выполняются работы по улучшению этих процессов.

- **Технический писатель** или разработчик пользовательской (и иной) документации как части программного продукта. Функциями технического писателя являются:
  - ✓ Разработка плана документирования, который включает состав, сроки подготовки и порядок тестирования документов.
  - ✓ Выбор и разработка стандартов и шаблонов подготовки документов
  - ✓ Выбор средств автоматизации документирования
  - ✓ Разработка документации
  - ✓ Организация тестирования документации. Участие в тестировании продукта. Технический писатель все время работает с продуктом (его готовыми версиями) и выступая от имени пользователя видит все недочеты и несоответствия.
- **Технолог разработки ПО** обеспечивает выполнение следующих задач:
  - ✓ Поддержка модели ЖЦ - создание служб и структур по поддержке работоспособности принятой модели ЖЦ ПО. В поддержке модели ЖЦ принимают участие все. Но контроль возложен на технолога.
  - ✓ Создание и сопровождение среды сборки продукта. Функция особенно важна на завершающих этапах разработки или при использовании модели прототипирования. В такой ситуации сборка будет проводиться достаточно часто (в некоторых случаях - ежедневно). Среда сборки должна быть подготовлена заранее, сборка должна проводиться быстро и без сбоев. С учетом сборки версий это не простая задача.
  - ✓ Создание и сопровождение процедуры установки с тем, чтобы каждая сборка устанавливалась автоматически с учетом версии и конфигураций сред.
  - ✓ Управление исходными текстами - сопровождение и администрирование системы управления версиями исходных текстов.

Подводя итог, следует отметить, что ролевые модели проектных команд могут быть самыми разнообразными.

*Модели организации команд*

*«...методологи разрабатывают сложные системы, в которых есть весьма изменчивые и нелинейные компоненты – люди»*

Алистэр Коуберн

Как организовать работу команды? Команды из 10 человек и команды из 500 человек? Есть ли различия и в чем они состоят? Надо ли организовывать работу по жесткой технологии или надо предоставить свободу действий? Можно ли найти методологию (технологию) выполнения проекта, обеспечивающую успех?

Алистер Коубен [35] – специалист в области технологий выполнения ИТ проектов - приводит данные 23 проектов различной степени сложности, выполнявшихся по различным технологиям и имеющие различные результаты. Пытаясь проанализировать результаты применения различных технологий в тех или иных условиях, он приходит к выводу:

- Практически любую методологию можно с успехом применять в каком-нибудь проекте.
- Любая методология может привести к провалу проекта.

Главную причину он видит в том, прямо перед нами всегда находится нечто, чего мы не замечаем: люди. Именно человеческие качества обеспечивают успех тому или иному проекту, именно они являются фактором первостепенной важности, основываясь на котором надо строить прогнозы о проекте.

Исследованию вопросов человеческого фактора (Peopleware) уделяется достаточно много внимания, наиболее известными работами являются [36-39].

Проблемы человеческого фактора связаны с тем (проявляются в том), что участвующие в проекте **люди**:

- *Все разные* – по характеру, темпераменту, активности, целям – нет двух одинаковых людей.
- *Все похожие* – участие в проекте объединяет людей общностью целей, поиском путей достижения этих целей.
- *Различаются по типу*:
  - ✓ индивидуалисты - члены команды;
  - ✓ генераторы идей - исполнители;
  - ✓ ответственные – безответственные;

- *Постоянны и изменчивы* – люди, как правило, проявляют постоянство своих привычек и свойств характера, но при этом способны проявлять «противоположные» качества: индивидуалист – командные качества, исполнитель – генерировать идеи,
- *Многообразны* – надо понимать, что многообразие людей является основной гарантией выживания человечества вообще и возможности выполнять ИТ проекты в частности. Если бы все были индивидуалисты или все командники, все генераторы идей или все исполнители, то вряд ли удалось выполнить хотя бы один проект, а мир стал бы ужасен.

Как же управлять такими людьми? Рассмотрим основные модели проектных групп, которые применяются на практике

### *Административная модель (теория X)*

Это традиционный стиль управления, связанный с иерархической административно-командной моделью, которую используют военные организации. В основе лежит *теория X*, которая утверждает, что такой подход необходим, поскольку большинство людей по своей природе не любит работу и будет стремиться избежать ее, если у них есть такая возможность. Однако менеджеры должны принуждать, контролировать, направлять сотрудников и угрожать им, чтобы получить от них максимальную отдачу. *Девиз теории и модели: Люди делают только то, что вы контролируете.* Или в более мягком варианте: Люди делают то, что они не хотят делать, только если вы их контролируете. В конце концов, теория утверждает, что большинство людей предпочитают, чтобы им говорили, что следует делать и им не придется ничего решать самим.

*Характерные черты модели:*

- Властная пирамида – решения принимаются сверху-вниз;
- Четкое распределение ролей и обязанностей;
- Четкое распределение ответственности;
- Следование инструкциям, процедурам, технологиям;
- Роль менеджера: планирование, контроль, принятие основных решений.

*Преимущества модели:* ясность, простота, прогнозируемость. Модель хорошо сочетается с каскадной моделью жизненного цикла и применима в тех же случаях, что и каскадная модель. Модель эффективна в случае установившегося процесса.

*Недостатки модели* связаны с тем, что административная система стремится самосохранению (стабильности) и плохо восприимчива к изменению ситуации – новые типы проектов, применение новых технологий, оперативная реакция на изменение рынка.

Кроме того, в административной модели плохо уживаются индивидуалисты и генераторы идей.

Административная система (модель) – это тяжелый паровоз, идущий в «середине» и не поддающийся на «крайности» поиска новых путей и решений. Она воспринимает новые решения и технологии, но только проверенные, отработанные и стандартизированные. В этом ее сила, слабость и проявление принципа многообразия. Видимо, именно к ней в наибольшей степени применим термин «промышленное программирование».

### ***Модель хаоса (теория Y)***

В основе модели хаоса лежит *Теория Y*, которая является полной противоположностью Теории X. *Основной тезис: Работа — естественная и приятная деятельность и большинство людей*, на самом деле, очень ответственные и не уваливают от работы.

*Характерными чертами модели хаоса являются:*

- Отсутствие явно выраженных признаков власти;
- Роль менеджера – поставить задачу, обеспечить ресурсами, не мешать и следить, чтобы не мешали другие;
- Отсутствие инструкций и регламентированных процедур;
- Индивидуальная инициатива - решения по проблеме принимаются там, где проблема обнаружена;
- Процесс напоминает творческую игру участников на основе дружеской соревновательности.

Преимущества такой модели в том, что творческая инициатива участников ничем не связана и потенциал участников раскрывается в полной мере. Это бывает особенно эффективно в случае, когда для решения проблемы требуется поиск новых подходов, методов, идей и средств. Команда становится командой «прорыва», а работа проходит в форме игры, цель которой – поиск наилучшего результата. Процесс напоминает случайный поиск, когда идеи и решения рождаются при живом и как бы случайном обсуждении проблем в коридоре, столовой на пикнике. Собрать такую команду в рабочей комнате и устроить обсуждение по регламенту часто просто не удастся – это команда творческих индивидуалистов.

Недостатки модели связаны с тем, что при определенных условиях команда прорыва может стать командой провала. Причинами провала могут быть:

- Творческая соревновательность переходит в конкуренцию сначала идей, а потом - личностей.

- Процесс начинает преобладать над целью проекта – высказанные идеи не доводятся до конца и сменяются новыми идеями, преобладание получают «красивые» идеи, лежащие в стороне от основных целей проекта.
- Люди, способные к генерации идей, редко обладают терпением доведения идей до полной реализации.

Модель хаоса – это то, что нужно для освоения новых земель. Модель хаоса не противоречит административной модели – она ее дополняет и может эффективно с ней соседствовать (но в разных комнатах!). Многие мускулистые корпоративные бегемоты полагаются на исследовательские «отделы скунсов», откуда они черпают новые идеи, технологии и продукты.

#### 4.2.3. *Открытая архитектура (теория Z)*

Административная и хаотическая модели являются двумя «крайностями», между которыми находятся множество моделей, сочетающих преимущества «крайних» моделей. Одной из таких моделей является модель открытой архитектуры, основанная на Теории Z. Эта теория была сформулирована Уильямом Оучи на основе изучения опыта японского стиля управления (Theory Z: How American Business Can Meet the Japanese Challenge,» Perseus Publishing, 1981). Теория Z предполагает (но не декларирует) наличие внутреннего механизма управления, основанного на влиянии со стороны коллег и группы в целом. Дополнительное воздействие оказывают культурные нормы конкретной корпорации. *Основной принцип* модели можно сформулировать так: *«Работаем спокойно. Работаем вместе»*.

*Особенностями этой модели являются:*

- Адаптация к условиям работы – если делаем независимые модули, то расходимся и делаем, если нужна архитектура базы данных, то собираемся вместе и обсуждаем идеи.
- Коллективное обсуждение проблем, выработка консенсуса и принятие решения – не все могут согласиться, но принятое решение является коллективным и в силу этого – обязательным для всех.
- Распределенная ответственность – отвечают все, кто обсуждал, вырабатывал, принимал.
- Динамика состава рабочих групп в зависимости от текущих задач.
- Отсутствие специализации – участники меняются ролями и функциями и могут при необходимости заменить друг друга.



- Задача менеджера – активное (но рядовое, не руководящее) участие в процессе, контроль конструктивности обсуждений, обеспечение возможности активного участия всех.

Открытая архитектура является более гибкой, адаптируемой, настраиваемой на ситуацию. Она дает возможность проявить себя всем членам команды – в ней могут уживаться и индивидуалисты и коллективисты. Коллективное обсуждение высказанных идей позволяет оставлять только прагматичные идеи.

#### **4.2.4 Модель проектной группы MSF for Agile Software Development**

##### ***Основные принципы построения команды***

Рассмотрим, наверное, самую главную из отличительных черт MSF в сравнении с другими методологиями - модель проектной группы и *принципы*, положенные в основу построения команды.

Методология MSF считает, что успешная работа команды над проектом существенным образом зависит от ее структуры и распределения зон ответственности *ролевых групп* (более подробно о составе проектной группы далее) внутри команды. Построение команды в MSF соответствует ряду ключевых концепций (key concepts), часть которых кажутся самоочевидными (первые три), другие чем-то сродни «ноу-хау» (последние две):

- *Концентрация на нуждах заказчика* (customer-focused mindset) - главный приоритет любой хорошо работающей проектной группы. Означает обязательное понимание бизнес-задач заказчика и стремление к их решению со стороны команды. Не менее важным является активное участие заказчика в проектировании решения и получение его отзывов в ходе процесса разработки.
- *Нацеленность на конечный результат* (product mindset) - каждый участник проектной группы должен рассматривать собственную работу в качестве самостоятельного проекта или же вклада в какой-либо больший проект. Установка на конечный продукт означает, что получению конечного результата проекта уделяется больше внимания, чем процессу его достижения. Из этого не следует, что сам процесс может быть плох или непродуман - просто он существует для получения конечной цели, а не ради себя самого.
- *Установка на отсутствие дефектов* (zero-defect mindset) - это стремление к высочайшему уровню качества. Она означает, что цель команды - выполнение своей работы с максимально возможным качеством, в идеале таким образом, что если от команды потребуют поставить результат завтра, она будет способна

поставить что-то работающее. В успешной команде каждый сотрудник чувствует ответственность за качество продукта. Она не может быть делегирована одним членом команды другому или же от одной ролевой группы другой.

- «*Проектная группа - команда равных*» (*team of peers*). Концепция означает равноправное положение каждой из ролей в команде. Чтобы достичь успеха в рамках команды равных, каждый из ее членов, независимо от роли, должен нести ответственность за качество продукта, понимать интересы заказчика и сущность решаемой бизнес-задачи. В то же время, принятие решения методом консенсуса между ролями не тождественно принятию решения методом консенсуса между сотрудниками. Каждая ролевая группа требует определенной организационной иерархии для распределения работы и управления ее ресурсами.
- *Стремление к самосовершенствованию* (*willingness to learn*) - это приверженность идее неустанного саморазвития посредством накопления опыта и обмена знаниями. Оно позволяет членам проектной группы извлекать пользу из отрицательного опыта сделанных ошибок, равно как и воспроизводить успехи, используя проверенные методы работы других людей. По окончании основных фаз проекта и по завершению проекта в целом предполагается проведение открытых обсуждений его состояния и доброжелательный, но объективный анализ.

К концепции команды равных в MSF тесно примыкает идея о том, что каждая ролевая группа имеет зону ответственности и защищает интересы заинтересованных лиц из этой зоны (более подробно об этом далее).

Модель проектной группы в MSF может масштабироваться в зависимости от числа участников.

### ***Ролевые группы и роли***

Методология MSF основана на постулате о качественных целях, достижение которых определяет успешность проекта. Эти цели обуславливают модель проектной группы. В то время как за успех проекта ответственна вся команда, каждая из ее ролевых групп, определяемых моделью, ассоциирована с одной из целей и работает над ее достижением.

MSF for Agile Software Development выделяет **7 ролевых групп** (см. рисунок 4.2):

- Управление программой (*program management*);
- Управление продуктом (*product management*);
- Управление выпуском (*release operations*);

- Архитектура продукта (*architecture*);
- Разработка (*development*);
- Тестирование (*test*);
- Удовлетворение потребителя (*user experience*);

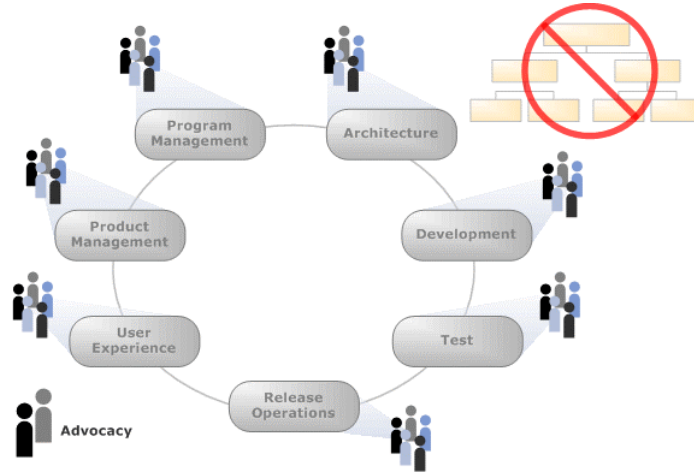


Рисунок 4.2 - Модель команды в MSF 4.0 – ролевые группы [36]

и **6 ролей** (см. рисунок 4.3):

- **менеджер проекта** (*project manager*) – ролевая группа «Управление программой», который отвечает за соблюдение ограничений проекта, его основная задача – вести процесс разработки таким образом, чтобы нужный продукт был выпущен в нужное время, он координирует деятельность всех членов группы, он должен полностью понимать все модели и процессы, повышающие эффективность труда;

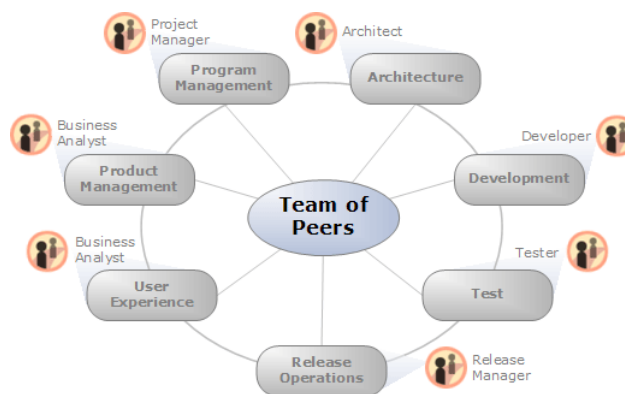


Рисунок 4.3 - Модель команды в MSF 4.0 – роли

- **бизнес-аналитик** (*business analyst*) или **менеджер продукта** – ролевые группы «Управление продуктом» и «Удовлетворение потребителя», который отвечает за удовлетворение требований заказчика. Для этого менеджер продукта выступает представителем заказчика в группе разработчиков и представителем команды у заказчика. Менеджер продукта совместно с менеджером проекта должны прийти к компромиссному решению относительно функциональных возможностей продукта, сроков его разработки и финансирования проекта, так называемый *треугольник ограничений проекта* (рисунок 4.4). Существует закон Лермана: «Любую техническую проблему можно преодолеть, имея достаточно времени и денег» (Следствие Лермана: «Вам никогда не будет хватать либо времени, либо денег»). Для того, чтобы решить проблему свойственной ИТ-проектам неопределенности и рискованности, одним из ключевых факторов их успеха являются эффективные компромиссные решения (*trade-offs*). Хорошо известна взаимозависимость между ресурсами проекта (людскими и финансовыми), его календарным графиком (временем) и реализуемыми возможностями (рамками). Эти три переменные образуют треугольник компромиссов (*tradeoff triangle*), приведенный на рисунке 3.17.

После достижения равновесия в этом треугольнике изменение на любой из его сторон для поддержания баланса требует модификаций на другой (двух других) сторонах и/или на изначально измененной стороне. Нахождение верного баланса между ресурсами, временем разработки и возможностями – ключевой момент в построении решения, должным образом отвечающего нуждам заказчика;



Рисунок 4.4 - Треугольник ограничений проекта

- **релиз-менеджер** (*release manager*) или **логистик** – ролевая группа «Управление выпуском», который отвечает за простоту развертывания и постоянное сопровождение. Логистик обязан разбираться в инфраструктуре продукта и требованиях к его сопровождению, кроме этого он должен уметь координировать установку программного обеспечения и оценить ее результаты. От него требуется хорошая техническая подготовка, он должен уметь устанавливать и настраивать

пользовательские системы. Если проект крупный, то логистик должен иметь опыт развертывания крупномасштабных приложений на нескольких десятках или сотнях компьютерах. Одна из задач логистика – консультация группы сопровождения и пользователей после развертывания системы;

- **архитектор** (*archrect*) – ролевая группа «Архитектура», который отвечает за разработку дизайна проекта (продукта). Архитектор на самом деле входит в группу разработчиков, но на нем лежит ответственность за принятие проектных решений;
- **разработчик** (*developer*) – ролевая группа «Разработка», отвечает за реализацию системы и разрабатывает комплект проектной документации. Как программисты разработчики отвечают за низкоуровневое проектирование и оценку затрат на реализацию продукта. Разработчики, как правило, сами оценивают сроки выполнения своей работы. Такая концепция MSF – создание графиков ответственного за выполнение конкретного участка членами команды – называется *составлением расписания «снизу-вверх»* (рисунок 4.6). Такой подход повышает точность оценок (т.к. они основаны на конкретном опыте разработчика) и повышает ответственность исполнителя (т.к. план работ одновременно является обязательством члена команды). Планирование работ – задача всей проектной группы, но в основании пирамиды лежит работа разработчиков. Менеджер программы добавляет в общий план работ некоторый резерв времени, чтобы гарантировать соблюдение сроков выполнения работ даже при возникновении непредвиденных обстоятельств.

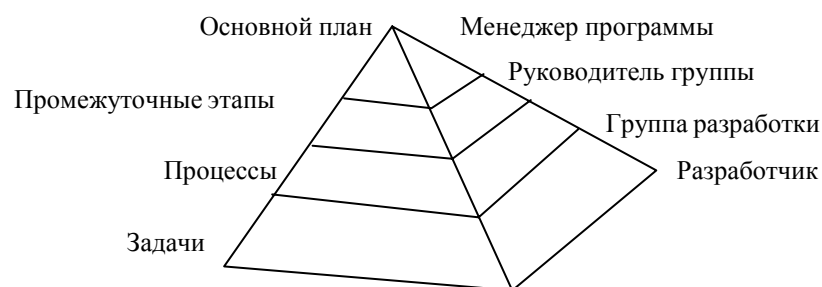


Рисунок 4.6 – Составление графика «снизу-вверх»

- **тестер** (*tester*) – ролевая группа «Тестирование», отвечает за выявление и устранение проблем (дефектов) не только на уровне кода продукта, но и его спецификации и документацию к нему. При этом испытание продукта происходит в реальных условиях.

**Рекомендации по возможному объединению ролей**

Модель проектной группы в MSF может масштабироваться в зависимости от числа участников. Масштабирование модели проектной группы MSF for Agile Software Development на случай больших команд выходит за рамки нашего курса. Мы рассмотрим ситуацию, когда один человек может выполнять в проектной группе несколько ролей (небольшая команда, относительно несложный проект). В этом случае MSF предлагает рекомендации по возможному объединению ролей (таблица 4.1).

Таблица 4.1 – Возможное объединение ролей

	Архитектура продукта	Управление продуктом	Управление программой	Разработка	Тестирование	Удовлетворение потребителя	Управление выпуском
Архитектура продукта		Нет	Да	Да	Не желательно	Не желательно	Не желательно
Управление продуктом	Нет		Нет	Нет	Да	Да	Не желательно
Управление программой	Да	Нет		Нет	Не желательно	Не желательно	Да
Разработка	Да	Нет	Нет		Нет	Нет	Нет
Тестирование	Не желательно	Да	Не желательно	Нет		Да	Да
Удовлетворение потребителя	Не желательно	Да	Не желательно	Нет	Да		Не желательно
Управление выпуском	Не желательно	Не желательно	Да	Нет	Да	Не желательно	

**Координация работы с внешними группами**

Проектная группа, рассчитывающая на успех, должна взаимодействовать с внешними группами – как с заказчиками, так и с пользователями, так и с другими разработчиками [40]. Этим занимаются менеджер программы, менеджер продукта (бизнес-аналитик) и релиз-менеджер (логистик). В обязанность этих ролей входят как внутренние, так и внешние контакты, в то время, как архитекторы, разработчики и тестеры изолированы от общения с внешним миром (такая изоляция приводит к повышению эффективности труда этих двух групп). Важно, чтобы взаимодействие этих групп было понятным и явным. На рисунке 4.7 проиллюстрирована координация взаимодействий, связанная как с бизнесом, так и с технологиями.

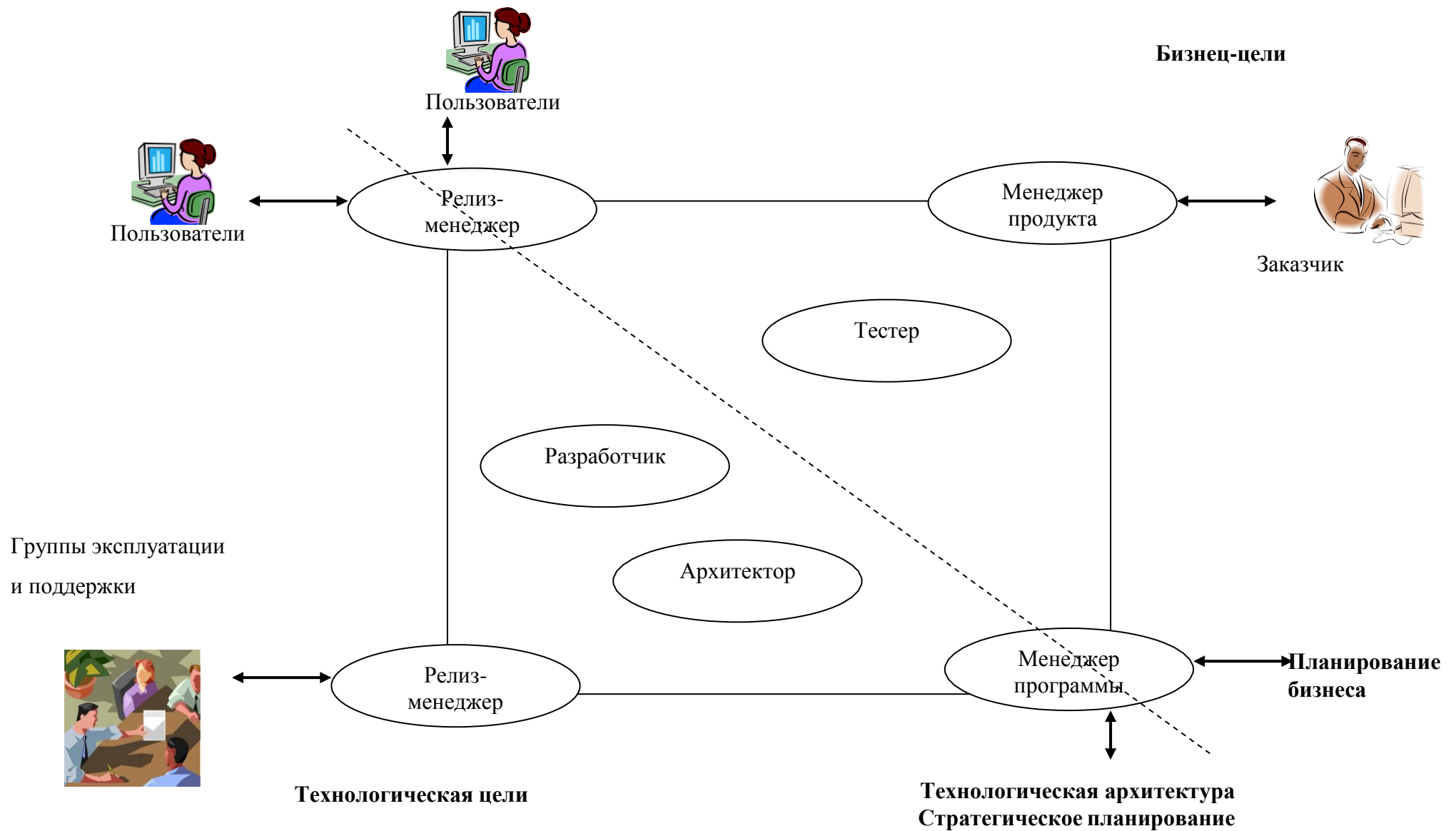


Рисунок 4.7 – Взаимодействие проектной группы с представителями заказчика

***Учебный пример. Формирование команды***

Пусть проектная группа состоит из 4 человек. Представим возможное распределение ролей, позволяющее выполнить поставленную учебную задачу.

Каждому участнику нашей команды невозможно выдать по одной роли. Разработчиков должно быть больше одного, а остальные роли придется совмещать. Как именно, сейчас обсудим.

Во-первых, следуя рекомендациям MSF по объединению ролей, дадим одному из разработчиков еще и роль архитектора.

Во-вторых, отбросим в сторону другую крайность – разработчиками не могут быть все. Отдельный участник команды должен заниматься тестированием. Ему же можно выдать «в нагрузку» роль бизнес-аналитика.

Незадействованными остались ролевые группы «Управление программой» и «Управление выпуском». Соответственно роли менеджер проекта и релиз-менеджер достаются еще одному участнику.

В итоге получаем следующее (возможное) распределение:

- ✓ Участник 1 – менеджер проекта и релиз-менеджер;
- ✓ Участник 2 – архитектор и разработчик;
- ✓ Участник 3 – бизнес-аналитик и тестер;
- ✓ Участник 4 – разработчик.



## 5 УПРАВЛЕНИЕ РИСКАМИ В MSF FOR AGILE SOFTWARE DEVELOPMENT

Дисциплина управления рисками MSF возводит процесс управления рисками в ранг стратегической задачи, затрагивающей все фазы проекта. В рамках MSF управление рисками – это процесс выявления, анализа и превентивной работы над рисками в целях избежания их превращения в проблемы, приносящие ущерб или иной вред.

В ходе всего проекта команда должна уделять внимание дисциплине управления рисками. Основные ее характеристики:

- она всеобъемлюща и принимает во внимание все составляющие проекта: людей, бизнес-процессы, технологические элементы и т.д.;
- она включает в себя пошаговый, систематический и воспроизводимый процесс управления рисками проекта;
- ее использование непрерывно на протяжении всего жизненного цикла проекта;
- она превентивна и не исходит из идеологии действия по факту случившегося.
- она вовлекает всю проектную группу в непрерывное извлечение уроков из полученного опыта.

### *Основные сведения о рисках*

Прежде всего, определим, что такое «риск». Заглянем в «Толковый словарь русского языка» С.И. Ожегова: «Риск – 1. Возможность опасности, неудачи. 2. Действие наудачу в надежде на счастливый исход» [41]. Отметим, что понятие «риск» включает в себя два по сути противоположных толкования: одно со знаком минус, второе со знаком плюс.

Риск проекта (*project risk*) понимается в MSF именно в таком полном виде – как всякое событие или условие, которое может оказать как негативное, так и позитивное влияние на итоги проекта [19]. Такое же расширенное понимание спекулятивного риска (*speculative risk*) присутствует, например, в финансовой индустрии, где решения в условиях неопределенности могут привести к получению прибыли точно так же, как и к убыткам. Указанное понимание противоположно понятию чистого риска (*pure risk*) в индустрии страхования, где неопределенности связываются только лишь с потенциальными убытками в будущем.

Важно отметить, что риски не есть проблемы. Проблемы – это нечто, имеющие место в настоящее время, в то время как риски относятся к будущему и носят вероятностный характер (могут и не состояться). Однако риски могут стать проблемами, если ими эффективно не управлять.

Цель управления рисками – максимизировать их положительное влияние (открывающиеся возможности), но при этом минимизировать связанные с ними негативные факторы (убытки).

Дисциплина управления рисками в MSF основана на убеждении, что такое управление должно выполняться превентивно; это часть формального и систематического процесса, трактующего усилия, затрачиваемые на управление рисками, с позитивной точки зрения.

Говоря о рисках, MSF выделяет несколько ключевых концепций:

1. *Риск – неотъемлемая часть всякого проекта или процесса.* Несмотря на то, что различные проекты могут быть связаны с большим или меньшим числом рисков, не существует ни одного проекта, полностью свободного от них. Цель состоит не в том, чтобы избежать рисков, а в том, чтобы предвосхищать потенциальные проблемы и заблаговременно готовиться к их решению, если они возникнут.
2. *Выявление рисков нужно всячески одобрять.* Проектная группа должна смотреть на выявление рисков как на позитивную деятельность. Знание о существовании рисков – необходимое условие эффективной работы над ними. Следовательно, перспективы успеха проектной группы от проведения работы по выявлению рисков лишь увеличиваются.
3. *Оценка рисков должна вестись постоянно.* Обстоятельства, в которых проектная группа работает над созданием решения, обладают постоянной изменчивостью, следовательно, команда должна регулярно проводить переоценку выявленных рисков и постоянно следить за появлением новых. Управление рисками должно быть интегрировано в общий жизненный цикл проекта.
4. О положении дел в проекте нужно судить не по количеству рисков, связанных с его выполнением, а по степени проработанности процедуры их выявления, анализа и управления ими.

### ***Планирование управления рисками***

Во время проектных фаз выработки концепции и планирования проектная группа должна разработать формальный документ, описывающий управление рисками в проекте. В этом документе должны быть даны ответы на целый ряд вопросов, из которых рассмотрим основные.

- Как будет реализовываться процесс управления рисками? Из каких шагов состоит этот процесс?

- Кто будет осуществлять действия по управлению рисками? Какие для этого требуются навыки/квалификация? Требуется ли дополнительное обучение?
- Как будут строиться планы управления рисками и планы мероприятий по смягчению возможных негативных последствий?
- Как деятельность по управлению рисками будет интегрирована в общий план проекта?
- Какие действия будут предпринимать отдельные члены проектной группы для управления рисками?
- Какие ресурсы доступны для управления рисками?
- Каковы временные ограничения в мероприятиях, связанных с управлением рисками?

Поскольку риски являются неотъемлемой частью всех фаз всех проектов от начала и до конца, должны быть изначально выделены и должным образом распределены ресурсы, необходимые для эффективного управления рисками. Планирование управления рисками осуществляется проектной группой во время фаз выработки концепции и планирования, и результирующий план управления рисками должен определять конкретные действия, ответственность за которые возложена на определенных членов проектной группы. Эти задачи должны быть интегрированы в сводный план проекта (*master project plan*) и в сводный календарный график проекта (*master project schedule*).

### ***Процесс управления рисками***

Дисциплина управления рисками MSF отстаивает превентивное управление рисками, непрерывную оценку имеющихся рисков и интеграцию этих процессов в общую деятельность по принятию решений на протяжении всего жизненного цикла проекта или бизнес-процесса.

Процесс управления рисками MSF определяет шесть логических шагов, посредством которых проектная группа управляет текущими рисками, разрабатывает и исполняет стратегии управления рисками и извлекает уроки из своего опыта для использования на уровне всего предприятия (см. рисунок 5.1).

1. *Выявление рисков (risk identification)* – этап, позволяющий членам проектной группы вынести на обсуждение всей команды факты наличия рисков. Выявление рисков является начальной стадией процесса управления ими. Оно должно быть осуществлено как можно раньше, и к нему необходимо постоянно возвращаться на протяжении всего жизненного цикла проекта.

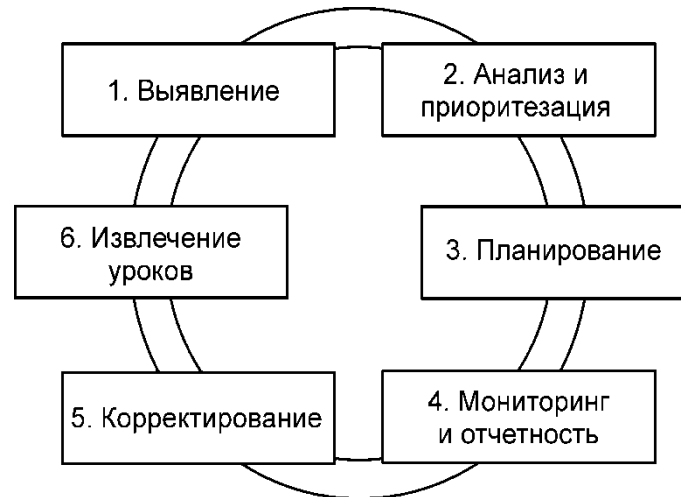


Рисунок 5.1 - Процесс управления рисками MSF [31]

2. *Анализ рисков (risk analysis)* – этап преобразования накопленных во время предыдущего шага оценок и данных в форму, позволяющую осуществить приоритезацию рисков. Приоритезация рисков (*risk prioritization*) позволяет проектной группе управлять наиболее важными из них, выделяя для этого необходимые ресурсы.
3. *Планирование рисков (risk planning)* выполняется исходя из информации, полученной на этапе их анализа, и имеет целью выработку стратегий, планов и конкретных шагов. Календарное планирование рисков (*risk scheduling*) интегрирует эти планы в повседневный процесс управления проектом, обеспечивая непрерывность управления рисками. Эта стадия напрямую увязывает планирование рисков с планированием проекта в целом.
4. *Мониторинг рисков (risk tracking)* выполняется для наблюдения за конкретными рисками и прогрессом в осуществлении составленных планов. Мониторингу должны быть подвергнуты сделанные *оценки вероятности (probability)* риска, его *угрозы (impact)*, *ожидаемая величина риска (exposure)* и прочие факторы, способные повлиять на приоритет рисков. Наблюдению подвергаются также составленные планы, имеющиеся ресурсы и принятый календарный график.
5. *Корректирование ситуации (risk control)* представляет собой процесс исполнения принятых в отношении рисков планов и контроля за ходом их исполнения. Этот процесс также включает в себя инициирование изменений всего проекта (*project change control requests*), если изменения в состоянии рисков либо в соответствующих планах влияют на прогнозируемый объем работы, требуемые ресурсы или сроки.

6. *Извлечение уроков* (risk learning) формализует процесс усвоения накопленного за время работы над проектом опыта.

Необходимо отметить, что описанные этапы являются логическими шагами и не обязательно должны следовать друг за другом в строгом хронологическом порядке. Проектные группы могут циклически повторять шаги выявления-анализа-планирования по мере обнаружения дополнительных факторов, влияющих на проект.

### ***Управление рисками как составная часть жизненного цикла проекта***

Процесс управления рисками в MSF тесно интегрирован с общим жизненным циклом проекта. Оценка рисков может быть начата даже на этапе выработки концепций, поскольку в этот момент проектная группа и заинтересованные стороны начинают формировать видение проекта, его границ и рамок. По результатам анализа и планирования рисков необходимые планы по предотвращению и смягчению последствий должны быть сразу включены в календарный график проекта и его сводный план.

В ходе выполнения проекта команда постоянно проводит мониторинг рисков, осуществляет необходимые корректирующие действия в соответствии с расписанием и планом проекта, и при наступлении связанных с триггерами рисков событий.

Действия по выявлению и анализу новых рисков могут проводиться по достижении *вех (milestones)* проекта (подробнее о вехах далее в этой лекции). Также в этот момент можно резюмировать извлеченные уроки.

По ходу проекта тип рисков, которым уделяется внимание, также должен изменяться. На ранних этапах доминируют риски связанные с бизнесом, рамками проекта, требованиями к конечному продукту и проектированием этого продукта. С течением времени начинают играть важную роль технологические риски, связанные с реализацией проекта. Затем внимание переходит к рискам поддержки и сопровождения. Для организации деятельности по выявлению рисков в моменты основных фазовых переходов полезно использовать контрольные перечни (*checklists*) и классификации рисков.

### ***5.6 Пример «Выделение рисков»***

Рассмотрим учебный пример «Система бронирования билетов для авиакомпании» и выделим некоторые возможные риски (таблица 5.1).

Таблица 5.1 – Возможные риски

	Наименование риска	Комментарий
1	Не успеем сдать проект во время	Из-за неправильной организации работ затратим больше времени, чем заявлено в контракте
2	Не хватит квалификации персонала	При создании продукта используется новая технология (JNL), персонал с ней не работал и может не разобраться
3	Один из членов команды заболит	Команда не многочисленна и отсутствие одного из членов команды ведет задержке в работах
4	Заказчик изменит требования	В данный момент требования согласованы с заказчиком и утверждены, но в процессе работы заказчик может захотеть добавить в решение новую функциональность
5	Авария на подстанции, отключение электричества	Потеряем время, пока авария будет устраняться.
6	Отключат доступ к Интернету	Ухудшится возможность быстрого получения необходимых сведений, пропадет электронная почта и другие средства коммуникации
7	Заказчик вовремя не оплатит счета	Задержка в покупке необходимого оборудования
8	Из-за необнаруженной вовремя ошибки система нанесет урон заказчику	На время устранения ошибки пассажиры не смогут заказывать билеты через систему. Потребуется “ручная” работа персонала. Компания потеряет возможных клиентов и часть прибыли.
9	На стороне заказчика нет заявленной в требованиях аппаратуры	Не сможем адекватно развернуть систему
10	Не сможем подобрать необходимые кадры	Придется совмещать роли

Как видно, рисков не так уж много, однако можно заметить, что они происходят из самых разных областей и связаны с людьми (риск №3), с процессами (риски №1, №8, №10), с технологиями (риск №2), с внешними условиями (риски №5, №6), с заказчиком (риски №4, №7, №9). Помимо представленных риски могут иметь и другие источники.

Далее MSF рекомендует для каждого риска представить формулировку – выражение на естественном языке причинно-следственной связи между реально существующим фактором проекта и потенциально возможным, еще не случившимся событием или ситуацией. Первая часть формулировки риска называется *условием* (*condition*) и содержит описание существующего фактора или особенности проекта, которые, по мнению проектной группы, могут сделать результат проекта убыточным либо же сократить получаемую от проекта прибыль. Вторая часть формулировки риска называется *(по)следствием* (*consequence*). Она описывает ту нежелательную ситуацию, которой следует избежать.

В качестве примера представим формулировки некоторых выявленных выше рисков (таблица 5.1).

Таблица 5.1 – Перечень причин и последствий возникновения рисков

Первопричина	Условие	Последствие	Приносимый ущерб
Нехватка кадров	Один из членов команды заболеет	Функции заболевшего придется передать другому	Потери времени
Форс-мажор	Авария на подстанции, отключение электричества	Потеряем время, пока авария будет устраняться	После устранения аварии придется увеличить нагрузку, чтобы наверстать потери времени
Организация работы	Не сможем подобрать необходимые кадры	Участникам придется совмещать роли	Дополнительные трудозатраты, снижение качества продукта, увеличение времени разработки решения

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

- 1 Программы следующего десятилетия // Открытые системы. – Декабрь, 2001. – С.60 –71.
- 2 Липаев, В.В. Программная инженерия. Методологические основы: Учеб. [Текст] / В. В. Липаев. - М.: ТЕИС, 2006. – 608 с.
- 3 Аптекарь, М. Д. История инженерной деятельности/ М. Д. Аптекарь, С. К. Рамазанов, Г. Е. Фрегер. – Киев, 2003. – 204 с. : ил.
- 4 ГОСТ Р ИСО/МЭК 12207-99. Информационная технология. Процессы жизненного цикла программных средств. – Введ. 2000-07-01. - М.: ИПК Издательство стандартов, 2000. – 75 с.
- 5 Sommerwyl, I. Инженерия программного обеспечения/ Иан Соммервиль. – М., СПб, Киев: Издательский дом «Вильямс», 2002. – 626 с. : ил.
- 6 Карпенко, С.Н. Введение в программную инженерию [Электронный ресурс]. - <http://www.software.unn.ac.ru>.
- 7 Bauer F.L. Software Engineering. Information Processing, 71, 1972
- 8 Карпенко, С.Н. Введение в программную инженерию: курс лекций [Текст]. – Н.Новгород, изд-во ННГУ, 2005.
- 9 IEEE STD 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, IEEE Computer Society, 1990.
- 10 Schach, 99
- 11 Терехов, А.Н. Технология программирования: учеб. пособие [Текст]. – М.: Бином, 2006. – 147 с.
- 12 ядро знаний SWEBOK
- 13 Кодекс этики и профессиональной практики программной инженерии [Электронный ресурс]. - <http://www.acm.org/serving/se/code.htm>.
- 14 Вигерс, К. Разработка требований к программному обеспечению [Текст]: Пер. с англ. /К. Вигерс. – М.: Издательский торговый дом «Русская Редакция», 2004. - 576с.: ил.
- 15 Официальный сайт корпорации Opengroup [Электронный ресурс]. - [www.opengroup.org](http://www.opengroup.org).
- 16 <http://zachmaninternational.com/index.php/home-article/13>
- 17 [http://www.cio.gov/Documents/E-Gov\\_Guidance\\_July\\_25\\_Final\\_Draft\\_2\\_0a.pdf](http://www.cio.gov/Documents/E-Gov_Guidance_July_25_Final_Draft_2_0a.pdf)



- 18 Rational Unified Process. Best Practices for Software Development Teams [Электронный ресурс]. - [http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251\\_bestpractices\\_TP026B.pdf](http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf).
- 19 Руководство к своду знаний по управлению проектами (PMBOK-4) [Текст]: 4-е изд. - М.: Институт управления проектами, 2010. - 496 с.
- 20 Criteria for Evaluation of Software. ISO TC97/SC7 #383.
- 21 Липаев, В. В. Качество программного обеспечения [Текст] / В.В. Липаев. – М.: Финансы и статистика, 1983. – 263 с. : ил.
- 22 J. McCall, P. Richards, G. Walters. Factors in Software Quality. three volumes, NTIS AD-A049-014, AD-A049-015, AD-A049-055, November 1977.
- 23 B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. MacLeod, and M. J. Merritt. Characteristics of Software Quality. North Holland, 1978.
- 24 B. Boehm. Software Risk Management. IEEE Computer Society Press, CA, 1989.
- 25 ISO/IEC TR 15504. Information Technology – Software Process Assessment (Part 1–9). – 1997.
- 26 ISO/IEC 9126, Information Technology – Software quality characteristics and metrics (Part 1–4). – 1997.
- 27 Жоголев, Е. А. Технологии программирования [Текст]. – М.: Научный мир, 2004. – 216 с. : ил.
- 28 Royce, W.W. Managing the Development of Large Software Systems. [Электронный ресурс]. - <http://facweb.cti.depaul.edu/jhuang/is553/Royce.pdf>.
- 29 Фаулер, М. Рефакторинг: улучшение соответствующего кода [Текст]. - СПб.: Символ–Плюс, 2003. – 432 с.
- 30 Брукс, Ф. Мифический человеко-месяц или как создаются программные системы [Текст]. – СПб. : Символ-плюс, 1999. – 321 с. : ил.
- 31 Microsoft Solutions Framework. Модель процессов MSF. White Paper. 2002. [Электронный ресурс]. - <http://www.microsoft.com/msf>.
- 32 Якобсон, А.. Унифицированный процесс разработки программного обеспечения [Текст]/ А.Якобсон, Г. Буч, Дж. Рамбо. - СПб.: Питер, 2002.
- 33 Фаулер, М. Новые методологии программирования [Электронный ресурс]. - <http://www.maxkir.com/sd/newmethRUS.html>.
- 34 Салливан Эд. Время – деньги. Создание команды разработчиков программного обеспечения/ Пер.с англ. – М.: Русская редакция, 2002. – 364с.

- 
- 35 Коуберн, А. Люди как нелинейные и наиболее важные компоненты в создании программного обеспечения. [Электронный ресурс]. - <http://www.optim.ru/cs/2002/3/cobern/people.asp>.
- 36 MSF for Agile Software Development Process Guidance [Электронный ресурс]. - <http://go.microsoft.com/fwlink/?linkid=63524>.
- 37 Weinberg, J., The Psychology of Computer Programming, Silver Edition, Dorset House, 1998.
- 38 ДеМарко, Т., Т. Листер. Человеческий фактор: эффективные проекты и команды [Текст]/ Т.ДеМарко, Т. Листер/ Пер. с англ. - СПб: Символ-Плюс, 2004.
- 39 Константин, Л. Человеческий фактор в программировании [Текст]/ Л. Константин./ Пер. с англ. - СПб: Символ-Плюс, 2004. - 384 с.
- 40 Учебный курс MSF
- 41 Ожегов, С.И. Толковый словарь русского языка [Текст]/ С.И Ожегов, Н.Ю.Шведова. - М.: Издательство "Азъ", 1992.

## ПРИЛОЖЕНИЕ А

### Кодекс этики и профессиональной практики программной инженерии

#### A.1 Кодекс этики IEEE-CS/ACM

В разработке этических обязательств ведущую роль играют профессиональные сообщества, такие как:

- ACM – Association for Computing Machinery - Ассоциация по вычислительной технике;
- IEEE – Institute of Electrical and Electronic Engineers – Институт инженеров по электротехнике и электронике;
- CS - British Computer Society – Британское компьютерное общество.

совместно разработали и опубликовали IEEE-CS/ACM Software Engineering Code of Ethics and Professional Practices – Кодекс этики и профессиональной практики программной инженерии.

Члены этих организация принимают обязательство следовать этому кодексу в момент вступления в организацию. Кодекс содержит восемь Принципов, связанных с поведением и решениями, принимаемыми профессиональными программистами, включая практиков, преподавателей, менеджеров и руководителей высшего звена. Кодекс распространяется также на студентов и «подмастерьев», изучающих данную профессию.

Кодекс имеет краткую и полную версии.

#### A.2 Кодекс этики - Преамбула

Краткая версия кодекса:

- суммирует стремления кодекса на высоком уровне абстракции;
- полная версия показывает, как эти стремления отражаются на деятельности профессиональных программистов;
- без высших принципов детали кодекса станут казуистическими и нудными;
- без деталей стремления останутся возвышенными, но пустыми;
- вместе же они образуют целостный кодекс.

Программные инженеры должны добиваться, чтобы анализ, спецификация, проектирование, разработка, тестирование и сопровождение программного обеспечения стали полезной и уважаемой профессией. В соответствии с их приверженностью к

процветанию, безопасности и благополучию общества, программные инженеры будут руководствоваться следующими 8 Принципами.

### ***1. ОБЩЕСТВО***

- Программные инженеры будут действовать соответственно общественным интересам.

### ***2. КЛИЕНТ И РАБОТОДАТЕЛЬ***

- Программные инженеры будут действовать в интересах клиентов и работодателя, соответственно общественным интересам.

### ***3. ПРОДУКТ***

- Программные инженеры будут добиваться, чтобы произведенные ими продукты и их модификации соответствовал высочайшим профессиональным стандартам.

### ***4. СУЖДЕНИЕ***

- Программные инженеры будут добиваться честности и независимости в своих профессиональных суждениях.

### ***5. МЕНЕДЖМЕНТ***

- Менеджеры и лидеры программных инженеров будут руководствоваться этическим подходом к руководству разработкой и сопровождением ПО, а также будут продвигать и развивать этот подход.

### ***6. ПРОФЕССИЯ***

- Программные инженеры будут улучшать целостность и репутацию своей профессии соответственно с интересами общества.

### ***7. КОЛЛЕГИ***

- Программные инженеры будут честными по отношению к своим коллегам и будут всячески их поддерживать.

### ***8. ЛИЧНОСТЬ***

- Программные инженеры в течение всей своей жизни будут учиться практике своей профессии и будут продвигать этический подход к практике своей профессии.



---

## СОДЕРЖАНИЕ

Технология быстрой разработки приложений RAD .....	2
1    Лабораторная работа № 1 Разработка технического задания на программную систему.....	5
2    Лабораторная работа № 2 Описание и анализ предметной области .....	12
3    Лабораторная работа № 3 Постановка задачи .....	2
4    Лабораторная работа № 4 Разработка структуры системы .....	2
5    Лабораторная работа № 5 Разработка спецификации требований.....	2
6    Лабораторная работа № 6 Разработка информационно-логического проекта системы.....	2
7    Лабораторная работа № 7 Разработка алгоритмов обработки данных .....	2
8    Лабораторная работа № 8 Разработка прототипа интерфейса системы .....	2
9    Оформление отчета .....	3
Список использованных источников .....	2
Приложение А Пример оформления технического задания на разработку ПС .....	2
Приложение Б Структура содержания пояснительной записки .....	1
Приложение В Пример оформления реферата .....	19

---

## ТЕХНОЛОГИЯ БЫСТРОЙ РАЗРАБОТКИ ПРИЛОЖЕНИЙ RAD

Один из подходов к разработке программного обеспечения (ПО) в рамках спиральной модели жизненного цикла (ЖЦ) – получившая широкое распространение методология (технология) быстрой разработки приложений RAD (*Rapid Application Development* – быстрая разработка приложений) [1, 2]. Данная модель очень хорошо подходит к разработке учебных программ, т.к. включает в себя три составляющие:

- небольшую команду программистов (от 2 до 4 человек);
- короткий, но тщательно проработанный производственный график (от 2 до 4 мес.);
- повторяющийся цикл, при котором разработчики по мере того, как приложение начинает обретать форму, запрашивают и реализуют в продукте требования, полученные через взаимодействие с заказчиком.

Команда разработчиков должна представлять собой группу студентов, имеющих опыт в анализе, проектировании, кодировании и тестировании ПО, которые способны хорошо взаимодействовать как внутри самой команды, так и с пользователями и/или заказчиками. В качестве заказчика и пользователя системы выступает преподаватель.

ЖЦ ПО [1] по технологии RAD состоит из **четырёх фаз** (рисунок 1):

1. Анализа и планирования требований;
2. Проектирования;
3. Построения;
4. Внедрения.

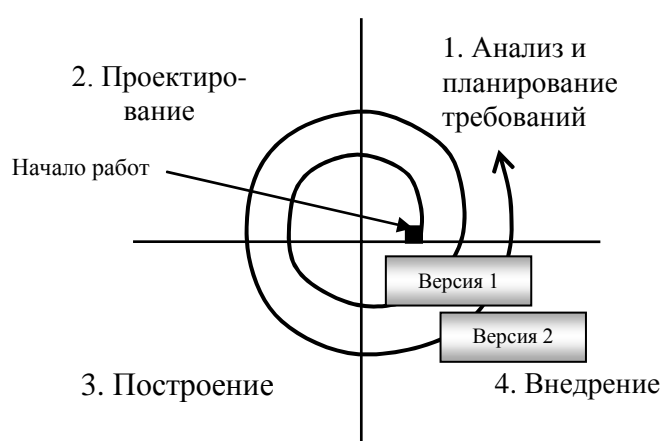


Рисунок 1 – Модель ЖЦ по технологии RAD

Рассмотрим адаптированный к учебному процессу вариант технологии RAD.

На **первой фазе анализа и планирования требований** преподаватель в вербальной форме формулирует постановку задачи: определяет функции, которые должна выполнять

---

система, выделяет наиболее приоритетные из них, требующие проработки в первую очередь, описывает информационные потребности (связи). На основании этих данных разработчики (под руководством преподавателя) формулируют требования к системе, которое фиксируется в виде технического задания (ТЗ) и подписывается всеми участниками проекта. В ТЗ ограничивается масштаб проекта, устанавливаются временные рамки для каждой из фаз. Для того, чтобы более четко и полно сформулировать постановку задачи, разработчики должны исследовать предметную область, в рамках которой выполняется разработка учебного проекта: познакомиться с основной терминологией, изучить информационные потоки объекта автоматизации, при необходимости изучить математический аппарат, который необходим для поддержки работы системы, познакомиться с основными достижениями в данной предметной области (найти и изучить основные функциональные возможности систем-аналогов).

*Результатом фазы должны быть:* список расставленных по приоритету функций будущей ПС; предварительная функциональная модель ПС; предварительная информационная модель ПС.

На **второй фазе проектирования** часть команды принимает участие в техническом проектировании системы под руководством преподавателя и, взаимодействуя с ним, уточняют и дополняют требования к системе, которые не были выявлены на предыдущей фазе. Более подробно рассматриваются *процессы системы*. При необходимости корректируется функциональная модель, создаются частичные прототипы: экранов, отчетов, устраняющие неясности или неоднозначности. Устанавливаются требования *разграничения доступа к данным*. На этой же фазе происходит определение необходимой документации. После детального определения состава процессов оценивается количество функциональных элементов разрабатываемой системы и ***принимается решение о разделении системы на подсистемы***. В подходе RAD каждый прототип развивается в часть будущей системы. Таким образом, на следующую фазу передается более полная и полезная информация.

*Результатом данной фазы должны быть:* общая информационная модель системы; функциональные модели системы в целом и подсистем; точно определенные интерфейсы между автономно разрабатываемыми подсистемами; построенные прототипы экранов, отчетов, диалогов и т.п.

На **третьей фазе построения** выполняется непосредственно сама быстрая разработка приложения (реализация подсистем). На данной фазе студенты производят итеративное построение реальной системы на основе полученных в предыдущей фазе моделей, а также



---

требований нефункционального характера. Преподаватель на этой фазе оценивает получаемые результаты и вносит коррективы, если в процессе разработки система перестает удовлетворять определенным ранее требованиям. Тестирование системы осуществляется в процессе разработки.

После окончания разработки подсистем производится постепенная интеграция каждой части системы с остальными, формируется полный программный код, выполняется тестирование системы в целом. Завершается физическое проектирование системы: определяется необходимость распределения данных; осуществляется анализ использования данных; производится физическое проектирование базы данных; определяются требования к аппаратным ресурсам; определяются способы увеличения производительности; завершается разработка документации проекта.

*Результатом фазы является готовая система, удовлетворяющая всем согласованным требованиям.*

На **четвертой фазе внедрения** производятся обучение пользователей, организационные изменения и параллельно с внедрением новой системы осуществляется работа с существующей системой ( до полного внедрения новой ). Учитывая, что технология RAD используется в рамках учебного процесса, данная фаза заменяется фазой «Доработка программной системы».

Технология RAD (как и любая другая) не может претендовать на универсальность, она хороша в первую очередь для относительно небольших проектов, разрабатываемых для конкретного заказчика, так называемое заказное ПО.

В заключение перечислим **основные принципы технологии RAD:**

- разработка приложений итерациями;
- необязательность полного завершения работ на каждом этапе ЖЦ;
- обязательное вовлечение пользователей на этапе разработки;
- тестирование и развитие проекта одновременно с разработкой;
- грамотное руководство разработкой, четкое планирование и контроль выполнения работ.

---

## ЛАБОРАТОРНАЯ РАБОТА №1

### РАЗРАБОТКА ТЕХНИЧЕСКОГО ЗАДАНИЯ НА ПРОГРАММНУЮ СИСТЕМУ

Во время лабораторного практикума должна быть разработана программная система (проект) средней сложности, при этом в соответствии с требованиями сегодняшнего дня разработка системы должна вестись не единолично, а командой разработчиков, каждый из которых в дальнейшем будет выполнять порученную ему часть проекта.

Тема проекта выдается преподавателем (в дальнейшем это руководитель проекта) на первом занятии, в соответствии с которой в течение первых двух недель команда студентов разрабатывает техническое задание по форме, приведенной в приложении А. Разработку ТЗ можно считать началом первой фазы ЖЦ будущей ПС.

Техническое задание разрабатывается в соответствии с ГОСТ 34.602-89 «Техническое задание на создание автоматизированной системы» [2] и в дальнейшем должно стать основным документом, по которому студенты ведут разработку проекта. Любые изменения ТЗ на систему должны быть согласованы с преподавателем и заверены его подписью.

Техническое задание состоит из трех частей:

- 1) содержание задания;
- 2) исходные данные;
- 3) календарный план выполнения работ.

**Часть 1 – «Содержание задания».** Данная часть ТЗ фиксирована, в ней перечислены основные составные этапы выполнения проекта. При разработке ПС в рамках лабораторного практикума и в дальнейшем при выполнении курсового проекта будут использоваться две основные методологии: объектно-ориентированного анализа и проектирования (ООАП) и методология структурного проектирования [3].

ООАП (Object-Oriented Analysis/Design) - технология разработки программных систем, в основу которых положена объектно-ориентированная методология представления предметной области в виде объектов, являющихся экземплярами соответствующих классов [4]. Методология ООАП тесно связана с концепцией автоматизированной разработки программного обеспечения (Computer Aided Software Engineering, CASE) и языком моделирования UML (Unified Modeling Language).

Методология объектно-ориентированного анализа и проектирования используется при выполнении лабораторной работы №2 – описании и анализе предметной области, где

студенты должны выявить взаимосвязи между основными информационными объектами ПС, определить их характеристики и порядок взаимодействия.

Методология структурного проектирования применяется на первой фазе проектирования (лабораторная работа №3) при разделении системы на подсистемы, здесь используется принцип проектирования «сверху вниз». За каждым студентом закрепляется определенный перечень работ (обычно это разработка отдельной подсистемы), выполнение которых в дальнейшем контролирует преподаватель.

**Часть 2 – «Исходные данные к проекту»** включает в себя следующие подразделы:

1. Характеристики объекта автоматизации (или управления);
2. Требования к информационному обеспечению.
3. Требования к техническому обеспечению.
4. Требования к программному обеспечению.
5. Общие требования к проектируемой системе.
6. Перечень дополнительных работ (если необходимо).

*Характеристики объекта автоматизации.* Здесь указываются общие характеристики объекта автоматизации, характерные для рассматриваемой предметной области:

- a) полное название объекта (ов);
- b) условия его функционирования;
- c) *количественные и качественные показатели* объекта, которые являются ограничениями процесса функционирования.

В качестве примера рассмотрим проект «Автоматизированная система составления и разгадывания линейного кроссворда по выбранной теме», ТЗ на который приведено в приложении Б.

Понятие «объект автоматизации» в явном виде в ГОСТ 34.602-89 [2] нигде не определено, но если внимательно прочитать п. 2.4.1. «В подразделе «Назначение системы» указывают вид автоматизируемой деятельности (управление, проектирование и т. п.) и перечень объектов автоматизации (объектов), на которых предполагается ее использовать»<sup>1</sup>, то из него следует, что под «объектами автоматизации» авторы понимали вовсе не процессы. Будем считать, что объектом автоматизации может быть только материальный (интеллектуальный) объект - организация, магазин, цех, отдел, и так далее.

---

<sup>1</sup> ГОСТ 34.602-89, с. 3.

**Комментарии к примеру.** Из названия темы следует, что в данном случае объект автоматизации – это линейный кроссворд, а виды автоматизируемой деятельности – это *процессы*:

- 1) составления/ генерирования кроссворда;
- 2) разгадывания кроссворда;
- 3) работы со словарем понятий.

Обращаю Ваше внимание на то, чем составление кроссворда отличается от генерирования: в первом случае пользователь вручную составляет кроссворд (добавляет понятия, удаляет понятия и т.п.), во втором (генерирование) - кроссворд составляется системой автоматически в соответствии с теми настройками, которые выполнил пользователь. Процесс составления во многом дублирует функции, которые будет выполнять пользователь при редактировании кроссворда, поэтому процесс редактирования кроссворда не нужно выделять отдельно.

Для каждого процесса в ТЗ должны быть указаны количественные показатели-ограничения, для того, чтобы их правильно выбрать, необходимо знать начальные сведения о структуре самого объекта, каким образом будет проходить его построение и разгадывание. Отправной точкой является определение линейного кроссворда (ЛК). Итак, ЛК – это «цепочка слов, которая строится *методом стыкования*, где последняя буква первого слова является первой буквой второго и т. д. В чайнворде, как и в кроссворде, используются только имена существительные в именительном падеже и единственном числе» [5]. Так как ЛК строится из слов, то необходимо указать ограничение на длину слова: минимальную длину слова можно определить равную 3, а максимальную – 15, потому что чаще всего кроссворды будут составляться на бытовые темы и сам ЛК не должен быть очень простым. Чтобы ЛК было интересно разгадывать, он не должен быть очень коротким, поэтому минимальное количество слов, например, 5, а максимальное – 15. Идем дальше: «слова в чайнворде не пересекаются, а только *стыкуются* друг с другом. Иногда цепочку слов изгибают для придания сетке причудливой формы. Длинная изогнутая цепочка может неоднократно пересекать саму себя, как слова в кроссворде, такая головоломка обычно называется кроссчайнвордом». Исходя из этого, можно задать следующее ограничение – на форму отображения ЛК: обычная (линейная), спираль, змейка, W-образная. «В линейных кроссвордах слова могут перекрываться не только одной, но и двумя или тремя буквами, поэтому их длина указывается в скобках при определении к слову» [5], эта часть описания ЛК дает еще одно ограничение: количество букв в пересечении - от 1 до 3. В качестве

---

пожелания, заказчик отметил, что ЛК необходимо строить в двух режимах: ручном и автоматическом (генерация кроссворда), из этого следует еще одно ограничение – составление кроссворда осуществляется с привязкой к словарю понятий. Для того чтобы системы была более универсальной (необходимо обеспечить создание тематических кроссвордов или на общие области знаний), заказчик предложил загружать в систему внешние словари понятий и обеспечить ему возможность редактировать их содержимое. При разгадывании кроссворда могут возникнуть затруднения, поэтому (в соответствии с пожеланиями заказчика) в системе должна быть организована система подсказок, количество которых можно связать с количеством слов – не менее 1 и не более 10% от количества слов.

*Требования к информационному обеспечению.* Разработка информационного обеспечения (ИО) – наиболее важная часть проекта, она может оказать существенное влияние на весь процесс разработки, поэтому уже на стадии разработки ТЗ необходимо определить:

- 1) на основании каких документов разрабатывается методическое и информационное обеспечение системы (нормативные и другие документы);
- 2) перечень исходных данных:
  - а) какие массивы данных используются и в каких форматах;
  - б) на каких носителях эти данные будут поставляться в систему;
- 3) перечень выходных данных:
  - а) какие массивы данных будут являться результатом работы ПС;
  - б) какие документы будут представлены пользователю и в каком виде (указывается вид носителя) и с какой периодичностью;
  - в) какие требования по целостности данных и их защите должны быть выполнены в проектируемой системе.

**Особо должны быть выделены файл-серверные и клиент-серверные части информационного обеспечения, если таковые имеются.**

*Комментарии к примеру.* Для разработки данной системы никаких нормативных документов не требуется (стандартов, инструкций и т.п.), поэтому необходимо только сослаться на информацию, где определена структура и свойства ЛК, а также требования к его построению. Также необходимо определить требования по входным и выходным данным. Большинство параметров ЛК будут задаваться пользователем в режиме диалога: он обязательно должен подключить словарь понятий, из которого будут формироваться задания для кроссворда. В данном случае «словарь понятий» – это текстовый файл определенной

структуры (каждая строка файла – это понятие и его расшифровка), который должен загружаться в систему из внешней памяти (с любого логического диска), с которым пользователь должен иметь возможность работать дополнительно. Обязательное условие заказчика – возможность создания коллекции ЛК, поэтому в системе должна быть предусмотрена возможность сохранения наиболее интересных кроссвордов в файл. На данном этапе еще трудно определить структуру этого файла (она должна учитывать и возможность дальнейшего разгадывания кроссворда с помощью данной системы), поэтому можно написать, что «структура файла определяется в процессе проектирования». Обязательным условием составления любого типа кроссвордов является его целостность (для данного случая - это отсутствие пустых клеток в середине ЛК), поэтому это обязательно должно быть записано в требованиях.

*Требования к техническому обеспечению.* Здесь формулируются ограничения по составу технических средств автоматизации с указанием конкретных типов оборудования и ЭВМ или их составляющих, используемых в проекте, если они заранее известны. Иначе в этом разделе указывается, что состав комплекса технических средств системы определяется в процессе проектирования системы.

*Требования к программному обеспечению.* Здесь приводится перечень используемых системных и прикладных программных средств, включая операционную систему, систему программирования, систему управления базами данных (в случае необходимости) и другие инструментальные средства (например, среда проектирования) с точным наименованием версий, если они заранее известны. Иначе указывается, что состав программного обеспечения определяется в процессе проектирования системы. Дополнительно могут быть указаны требования по совместимости разрабатываемого программного обеспечения с существующими системами.

*Общие требования к проектируемой системе.* В данной части ТЗ отдельно выделяется подраздел «Функции, реализуемые системой». В нем приводится подробный перечень функций, которые должна выполнять проектируемая система (или подсистема) в процессе ее эксплуатации. Отдельно должны быть выделены функции ввода данных, их обработки, передачи, хранения, а также формирования отчетов с выдачей на экран или печатающие устройства, функции управления, работа со справочниками и различные сервисные (обслуживающие систему) функции. Формулировка функций должна быть однозначной и конкретной, так как именно она является основой приемки проекта руководителем и проверки на полноту и качество реализованной системы или подсистемы.

---

**Комментарии к примеру.** Функции, которые должна выполнять данная система, определяются в первую очередь видами автоматизируемой деятельности, которые были определены в п.2.1 ТЗ, часть из них уже была выявлена в ходе обсуждения ограничений на ЛК. Среди неявных функций, про которые не должны забывать разработчики, это:

- а) Визуализация процессов работы с кроссвордом;
- б) Выдача сведений о системе (справочные данные о системе и о том, как с ней работать).

Нужно отметить, что многие перечисленные в ТЗ функции, не раскрываются подробно, их необходимо будут детализировать при разработке функциональной спецификации (лабораторная работа №5).

В других подразделах оговариваются специальные технические требования, предъявляемые к системе:

- по быстродействию (времени реакции на выполнение наиболее важных функций);
- по режиму работы (диалоговый/интерактивный, автоматический);
- по точности (в случае, если в системе производятся математические расчеты, требующие минимизации вычислительных погрешностей, или используются внешние информационные источники (датчики, измерители и т.п.));
- по достоверности;
- по условиям функционирования (диапазон температур, относительная влажность, давление, наличие в атмосфере пыли, вредных примесей и т.д.),

а также все другие количественные и качественные показатели, определяющие эффективность функционирования системы. Кроме того, в данном разделе указываются санитарные правила и нормы (СанПин 2.2.2./2.4.2198-07 [6]) и ГОСТы, требования которых необходимо учитывать при разработке такого класса систем, с учетом того, что системы разворачиваются на средствах вычислительной техники [7, 8].

*Часть 3 – Календарный план выполнения работ.* Технология RAD, как уже говорилось выше, требует жесткого следования плану-графику работ, поэтому в ТЗ оговариваются ключевые задания, по которым преподаватель должен проводить обязательный контроль. Каждый из перечисленных этапов должен завершаться *полностью готовой документацией*, согласованной с заказчиком (руководителем). Невыполнение в срок какого-либо из этапов может привести либо к сдвигу «контрольных точек» по оставшимся этапам, либо к незавершению проекта в срок.

В заключение хотелось бы отметить, что процесс составления ТЗ на систему:

- 
1. требует от разработчиков коллективных обсуждений и принятия ответственных решений;
  2. позволяет выявить наиболее «узкие» места проекта и оценить возможные риски;
  3. дает возможность команде разработчиков распределить между собой все виды выполняемых работ, сосредоточив в дальнейшем усилия на концептуальных аспектах проекта;
  4. определить наиболее приоритетные функции, которые будут составлять каркас системы.



---

## ЛАБОРАТОРНАЯ РАБОТА № 2

### ОПИСАНИЕ И АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

Как уже говорилось ранее, технология RAD включает в себя элементы методологии *объектно-ориентированного проектирования и анализа* предметной области. Для быстрой и эффективной разработки программной системы с минимальным браком требуется определить верное направление работы [9]. Для того чтобы правильно построить систему, сначала необходимо построить ее модель (этим вы будете заниматься позднее). Моделирование - это устоявшаяся и повсеместно принятая *инженерная методика*. Хорошая модель всегда включает элементы, существенно влияющие на результат, и не включает те, которые малозначимы на данном уровне абстракции. Модели строятся для того, чтобы лучше понимать разрабатываемую систему. При этом нужно помнить об основных принципах моделирования, один из которых гласит: «*Лучшие модели - те, что ближе к реальности*» [9]. Поэтому первое, с чего нужно начать разработку системы – это досконально изучить предметную область, в которой Вы будете работать.

В соответствии с методологией ООАП выделяются следующие шаги работы над проектом (системой).

1. *Описание предметной области*. Определение гласит: «Под предметной областью (application domain) принято понимать ту часть реального мира, которая имеет существенное значение или непосредственное отношение к процессу функционирования программы. Другими словами, предметная область включает в себя только те объекты и взаимосвязи между ними, которые необходимы для описания требований и условий решения некоторой задачи» [4]. Следовательно, разработчикам необходимо выделить основные объекты (компоненты), участвующие в функционировании системы, определить их наиболее существенные характеристики, взаимосвязи в рамках решаемой задачи, а также определить основные информационные потоки в системе. При этом отдельные компоненты выбираются таким образом, чтобы при последующей разработке их было удобно представить в форме классов и объектов. В этом случае немаловажное значение приобретает и сам язык представления информации о концептуальной схеме предметной области.

Сложность предметной области определяет количество объектов и связей между ними, поэтому описание должно включать в себя базовые *термины и определения*, сопровождаться различными примерами, в нем могут приводиться различного рода *классификации*, поясняющие различные свойства описываемых объектов. Если в системе используются

математические модели, то они также должны быть описаны с учетом специфики применения.

**Комментарии к примеру.** Для разрабатываемой системы в первую очередь необходимо дать определение кроссворда [10], привести краткую историю его создания, рассказать о разновидностях кроссвордов и особенностях их построения и разгадывания (привести в качестве иллюстраций «сетки» различных кроссвордов)<sup>2</sup>, при этом подробно рассказать об особенностях линейного кроссворда, различных формах его построения (привести иллюстрации). Это особенно важно, т.к. в техническом задании было зафиксировано 4 различные формы представления ЛК (на рисунке 1 представлен один из вариантов представления чайнворда).

Сын Гермеса.						Колючая рыба.			Единица земельной площади.	
Приставка, означает - САМО				Сладкий спиртной напиток				Голубой		

Рисунок 1 – Чайнворд, как разновидность линейного кроссворда

2. Обзор существующих *систем-аналогов* – неотъемлемая часть описания предметной области, т.к. позволяет разработчику определить основные концепции, которые необходимо будет реализовать в системе. Описание должно приводиться с указанием отличительных особенностей разработанных систем, с перечислением их достоинств и недостатков, в отчете обязательно приводятся экранные формы этих систем.

3. Результатом последнего этапа является *диаграмма объектов предметной области* и краткое описание их свойств и функций. При построении данной диаграммы<sup>3</sup> нужно помнить о том, что в данном случае объект – это «конкретная материализация абстракции», а не экземпляр класса, т.е. пока это понятие не программистское. Диаграмма объектов представляет статическую составляющую взаимодействующих между собой объектов, она должна включить в себя только те объекты предметной области, которые потом преобразуются в диаграмму классов. Связи между объектами показывают отношения между ними, при необходимости в диаграмме можно привести и атрибуты (свойства) объектов. На рисунке 2 приведен фрагмент диаграммы объектов для рассматриваемой в качестве примера системы.

<sup>2</sup> Рекомендуется рассказать о 4-5 разновидностях наиболее популярных кроссвордов.

<sup>3</sup> Диаграмма - это графическое представление множества элементов.

---

Диаграммы объектов не позволяют полностью описать объектную структуру системы, поэтому при их использовании нужно сосредоточиться на изображении интересующих вас наборов конкретных объектов [9].

**Комментарии к примеру.** Как видно из диаграммы, ЛК представляет собой (как и любой кроссворд) совокупность двух составляющих: *сетки*, на которой будут располагаться слова, и *задания*, которое по существу представляет собой набор *определений*, которые разъясняют смысл слов (понятий). Набор понятий и определений хранятся во внешнем файле (чаще всего текстовом), который мы ранее называли словарем понятий (см. лабораторную работу №1).

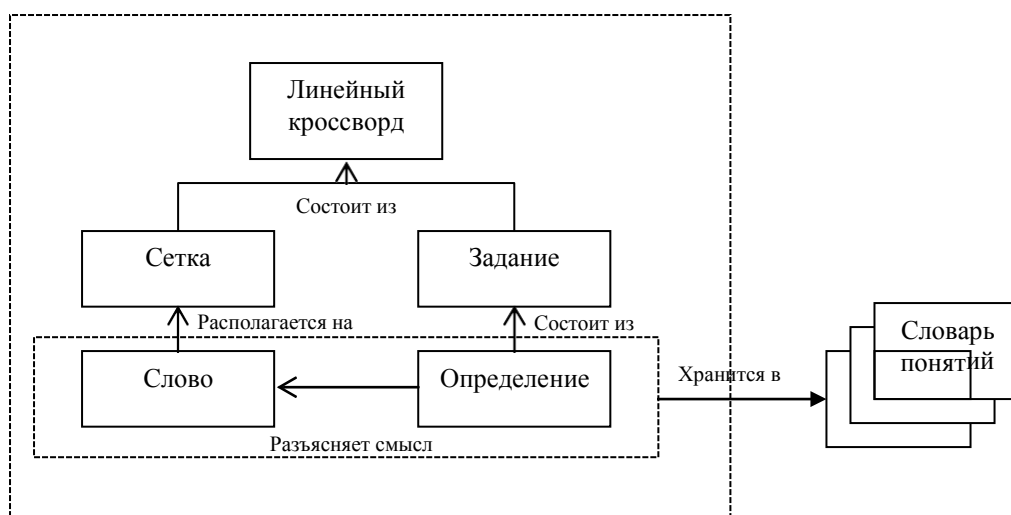


Рисунок 2 – Фрагмент диаграммы объектов предметной области

Коротким пунктиром выделены те сущности, которые могут представлять собой единое целое, длинным – внутренние объекты системы.

---

## ЛАБОРАТОРНАЯ РАБОТА № 3

### ПОСТАНОВКА ЗАДАЧИ

Постановка задачи – заключительный этап первой фазы ЖЦ системы. На данном этапе формулируются все требования, которым должна удовлетворять система. Постановка задачи пишется в повествовательной форме *в будущем времени* на основе ТЗ, в ней должны быть обязательно взаимоувязаны виды автоматизируемой деятельности (с привязкой к объекту(ам) автоматизации) со всеми ограничениями, накладываемыми на них, учтены особенности разрабатываемого информационного обеспечения и перечислены функции, которые должна выполнять система (с привязкой к процессам и информационному обеспечению).

**Комментарии к примеру.** Сначала необходимо сформулировать саму задачу, которая стоит перед разработчиками, обычно она включает в себя название проекта: «Перед авторами поставлена задача - разработать автоматизированную систему составления и разгадывания линейного кроссворда по выбранной теме, которая позволит ...». Далее последовательно описывается все процессы (виды автоматизируемой деятельности), в той последовательности, которая позволит получить требуемый результат. Для разрабатываемой системы таких процессов 3, каждый из них может функционировать независимо от других.

Далее по тексту: «Для достижения поставленной цели необходимо решить следующие задачи:

1. *Составление кроссворда* может проводиться в двух режимах: автоматическом (генерирование) и ручном. В любом случае пользователь должен предварительно задать параметры кроссворда<sup>4</sup>: определить его максимальную длину (не менее 15 и не более 255 символов), выбрать форму его представления (линейная, спиральная, ...), определить количество букв в пересечении (от 1 до 3), подключить словарь понятий (он хранится во внешнем текстовом файле определенной структуры<sup>5</sup>). При этом система должна провести проверку правильности этой структуры и, в случае несоответствия, выдать предупредительное сообщение (см. лабораторную работу №5, описание исключительных ситуаций) и обеспечить повторный ввод параметров. В системе должен осуществляться контроль типов и диапазонов значений параметров. В режиме ручного составления кроссворда (или его редактирования) пользователю должны предоставляться следующие возможности: удаление слова (последнего), добавление нового слова (в конец),

---

<sup>4</sup> см. раздел 2.1 ТЗ «Характеристики объекта автоматизации»

---

редактирование (замена) слова, если оно стоит в середине<sup>6</sup>. При этом должна быть обеспечена навигация по словам либо непосредственно на сетке кроссворда, либо с помощью специальных элементов управления<sup>7</sup>. В случае необходимости пользователь должен иметь возможность сохранить полученный кроссворд в файл (с целью дальнейшего его разгадывания или редактирования), структура файла должна быть определена в ходе проектирования<sup>5</sup>. Необходимо также предусмотреть контроль целостности создаваемого кроссворда (отсутствие пустых мест в середине кроссворда).

*2. Разгадывание кроссворда...*

*3. Работа со словарями понятий...*

В системе также должна быть обеспечена возможность получения справочной информации как о самой системе, так и предоставляемых ею возможностях.

Таким образом, система должна выполнять следующие функции<sup>8</sup>:

(здесь перечисляются все функции, которые были определены в разделе 2.5.1 ТЗ)».

---

<sup>5</sup> см. раздел 2.2 ТЗ «Требования к информационному обеспечению системы»

<sup>6</sup> Последняя функция не обязательна, т.к. достаточно сложна при реализации.

<sup>7</sup> см. раздел 2.5.1 ТЗ «Функции, реализуемые системой».

<sup>8</sup> Эта часть постановки задачи обязательна.

## ЛАБОРАТОРНАЯ РАБОТА № 4

### РАЗРАБОТКА СТРУКТУРЫ СИСТЕМЫ

*Построение структурной схемы программной системы.* На данном этапе система по функциональному признаку разделяется на основные подсистемы, между ними указываются информационные связи и/или связи по управлению, описывается основное назначение подсистем. При разработке структурной схемы используется методология структурного проектирования, в основе которой лежит алгоритмическая декомпозиция и иерархия вида «часть-целое», учитывающая, что внутренние связи элементов внутри подсистем сильнее, чем связь между подсистемами. Декомпозиция системы может повторяться многократно, вплоть до уровня конкретных процедур, при этом должна быть обеспечена целостность системы, а все составляющие компоненты взаимоувязаны. Для этого используются такие принципы разработки, как «сверху-вниз», «разделяй и властвуй», «иерархическое упорядочивание» и другие [3].

Дадим некоторые определения.

В первом приближении можно придерживаться нормативного понятия системы. **Система** (греч. - «составленное из частей», «соединение» от «соединяю») - множество элементов, находящихся в отношениях и связях друг с другом, которое образует определённую целостность, единство [11]<sup>9</sup>.

Как следует из определения, отличительным (главным свойством) системы является ее *целостность*: комплекс объектов, рассматриваемых в качестве системы, должен обладать общими свойствами и поведением. Очевидно, необходимо рассматривать и связи системы с *внешней средой*. В самом общем случае понятие «система» характеризуется:

- наличием множества элементов;
- наличием связей между ними;
- целостным характером данного устройства или процесса.

Система должна представлять собой совокупность элементов (объектов, субъектов), находящихся между собой в определенной зависимости и составляющих некоторое единство (целостность), направленное на достижение определенной цели. Система может являться элементом другой системы более высокого порядка (*надсистема*) и включать в себя системы более низкого порядка (*подсистемы*). То есть систему можно рассматривать как набор подсистем, организованных для достижения определенной цели и описанных с помощью

---

<sup>9</sup> Большой Российский энциклопедический словарь, с. 1437.

набора моделей (возможно, с различных точек зрения), а подсистему – как группу элементов, часть которых составляет спецификацию поведения, представленного другими ее составляющими [9].

К типовым можно отнести следующие подсистемы:

- 1) подсистему управления;
- 2) подсистемы ввода-вывода:
  - a) подсистему настройки параметров;
  - b) файловую подсистему;
  - c) подсистему визуализации;
  - d) подсистему документирования;
  - e) подсистему взаимодействия с базой данных;
- 3) справочную подсистему.

**Полученная в результате декомпозиции структура системы должна сопровождаться кратким описанием включенных в нее подсистем.**

**Комментарии к примеру.** Приведем примерную структурную схему разрабатываемой системы и описание некоторых подсистем (рисунок 3).

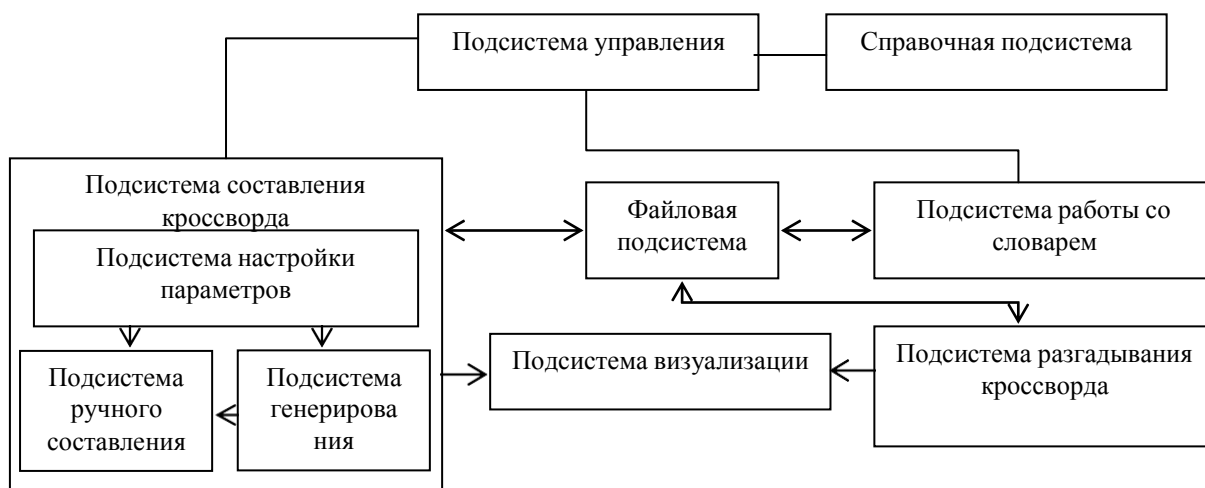


Рисунок 3 – Структурная схема системы

В состав системы входят следующие подсистемы:

- 1) *Подсистема управления*, которая отвечает за взаимодействие подсистем между собой и представлена в виде иерархического меню;
- 2) *Подсистема составления кроссворда*, в состав которой входят:
  - a) *подсистема настройки параметров*, которая отвечает за ввод (выбор) значений параметров кроссворда и проверку корректности этих значений;

- b) *подсистема ручного составления*, которая ...
- c) *подсистема генерирования*, которая ...
- 3) *Подсистема разгадывания*, которая ...
- 4) *Файловая подсистема*, которая ...
- 5) *Подсистема работы со словарем*, которая ...
- 6) *Подсистема визуализации*, которая ...
- 7) *Справочная подсистема*, которая содержит сведения о системе (руководство пользователю) и ее об ее разработчиках.



## ЛАБОРАТОРНАЯ РАБОТА № 5

### РАЗРАБОТКА СПЕЦИФИКАЦИИ ТРЕБОВАНИЙ

Разработка спецификации программного обеспечения является одним из фундаментальных процессов технологии разработки ПО. Этот процесс анализа, формирования, документирования и проверки функциональных возможностей и ограничений системы называется в терминологии программной инженерии «разработка требований» (спецификация требований). Он является критическим этапом в создании всех видов программных систем, что обусловлено тем, что ошибки, допущенные на этой стадии, ведут к возникновению серьезных проблем на этапах проектирования и разработки. Опыт индустрии информационных технологий однозначно показывает, что вопросы, связанные с управлением требованиями, оказывают критически-важное влияние на программные проекты, в определенной степени - на сам факт возможности успешного завершения проектов. Только систематичная работа с требованиями позволяет корректным образом обеспечить моделирование задач реального мира и формулирование необходимых приемочных тестов для того, чтобы убедиться в соответствии создаваемых программных систем критериям, заданным реальными практическими потребностями.

Дадим некоторые определения.

*Требования* - это свойства, которыми должно обладать ПО для адекватного определения функций, условий и ограничений выполнения ПО, а также объемов данных, технического обеспечения и среды функционирования [12, 13].

На практике часто применяется подход, используемый в различных методологиях разработки ПО и базирующийся на определении групп требований к продукту. Такой подход обычно включает группы (типы, категории) требований, например: программные, системные, функциональные, нефункциональные (в частности, атрибуты качества) и т.п.

*Программные требования* (Software Requirements) - свойства программного обеспечения, которые должны быть надлежащим образом представлены в нём для решения конкретных практических задач [14]. Данная область знаний касается вопросов извлечения (сбора), анализа, специфицирования и утверждения требований к разрабатываемой ПС.

*Функциональные требования* задают «что» система должна делать; *нефункциональные* – с соблюдением «каких условий» (например, скорость отклика при выполнении заданной операции). При разработке этих требований в первую очередь необходимо учитывать потребности пользователя (заказчика). *Пользовательские требования* (User Requirements) –

описывают цели/задачи пользователей системы, которые должны достигаться/выполняться пользователями при помощи создаваемой программной системы. Часто пользовательские требования представляют в виде сценариев (вариантов использования) Use Case (см. лабораторную работу №5).

Среди нефункциональных требований на первый план выходят атрибуты качества и ограничения. *Атрибуты качества (Quality Attributes)* описывают дополнительные характеристики продукта в различных «измерениях», важных для пользователей и/или разработчиков. Атрибуты касаются вопросов портируемости, интероперабельности (прозрачности взаимодействия с другими системами), целостности, устойчивости и т.п. Данный вид требований мы будем называть *спецификацией качества*. *Ограничения (Constraints)* включают в себя формулировки условий, модифицирующих требования или наборы требований, сужая выбор возможных решений по их реализации. В частности, к ним могут относиться параметры производительности, влияющие на выбор платформы реализации и/или развертывания (протоколы, серверы приложений, баз данных, ...), которые, в свою очередь, могут относиться, например, к внешним интерфейсам.

*Спецификация требований к ПО (SRS)* - процесс формализованного описания функциональных и нефункциональных требований, требований к характеристикам качества в соответствии со стандартом качества ISO/IEC 9126-94, которые будут отрабатываться на этапах ЖЦ ПО.

В спецификации требований отражается:

- структура ПО;
- требования к функциям, качеству и документации;
- задается в общих чертах архитектура системы и ПО, алгоритмы, логика управления и структуры данных.

#### **Характеристики правильно составленной спецификации требований к ПО**

Спецификация требований должна быть [15]:

- а) *Корректной* (каждое требование, изложенное в ней, является требованием, которому должно удовлетворять программное обеспечение);
- б) *Однозначной* (каждое изложенное в ней требование может интерпретироваться только однозначно. Как минимум, для этого требуется, чтобы каждая характеристика конечного продукта была описана с использованием одного уникального термина);
- с) *Полной* (все существенные требования должны быть подтверждены и обработаны любые внешние требования, налагаемые спецификацией системы; определены отклики

программного обеспечения на все классы входных данных, которые могут быть реализованы, во всех возможных ситуациях, как на допустимые, так и недопустимые входные значения; полные обозначения и ссылки на все рисунки, таблицы и схемы в SRS и определение всех терминов и единиц измерения);

- d) *Непротиворечивой* (никакой набор отдельных требований, описанных в ней, не находится в противоречии с ней или с каким-то документом более высокого уровня);
- e) *Упорядоченной* по ее значимости и/или устойчивости (каждое требование должно иметь идентификатор и относиться к определенному классу требований: необходимые, условные и необязательные);
- f) *Проверяемой* (каждое требование, изложенное в ней, может быть проверено.);
- g) *Модифицируемой* (структура и стиль SRS таковы, что любые изменения требований могут быть выполнены легко, полностью и непротиворечивым образом при сохранении структуры и стиля);
- h) *Отслеживаемой* (должен четко прослеживаться источник каждого из ее требований и SRS облегчает обращение к каждому из требований при дальнейшей разработке или модернизации документации).

В процессе работы с требованиями участвуют так называемые «заинтересованные лица»<sup>10</sup>, к числу которых мы будем относить:

- *Пользователей* (людей, кто будет непосредственно использовать ПО; пользователи могут описать задачи, которые они решают (планируют решать) с использованием ПС, а также ожидания по отношению к атрибутам качества, отображаемые в пользовательских требованиях, в этой роли выступает преподаватель);
- *Заказчиков* (людей, которые являются целевой аудиторией на рынке программного обеспечения, в этой роли выступает преподаватель);
- *Инженеров по программному обеспечению* (людей, ответственных за техническую оценку путей решения поставленных задач и последующую реализацию требований заказчиков, в этой роли выступают студенты - члены команды разработчиков).

Все заинтересованные лица должны прийти к соглашению о том, какая система должна быть создана, чтобы в ней воплотились необходимые дидактические и технологические свойства. Данные соглашения фиксируются в документе, с которым могут сверяться и на который могут ссылаться все участники проекта, - интегрированная спецификация требова-

---

<sup>10</sup> Заинтересованное лицо - некто, имеющий возможность (в том числе, материальную) повлиять на реализацию проекта/продукта

ний, созданная на основе совместного использования традиционной функциональной спецификации и спецификации качества системы.

**Комментарии к примеру.** В спецификации качества перечисляются основные требования по показателям качества:

- описывается уровень надежности ПС;
- формулируются требования по быстродействию;
- требования к разработке интерфейса и т.п.

*Функциональная спецификация* включает в себя:

- перечень всех функций системы с привязкой их к конкретной подсистеме и к информационной среде (входные и выходные данные), фрагмент функциональной спецификации приведен в таблице 1;
- перечень исключительных ситуаций и реакцию системы на их возникновение, при необходимости приводится перечень ошибок, которые могут возникать в системе и соответствующие им системные сообщения, примеры исключительных ситуаций приведены в таблице 2.

Функциональная спецификация должна в полном объеме отображать информационные связи проектируемой системы как с внешним миром, так и между подсистемами. При необходимости расписываются информационные связи для сложных подсистем (спецификация второго уровня).

*Исключительная ситуация* – это ситуация, при которой система не может выполнить возложенных на нее функций или которая может привести к денормализации работы системы.

Таблица 2 – Перечень исключительных ситуаций

Название подсистемы	Название исключительной ситуации	Реакция системы
1 Справочная	1.1 Не возможно открыть файл справки	Выдача сообщения «Файл справки поврежден»
	1.2 Не возможно найти файл справки	Выдача сообщения «Отсутствует файл справки»
2 Файловая	2.1 Попытка открытия файла с несобственным форматом	Выдача сообщения «Файл поврежден или недопустимого формата»
	2.2 Файл с заданным именем не существует	Выдача аналогичного сообщения
...	...	...

**МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНОМУ ПРАКТИКУМУ  
ПО ДИСЦИПЛИНЕ «ПРОГРАММНАЯ ИНЖЕНЕРИЯ»**

Таблица 1 – Перечень функций, выполняемых системой (фрагмент)

Название подсистемы	Название функции	Информационная среда			
		Входные данные		Выходные данные	
		Назначение (наименование)	Тип, ограничения	Назначение (наименование)	Тип, ограничения
1 Справочная	1.1 Выдать сведения о разработчиках	Сведения о разработчиках системы (ФИО, номер группы)	Текст (МЕМО)	Визуальное отображение информации	-
	1.2 Выдать сведения о системе	Файл справки	Текстовый (*.HTML)	Код ошибки	целое
2 Настройки параметров	2.1 Задать количество букв в пересечении	Диапазон количества букв: минимальное максимальное	Целое 1 3	Текущее значение букв в пересечении	Целое
	2.2 Подключить словарь понятий	Имя файла	Строка, *.dict	Список понятий и их определений	Динамический массив строк
				Код ошибки	Целое
	2.3 Задать длину кроссворда	Диапазон длин: минимальное максимальное	Целое 15 250	Текущее значение длины	Целое
				Код ошибки	Целое
	...	...	...	...	...
3 Файловая	3.1 Загрузить файл с кроссвордом	Имя файла	Строка, *.kros	Кроссворд	Объект, структура определяется в ходе проектирования
				Код ошибки	Целое

## ЛАБОРАТОРНАЯ РАБОТА № 6

### РАЗРАБОТКА ИНФОРМАЦИОННО-ЛОГИЧЕСКОГО ПРОЕКТА СИСТЕМЫ

Одной из широко используемых методик документирования требований является построение ряда моделей системы. Эти модели используют графические представления, показывающие решения как исходной задачи, так и разрабатываемой системы [16]. Как правило, графическое представление более понятно, чем описание требований на естественном языке.

*Моделирование* – это устоявшаяся и повсеместно принятая инженерная методика. Модели являются связующим звеном между процессом анализа и процессом проектирования системы. Что такое модель? *Модель* – это упрощенное представление реальности, по существу – это «чертеж» системы: в нее может входить как детальный план, так и более абстрактное представление системы «с высоты птичьего полета» [9]. Хорошая модель всегда включает элементы, которые существенно влияют на результат, и не включает те, которые малозначимы на данном уровне абстракции.

■ **Модель строится для того, чтобы лучше понимать разрабатываемую систему.**

Моделирование позволяет решить четыре различные задачи:

- 1 Визуализировать систему в ее текущем или желательном для нас состоянии;
- 2 Описать структуру или поведение системы;
- 3 Получить шаблон, позволяющий сконструировать систему;
- 4 Документировать принимаемые решения, используя полученные модели.

Моделирование предназначено не только для создания больших систем. От моделирования может выиграть любой проект. Даже при создании одноразовых программ, когда зачастую бывает полезно выбрать неподходящий код из-за преимущества в скорости разработки, которое дают языки визуального программирования, моделирование поможет коллективу разработчиков лучше представить план системы, а значит, выполнить проект быстрее и создать именно то, что подразумевал первоначальный замысел. Чем сложнее проект, тем более вероятно, что из-за отсутствия моделирования он свернется раньше времени – или будет создано не то, что нужно. Все полезные и интересные системы с течением времени обычно усложняются. Пренебрегая моделированием в самом начале создания системы можно серьезно пожалеть об этом, когда будет уже слишком поздно.

Моделирование имеет богатую историю во всех инженерных дисциплинах. Длительный опыт его использования позволил сформулировать *четыре основных принципа*.

- 1) Выбор модели оказывает определяющее влияние на подход к решению проблемы и на то, как будет выглядеть это решение (различные точки зрения на мир приводят к созданию различных систем, со своими преимуществами и недостатками).
- 2) Каждая модель может быть представлена с различной степенью точности (лучшей моделью будет та, которая позволяет выбрать уровень детализации в зависимости от того, кто и с какой целью на нее смотрит).
- 3) Лучшие модели – те, что ближе к реальности (поскольку модель всегда упрощает реальность, задача в том, чтобы это упрощение не повлекло за собой какие-то существенные потери).
- 4) Нельзя ограничиваться созданием только одной модели. Наилучший подход при разработке любой нетривиальной системы – использовать совокупность нескольких моделей, почти независимых друг от друга.

*Унифицированный язык моделирования* (Unified Modeling Language – UML) – это стандартный инструмент для разработки «чертежей» программного обеспечения. Его можно использовать для визуализации, спецификации, конструирования и документирования артефактов программных систем. UML подходит для моделирования любых систем – от информационных систем масштаба предприятия до распределенных Web-приложений и даже встроенных систем реального времени.

В лабораторном практикуме мы будем использовать UML для специфицирования (построения точных, недвусмысленных и полных моделей) системы и ее документирования. Язык UML предоставляет стандартный способ написания проектной документации на системы, включая концептуальные аспекты, такие как бизнес-процессы и функции системы, а также конкретные аспекты, такие как выражения языков программирования, схемы баз данных и повторно используемые компоненты ПО.

**Язык UML не является языком программирования (он инвариантен к языкам программирования).**

В нотации языка UML определены следующие виды канонических диаграмм:

- 1) вариантов использования (use case diagram);
- 2) классов (class diagram);
- 3) кооперации (collaboration diagram);
- 4) последовательности (sequence diagram);
- 5) состояний (statechart diagram);
- 6) деятельности (activity diagram);

- 7) компонентов (component diagram);
- 8) развертывания (deployment diagram).

Перечень этих диаграмм и их названия являются каноническими в том смысле, что представляют собой неотъемлемую часть графической нотации языка UML. Более того, процесс ООАП неразрывно связан с процессом построения этих диаграмм. При этом совокупность построенных таким образом диаграмм является самодостаточной в том смысле, что в них содержится вся информация, которая необходима для реализации проекта сложной системы. Каждая из этих диаграмм детализирует и конкретизирует различные представления о модели сложной системы в терминах языка UML.

Диаграмма вариантов использования представляет собой наиболее общую концептуальную модель сложной системы, которая является исходной для построения всех остальных диаграмм. Диаграмма классов, по своей сути, логическая модель, отражающая статические аспекты структурного построения сложной системы. Диаграммы кооперации и последовательностей представляют собой разновидности логической модели, которые отражают динамические аспекты функционирования сложной системы. Диаграммы состояний и деятельности предназначены для моделирования поведения системы. И, наконец, диаграммы компонентов и развертывания служат для представления физических компонентов сложной системы и поэтому относятся к ее физической модели.

Рассмотрим более подробно каждую из перечисленных диаграмм.

### **Диаграмма вариантов использования**

Визуальное моделирование с использованием нотации UML можно представить как процесс поуровневого спуска от наиболее общей и абстрактной концептуальной модели исходной бизнес-системы к логической, а затем и к физической модели соответствующей ПС. Вначале строится модель в форме так называемой диаграммы вариантов использования (*use case diagram*), которая описывает функциональное назначение системы и является исходным концептуальным представлением в процессе ее проектирования и разработки. На ней изображаются отношения между актерами и вариантами использования. Создание диаграммы вариантов использования имеет следующие цели:

- Определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы.
- Сформулировать общие требования к функциональному поведению проектируемой системы.



- Разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей.
- Подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями.

*Актёр* (actor) - согласованное множество ролей, которые играют внешние сущности по отношению к вариантам использования при взаимодействии с ними (это может быть любой объект, субъект или система, взаимодействующая с моделируемой бизнес-системой извне, т.е. человек, техническое устройство, программа и т.п.).

*Вариант использования* - внешняя спецификация последовательности действий, которые система или другая сущность могут выполнять в процессе взаимодействия с актерами (он определяет набор действий, совершаемый системой при диалоге с актером).

*Цель спецификации варианта использования* заключается в том, чтобы зафиксировать некоторый аспект или фрагмент поведения проектируемой системы без указания особенностей реализации данной функциональности. В этом смысле каждый вариант использования соответствует *отдельному сервису*, который предоставляет моделируемая система по запросу актера, т. е. определяет один из способов применения системы. Сервис, который инициализируется по запросу актера, должен представлять собой законченную последовательность действий. Это означает, что после того как система закончит обработку запроса актера, она должна возвратиться в исходное состояние, в котором снова готова к выполнению следующих запросов.

Диаграмма вариантов использования содержит конечное множество вариантов использования, которые в целом должны определять все возможные стороны ожидаемого поведения системы. Для удобства множество вариантов использования может рассматриваться как отдельный пакет. Применение вариантов использования на всех этапах работы над проектом позволяет не только достичь требуемого уровня унификации обозначений для представления функциональности подсистем и системы в целом, но и является мощным средством последовательного уточнения требований к проектируемой системе на основе их итеративного обсуждения со всеми заинтересованными специалистами.

**Комментарии к примеру.** На рисунке 4 приведен фрагмент диаграммы вариантов использования для разрабатываемой системы (он соответствует функциональной спецификации, приведенной в таблице 1, и дополняет ее новыми функциями).

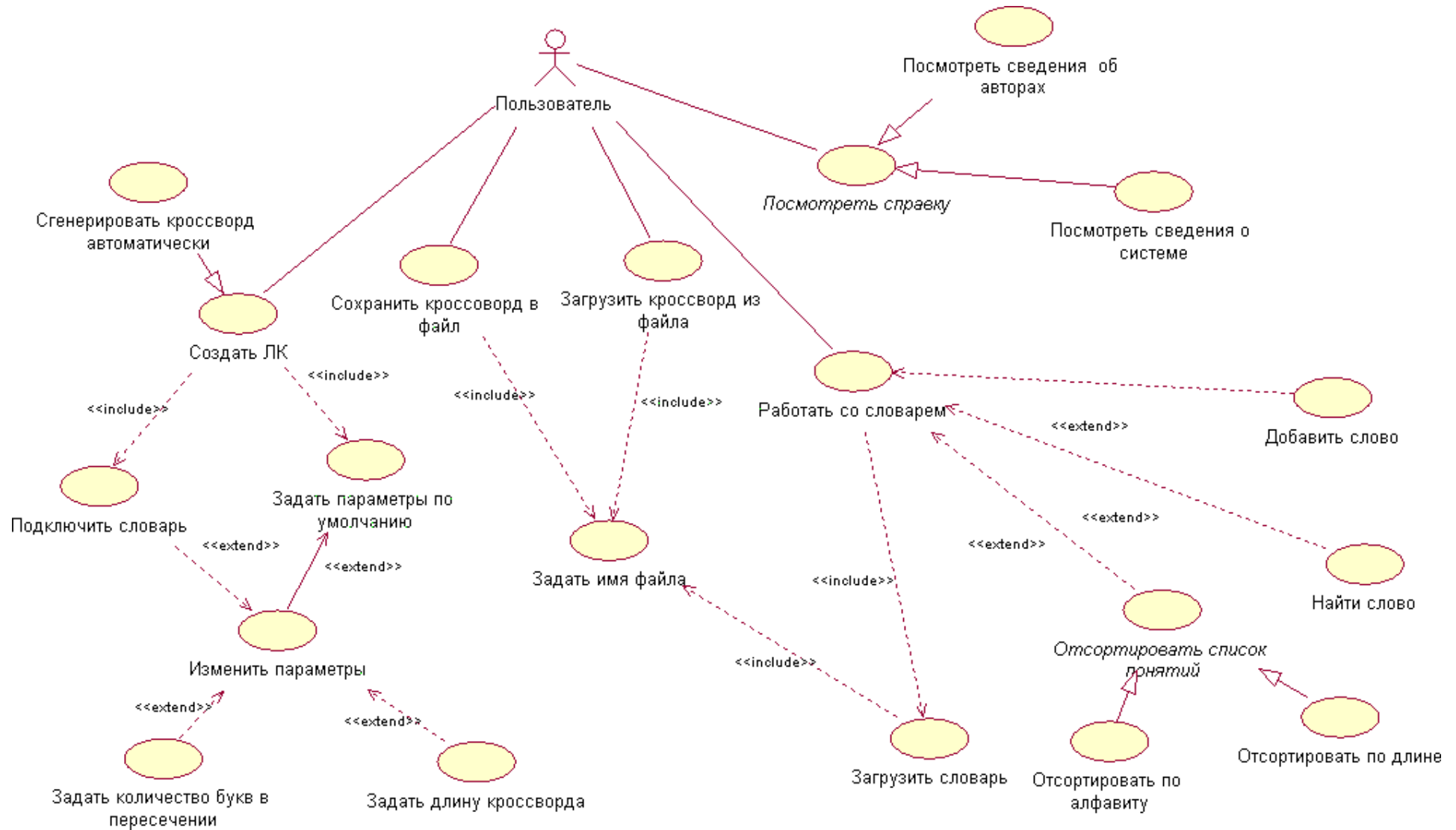


Рисунок 4 – Диаграмма вариантов использования системы (фрагмент)

Любая диаграмма, входящая в Ваш UML-проект должна быть прокомментирована: кратко описаны все основные сервисы, которые приведены на диаграмме. Например, «Пользователь системы должен иметь возможность посмотреть справочную информацию (сведения о разработчиках и сведения о самой системе); создавать кроссворд в автоматическом режиме, для этого он должен подключить словарь понятий и определить параметры кроссворда (выбрать значения, заданные по умолчанию, или изменить параметры); и т.д.»

### **Диаграмма классов**

Диаграммы классов – это наиболее часто используемый тип диаграмм, которые создаются при моделировании объектно-ориентированных систем, они показывают набор классов, интерфейсов и коопераций, а также их связи. На практике диаграммы классов применяют для моделирования статического представления системы, они служат основой для целой группы взаимосвязанных диаграмм – диаграмм компонентов и диаграмм размещения.

## ЛАБОРАТОРНАЯ РАБОТА № 7

### РАЗРАБОТКА АЛГОРИТМОВ ОБРАБОТКИ ДАННЫХ

*Выбор и обоснование алгоритмов* (или разработка и описание алгоритмов). Если для организации работы системы можно использовать уже известные алгоритмы, то необходимо провести их сравнительный анализ (по эффективности) и выбрать наилучший для данной системы (при введенных ограничениях). В противном случае пользователь разрабатывает свои алгоритмы, обосновывая их необходимость. Описание алгоритма ведется в вербальной форме и с помощью граф-схем алгоритмов /7/.

*Описание логической модели данных.* Если в проекте данные необходимо хранить в базе данных (БД), то на данном этапе должна быть разработана концептуальная и логическая модель БД, выделены и описаны основные сущности, определены между ними отношения. Модели должны быть представлены в соответствующей нотации (ER-модель (сущность - связь), SHM-модель (семантическую иерархическую модель) /3/). Переход к реляционной модели производится в соответствии с правилами, приведенными в /4/. Обязательным условием является нормализация реляционной модели информационной базы системы.

## 1 ЛАБОРАТОРНАЯ РАБОТА № 7

### РАЗРАБОТКА ПРОТОТИПА ИНТЕРФЕЙСА СИСТЕМЫ

*Физическое проектирование программной системы* - завершающий этап разработки системы. Он включает в себя:

- разработку пользовательского меню, которое должно быть ориентировано на структуру системы;
- описание интерфейса с обоснованием выбора того или иного стандарта оформления /1/.
- разработка модулей системы и описание их спецификаций, взаимодействие модулей должно быть представлено в виде диаграммы модулей с указанием иерархии модулей.

*Реализация проекта* и предъявление ПС (подсистемы) руководителю. Реализация проекта производится строго в соответствии с логическим проектом по технологии быстрой разработки приложений RAD (Rapid Application Development), в основе которой лежит спиральная модель жизненного цикла ПС, в определенной среде разработки, при необходимости используются дополнительные инструментальные средства (например, CASE-инструменты в виде специализированных пакетов и сред проектирования), производится автономная и комплексная отладка и тестирование. Руководитель проверяет полноту и качество реализации функций, соответствие системы техническому заданию и логическому проекту. Для демонстрации работоспособности системы необходимо подготовить нескольких тестовых примеров. При необходимости производится доработка реализации с повторным предъявлением системы, после доработки система выносится на защиту.

*Полное оформление документации проекта и защита проекта.* После приемки реализации студент оформляет пояснительную записку к ПС со всеми требуемыми приложениями.

## СОДЕРЖАНИЕ

1	Общие положения .....	4
2	Темы проектов .....	6
3	Задание на программную систему .....	7
4	Содержание курсового проекта .....	10
	Описание и анализ предметной области .....	10
	Проектирование системы .....	10
	Реализация системы .....	12
5	Оформление отчета .....	15
6	Список использованных источников .....	17
	Приложение А Пример оформления титульного листа .....	19
	Приложение Б Пример оформления реферата .....	19
	Приложение В Пример оформления технического задания на разработку ПС .....	20
	Приложение Г Структура содержания пояснительной записки .....	25

## 1 ОБЩИЕ ПОЛОЖЕНИЯ

Дисциплина «Методология программной инженерии» включена в учебный план направления подготовки магистров 090404 – «Программная инженерия» «Системы искусственного интеллекта».

Теоретические сведения по данной дисциплине изложены в курсе лекций [1] и закреплены студентами при выполнении лабораторного практикума [2]. Все это позволит студентам выполнить курсовой проект в соответствии с предъявляемыми требованиями.

Во время курсового проектирования должна быть разработана программная система среднего уровня сложности, тема курсового проекта должна соответствовать основным направлениям подготовки бакалавров по данной специальности. Разработка системы ведется коллективно. Команда разработчиков должна представлять собой группу студентов, имеющих опыт в анализе, проектировании, кодировании и тестировании программного обеспечения, способных хорошо взаимодействовать как внутри самой команды, так и с пользователями и/или заказчиками. В качестве заказчика и пользователя системы выступает преподаватель.

Курсовой проект по дисциплине «Программная инженерия» выполняется в 6 семестре, его цель - подготовить студентов к выполнению выпускной квалификационной работы (ВКР). Все разделы, включенные в курсовой проект, входят и в состав ВКР.

Тему курсового проекта выдает ведущий преподаватель (в дальнейшем - руководитель проекта) в течение первых двух недель семестра, в соответствии с ней студенты разрабатывают техническое задание по форме, описанной в разделе 3. Техническое задание в дальнейшем является основным документом, по которому студент ведет разработку проекта. Любые изменения технического задания на систему должны быть согласованы с руководителем и заверены его подписью.

В соответствии с техническим заданием студент в указанные сроки должен провести предпроектный анализ, концептуальное и логическое проектирование системы и представить руководителю полностью разработанную систему (или ее часть) (см. раздел 4). После проверки логического проекта и его утверждения студент выполняет его физическую реализацию проекта в заданной программной среде.

Документация по проекту ведется в соответствии со стандартом предприятия (СГАУ) [3] по всем стадиям проектирования, изложенным в разделе 4, она должна отражать

---

наиболее существенные стороны системы (подсистемы) и ее отличительные особенности, в том числе, в документации должны быть отражена структура системы, ее функциональные возможности и описание всех обеспечений системы: информационного, программного и технического, алгоритмического и т.п. Документация по проекту является основой *пояснительной записки*, которая предъявляется руководителю вместе с завершенным проектом (содержание пояснительной записки приведено в приложении Г).

Завершающая стадия проекта – его отладка и тестирование и сдача в эксплуатацию. Студенты предъявляет руководителю проекта для проверки на ЭВМ завершенную реализацию системы (подсистемы) и при необходимости производят ее доработку. После приемосдаточных испытаний студенты делают *презентацию* своей системы и вместе с пояснительной запиской защищают проект.

## **4 СОДЕРЖАНИЕ КУРСОВОГО ПРОЕКТА**

В рамках курсового проекта группа студентов (в дальнейшем *команда*) разрабатывает программную систему или подсистему (в дальнейшем *проект*), причем каждый студент отвечает за полную проработку своей части проекта. Проектирование системы (подсистемы) производится с применением технологии быстрой разработки приложений (RAD – Rapid Application Development) [4], которая поддерживается методологией *структурного проектирования* и включает элементы *объектно-ориентированного проектирования* и анализа предметной области.

Рассмотрим основные этапы работы над проектом.

### **Описание и анализ предметной области**

1. На данном этапе выделяются основные объекты, участвующие в функционировании системы, определяются их наиболее существенные характеристики, взаимосвязи в рамках решаемой задачи, а также определяются основные информационные потоки в системе. При необходимости приводятся различного рода *классификации*, используемые математические модели, базовые *термины* и *определения*, делается *обзор существующих систем-аналогов* с указанием их отличительных особенностей, достоинств и недостатков (приводятся экранные формы этих систем). Описание предметной области позволяет разработчику определить основные объекты и связи между ними, которые необходимо будет реализовать в проекте. Результатом данного шага является диаграмма объектов предметной области и краткое описание их свойств и атрибутов. Заключительным шагом данного этапа является развернутая формулировка задания (*постановка задачи*). Постановка задачи пишется в повествовательной форме на основе технического задания, она должна обязательно отразить все разделы



---

ТЗ, особенное внимание необходимо уделить подробному описанию функций, которые должна выполнять система.

### Проектирование системы

Проектирование системы – самый сложный и продолжительный этап, который начинается с разработки структурной схемы системы. На данном этапе используется методология структурного проектирования, в соответствии с которой система по функциональному признаку разделяется на основные подсистемы, между ними указываются информационные связи и/или связи по управлению, описывается основное назначение подсистем [4]. При необходимости структура подсистем детализируется до уровня, достаточного для понимания реализации основной функциональности системы. После того, как определена структура системы, преподаватель осуществляет распределение задач, которые будут выполнять студенты, входящие в команду.

Следующим шагом при выполнении работ является разработка *спецификаций* для системы целиком и каждой подсистемы в отдельности [2].

Разработка *спецификации качества*, в которой перечисляются основные требования по показателям качества:

- 1) описывается уровень надежности ПС,
- 2) формулируются требования по быстродействию;
- 3) требования к разработке интерфейса и т.п.

Разработка *функциональной спецификации системы*, которая включает в себя:

- 1) Перечень всех функций системы с привязкой их к конкретной подсистеме и к информационной среде;
- 2) Перечень исключительных ситуаций и реакцию системы на их возникновение, при необходимости приводится перечень ошибок, которые могут возникать в системе и соответствующие им системные сообщения;

Функциональная спецификация должна в полном объеме отображать информационные связи проектируемой системы как с внешним миром, так и между подсистемами. При необходимости расписываются информационные связи для сложных подсистем (спецификация второго уровня).

Одновременно с функциональной спецификацией студенты проектируют интерфейс пользователя: разрабатывают основные и вспомогательные экранные формы, описывают диалог с пользователем.

На основании спецификации производится разработка информационно-логического проекта с использованием *методологии UML* (Unified Modeling Language) [5, 6]. UML позволяет специфицировать все существенные решения, касающиеся анализа, проектирования и

---

реализации, которые должны приниматься в процессе разработки и развертывания системы программного обеспечения [6]. В *UML-проект* входит определенный набор канонических взаимосвязанных диаграмм (моделей), необходимых для понимания системы в целом. При необходимости детализируется поведение отдельных подсистем.

Проектирование начинается с *диаграммы вариантов использования* (Use case), которая специфицирует поведение системы или ее части и представляет собой описание множества последовательностей действий (включая варианты), выполняемых системой для того, чтобы пользователь мог получить определенный результат [6]<sup>1</sup>. После этого переходя к построению *диаграммы классов*. Классы - это самые важные строительные блоки любой объектно-ориентированной системы, с их помощью описывают программные, аппаратные или чисто концептуальные сущности. Это могут быть абстракции, являющиеся частью предметной области, либо классы, на которые опирается реализация. Далее разрабатываются диаграммы, характеризующие поведение системы (состояний, последовательности, деятельности, кооперации), они позволяют моделировать динамические аспекты системы.

#### **Замечание**

Оформленный логический проект предъявляется руководителю проекта в сроки, установленные заданием, и, после его корректировки, принимается в качестве основы для реализации.

Если в системе данные необходимо хранить в базе данных (БД), то на данном шаге должна быть разработана концептуальная и/или логическая модель БД, выделены и описаны основные сущности, определены между ними отношения. Модели должны быть представлены в соответствующей нотации, в частности должна быть разработана ER-модель (сущность-связь) [8]. Переход к реляционной модели производится в соответствии с правилами, приведенными в [9]. Обязательным условием является нормализация реляционной модели информационной базы системы.

Заключительным шагом данного этапа является выбор и обоснование комплекса программных средств, в первую очередь языка программирования и среды разработки, операционной системы, инструментальных средств проектирования, поддерживающих методологию UML, при необходимости выбор системы управления БД.

#### **Реализация системы**

После окончания процесса проектирования можно переходить к разработке информационного и программного обеспечения системы. Для этого необходимо использовать средства разработки, выбранные на этапе проектирования (обычно это различные фреймворки (framework), RAD-студии и другие средства быстрой разработки приложений). Наиболее популярным языком разработки является C#, поэтому в качестве среды разработки студенты

---

чаще всего выбирают Visual Studio (2009 и выше) и платформу .NET, которые в производственной среде поддерживаются в выпущенных версиях Windows и по которым предоставляется техническая поддержка. Кроме того, Visual Studio предоставляет поддержку для интеграции данных в приложение, что так же важно при разработке учебных задач (средства Visual Database Tools помогают создавать и поддерживать базы данных, а также управлять приложениями, работающими с данными).

Разработка интерфейса пользователя (прототипы экранных форм) была выполнена на этапе проектирования, программная реализация должна включать в себя разработку пользовательского меню, которое должно быть ориентировано на структуру системы; выбор элементов управления; средства контроля ввода данных и т.п.

В пояснительной записке необходимо описать структуру меню и последовательность работы с системой на контрольном примере (*контрольный (тестовый) пример* – это описание последовательности выполнения действий пользователя при решении наиболее важной системной задачи).

Диаграммы, вошедшие в UML-проект, на данном этапе дополняются *диаграммами реализации* (диаграммы компонентов и развертывания), которые используются для моделирования статического вида системы с точки зрения реализации и развертывания: моделирование физических сущностей, развернутых в узле, например, исполняемых программ, библиотек, таблиц, файлов и документов. По существу, диаграммы компонентов и развертывания – это не что иное, как диаграммы классов, сфокусированные на системных компонентах и узлах [6]. Включение диаграммы развертывания в проект обязательно для систем с клиент-серверной архитектурой или полностью распределенных систем, т.к. они позволяют моделировать топологию таких систем.

Если в системе данные хранятся в базе данных (реляционной, объектно-ориентированной или гибридной объектно-реляционной), то необходимо преобразовать логическую схему в физическую (это лучше делать с помощью инструментальных средств, в том числе поддерживающих методологию UML). При физической реализации необходимо учитывать ограничения, накладываемые предметной областью, чтобы избежать избыточности в хранимых данных. Если в структура БД полностью соответствует диаграмме сущностных классов, то физическую модель данных можно не описывать, только провести необходимые расчеты для определения необходимого объема дискового пространства и оперативного запоминающего устройства, которые в свою очередь будут влиять на требования, предъявляемые к комплексу технических средств.

Программная реализация проекта предъявляется руководителю после проведения разработчиками автономной и комплексной отладки и тестирования. Руководитель проверяет

---

полноту и качество реализации функций, соответствие системы техническому заданию и логическому проекту. Для демонстрации работоспособности системы необходимо подготовить нескольких тестовых примеров. При необходимости производится доработка реализации с повторным предъявлением системы, после доработки система выносится на защиту.

После приемки реализации студенты оформляет пояснительную записку к ПС со всеми требуемыми приложениями и делают презентацию, с которой публично выступают перед учебной группой(ами).

#### **Замечание**

Если в процессе реализации возникает необходимость изменения структурных схем, диаграмм, логики процессов, то по согласованию с руководителем логический проект корректируется и переоформляется. Все отступления от утвержденного задания согласовываются с руководителем проекта и заверяются его личной подписью. При наличии большого числа корректировок (3 и более) задание переоформляется и утверждается вновь.

**Вариант 1**  
**Тест по дисциплине «Программная инженерия»**  
 Студент \_\_\_\_\_ группы \_\_\_\_\_

<b>Вопрос 1</b>	<b>Выберите правильное определение понятия «Программирование»</b>	<b>Балл</b>
1.	Процесс кодирования и отладки программы в рамках реального проекта	
2.	Все технические операции, необходимые для создания программы, включая анализ требований и все стадии разработки и реализации	
3.	Процесс написания спецификации программной системы	
<b>Вопрос 2</b>	<b>В какие годы сформировалась концепция объектно-ориентированного программирования?</b>	<b>Балл</b>
1.	В 50-е годы XX века	
2.	В 60-е годы XX века	
3.	В 70-е годы XX века	
4.	В 80-е годы XX века	
<b>Вопрос 3</b>	<b>Какие IT-проекты могут считаться успешными?</b> а) Если проект выпущен надлежащего качества б) Если проект выпущен вовремя в) Если проект интересным потенциальным пользователям г) Если расходы соответствуют изначальному бюджету	<b>Балл</b>
1.	а) и б)	
2.	а) б) в) г)	
3.	в) и г)	
4.	а) и в)	
<b>Вопрос 4</b>	<b>Требования к программному обеспечению – это</b>	<b>Балл</b>
1.	Общие свойства, которыми должно обладать ПО	
2.	Совокупность атрибутов, свойств или качеств ПО	
3.	Свойства, которыми должно обладать ПО для адекватного определения функций, условий и ограничений выполнения ПО, а также объемов данных, технического обеспечения и среды функционирования	
<b>Вопрос 5</b>	<b>Системные требования (system requirements) к программному обеспечению</b>	<b>Балл</b>
1.	определяют функциональность ПО, которую разработчики должны построить, чтобы пользователи смогли выполнить свои задачи в рамках бизнес-требований	
2.	определяют высокоуровневые цели организации или клиента (потребителя) - заказчика разрабатываемого ПО	
3.	Это высокоуровневые требования к ПО, содержащему несколько или много взаимосвязанных подсистем и приложений	
<b>Вопрос 6</b>	<b>Что такое предметная область?</b>	<b>Балл</b>
1.	Та часть реального мира, которая имеет существенное значение или непосредственное отношение к процессу функционирования программы	
2.	Некоторая абстрактная единица, которая обладает функциональностью, т. е. может выполнять определенные действия, связанные с решением поставленных задач	
3.	Это часть программного кода, которая предполагает построение программного обеспечения (ПО) из отдельных компонентов - физически отдельно существующих частей ПО, взаимодействующих между собой	
4.	Это набор «интеллектуальных пакетов программ»	

<b>Вопрос 7</b>	<b>Выберите наиболее полный перечень основных принципов объектной модели</b>	<b>Балл</b>
1.	Инкапсуляция, наследование	
2.	Инкапсуляция, наследование, полиморфизм	
3.	Абстрагирование, инкапсуляция, наследование, полиморфизм	
4.	Типизация, модульность, иерархичность	
<b>Вопрос 8</b>	<b>В определение какого показателя качества включается необходимость учета структуры программы и семантики языка программирования?</b>	<b>Балл</b>
1.	Корректность	
2.	Эффективность	
3.	Надежность	
4.	Отлаживаемость	
<b>Вопрос 9</b>	<b>Какие критерии качества программной системы являются для нее обязательными?</b>	<b>Балл</b>
1.	Функциональность и надежность	
2.	Функциональность и эффективность	
3.	Удобочитаемость и переносимость	
4.	Переносимость и полезность	
<b>Вопрос 10</b>	<b>Какие стандарты в разработке ПО носят рекомендательный характер?</b>	<b>Балл</b>
1.	Корпоративные	
2.	Отраслевые	
3.	Международные	
<b>Вопрос 11</b>	<b>Выберите наиболее точное определение жизненного цикла программного обеспечения</b>	<b>Балл</b>
1.	Непрерывный процесс, который начинается с момента принятия решения о необходимости создания программного обеспечения и заканчивается в момент его полного изъятия из эксплуатации	
2.	Набор взаимосвязанных работ, которые преобразуют исходные данные в выходные результаты	
3.	Процесс определения того, отвечает ли текущее состояние разработки, достигнутое на данном этапе, требованиям этого этапа	
4.	Набор фаз (этапов, стадий) проекта по созданию ПО, в которых выполняются отдельные процессы, разбитые на операции и задачи	
<b>Вопрос 12</b>	<b>Показатель качества (программной системы):</b>	<b>Балл</b>
1.	Набор метрик, с помощью которых оценивают взаимодействие программного обеспечения с его окружением	
2.	Характеристика качества программной системы, обладающая количественным значением	
3.	Степень удовлетворения потребностей, представленная посредством конкретного набора значений характеристик качества программной системы	
<b>Вопрос 13</b>	<b>Ромб в блок-схеме обозначает</b>	<b>Балл</b>
1.	условие («решение»)	
2.	ввод данных	
3.	вывод данных	
4.	вычисление	

<b>Вопрос 14</b>	<b>В каскадной модели на стадии разработки проекта выполняются следующие действия</b>	<b>Балл</b>
1.	Разрабатывается и формулируется логически последовательная техническая характеристика программной системы, включая структуры данных, архитектуру ПО, интерфейсные представления и процессуальную (алгоритмическую) детализацию	
2.	Определяются программные требования для предметной области системы, предназначение, линии поведения, производительность и интерфейсы	
3.	Происходит исследование требований, разрабатывается видение продукта и оценивается возможность его реализации	

<b>Вопрос 15</b>	<b>Верификация требований -это</b>	<b>Балл</b>
1.	Руководство процессами формирования требований на всех этапах ЖЦ и включает управление изменениями и атрибутами требований	
2.	Процесс проверки изложенных в спецификации требований, выполняющаяся для того, чтобы путем отслеживания источников требований убедиться, что они определяют именно данную систему	
3.	Процесс проверки правильности спецификаций требований на их соответствие, непротиворечивость, полноту и выполнимость, а также на соответствие стандартам	

<b>Вопрос 16</b>	<b>На фазе кодирования ПО активно используются следующие техники:</b> а) создания легко понимаемого исходного кода на основе использования соглашений об именовании, форматирования и структурирования кода; б) использование классов, перечисляемых типов, переменных, именованных констант и других выразительных сущностей; в) обработка ошибочных условий и исключительных ситуаций; г) восстановлении спецификации (графов вызовов, потоков данных и др.) по полученному коду системы для наблюдения за ней на более высоком уровне	<b>Балл</b>
1.	а) б) д)	
2.	б) в) д)	
3.	а) б) в) д)	
4.	а) б) в)	

<b>Вопрос 17</b>	<b>Какие характеристики качества в модель качества МакКола связаны с ревизиями ПО?</b>	<b>Балл</b>
1.	Корректность, надежность, функциональность	
2.	Сопровождаемость, оцениваемость, гибкость	
3.	Переиспользуемость, переносимость, способность к взаимодействию	

<b>Вопрос 18</b>	<b>При использовании какого метода тестирования код программы доступен тестировщикам?</b>	<b>Балл</b>
1.	Модульное тестирование	
2.	«Белого ящика»	
3.	«Черного ящика»	
4.	«Серого ящика»	

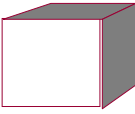

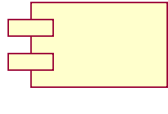
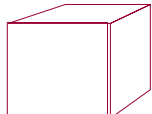
<b>Вопрос 19</b>	<b>Какой вид тестирования ориентирован на повторное выборочное тестирование системы или ее компонентов после внесения в них изменений на тех же тестах, что и до модификации?</b>	<b>Балл</b>
1.	Интеграционное тестирование	
2.	Регрессионное тестирование	
3.	Функциональное тестирование	
4.	Стресс-тестирование (нагрузочное)	

<b>Вопрос 20</b>	<b>Какие процессы ЖЦ ПО не относятся к основным</b>	<b>Балл</b>
1.	Сопровождение	
2.	Разработка	
3.	Эксплуатация	
4.	Обучение	

<b>Вопрос 21</b>	<b>На какой стадии разработки применяют варианты использования?</b>	<b>Балл</b>
1.	Сопровождения ПО	
2.	Проектирования ПО	
3.	Тестирования ПО	

<b>Вопрос 22</b>	<b>Какие модели ЖЦ в международных стандартах обычно квалифицируются как фундаментальная?</b>	<b>Балл</b>
1.	Каскадная	
2.	V-образная	
3.	Модель быстрого прототипирования	
4.	Модель жизненного цикла MSF	

<b>Вопрос 23</b>	<b>Кто не может быть актером в Use Case-диаграммах?</b>	<b>Балл</b>
1.	Типовой пользователь системы	
2.	Внешняя система, взаимодействующие с заданной	
3.	Отдельная (внутренняя) часть системы	
4.	Выделенный пользователь	

<b>Вопрос 24</b>	<b>Стандартным графическим обозначением устройства на диаграммах развертывания является</b>				<b>Балл</b>
	1.	2.	3.	4.	
					

<b>Вопрос 25</b>	<b>Стандартным графическим обозначением отношения обобщения на диаграммах использования является</b>	<b>Балл</b>
1.	сплошной линией между актером и вариантом использования	
2.	пунктирной линией со стрелкой между актером и вариантом использования, которая помечается ключевым словом «extend»	
3.	сплошной линией со стрелкой в форме незакрашенного треугольника	
4.	пунктирной линией со стрелкой между актером и вариантом использования, которая помечается ключевым словом «include»	

Итоговая оценка \_\_\_\_\_ (по 5-балльной шкале) ) \_\_\_\_\_



**Вариант 2**  
**Тест по дисциплине «Программная инженерия»**

**Студент \_\_\_\_\_ группы \_\_\_\_\_**

<b>Вопрос 1</b>	<b>Какой комитет занимается стандартизацией в области программной инженерии?</b>	<b>Балл</b>
1.	Международная организация по стандартизации (ИСО, ISO)	
2.	Международная электротехническая комиссия (МЭК, IEC)	
3.	Ни одна из этих организаций	
4.	Объединенный технический комитет (обе эти организации (ISO/IEC))	

<b>Вопрос 2</b>	<b>Дайте определение парадигмы программирования</b>	<b>Балл</b>
1.	Это совокупность идей и понятий, определяющая стиль написания программ	
2.	Это новая модель конструирования программ и взаимодействия ее с данными	
3.	Это семейство обозначений (нотаций), разделяющих общий способ (методику) реализаций программ	
4.	Это способ размышления о компьютерных системах	

<b>Вопрос 3</b>	<b>Какие причины могут привести к неудачам в реализации IT-проектов?</b> а) нет квалифицированных кадров или их недостаточное количество б) размыты границы выполнения проекта с) проект не интересен потенциальным пользователям д) перерасход средств на разработку проекта	<b>Балл</b>
1.	а) и б)	
2.	а) б) с) д)	
3.	с) и д)	
4.	а) и с)	

<b>Вопрос 4</b>	<b>Объектно-ориентированная декомпозиция заключается в следующем</b>	<b>Балл</b>
1.	Декомпозиция рассматривается как обычное разделение алгоритмов, где каждый модуль системы выполняет один из этапов общего процесса	
2.	Система представлена совокупностью автономных действующих лиц, которые взаимодействуют друг с другом, чтобы обеспечить поведение системы, соответствующее более высокому уровню	
3.	Декомпозиция представлена объектом-экземпляром класса, который состоит из методов объекта и его свойств	
4.	Декомпозиция показывает развитие систем в процессе их эволюции и построена на механизме наследования свойств	

<b>Вопрос 5</b>	<b>Принцип «разделяй и властвуй» заключается в следующем</b>	<b>Балл</b>
1.	Принцип организации составных частей проблемы в иерархические древовидные структуры с добавлением новых деталей на каждом уровне	
2.	Принцип решения сложных проблем путем их разбиения на множество меньших независимых задач, легких для понимания и решения	
3.	Выделение существенных аспектов системы и отвлечения от несущественных	

<b>Вопрос 6</b>	<b>Технология модульного программирования сформировалась в</b>	<b>Балл</b>
1.	в середине 80-х годов XX века	
2.	в начале 50-х годов XX века	
3.	в начале 90-х годов XX века	
4.	в 70-х годах XX века	

<b>Вопрос 7</b>	<b>Компонентное программирование это развитие идеологии</b>	<b>Балл</b>
1.	функционального программирования	
2.	структурного программирования	
3.	объектно-ориентированной технологии	
4.	логического программирования	
<b>Вопрос 8</b>	<b>Прямоугольник в блок-схеме обозначает</b>	<b>Балл</b>
1.	условие («решение»)	
2.	ввод данных	
3.	вывод данных	
4.	вычисление	
<b>Вопрос 9</b>	<b>Инженеры в программной инженерии - это специалисты, которые</b>	<b>Балл</b>
1.	Разрабатывают и внедряет методы автоматизации программирования, типовые и стандартные программы, программирующие программы, трансляторы, входные алгоритмические языки	
2.	Выполняют практические работы по реализации программ с применением теории, методов и средств компьютерных наук	
3.	Определяет вводимую в машину информацию, ее объем, методы контроля производимых машинной операций, форму и содержание исходных документов и результатов вычислений	
<b>Вопрос 10</b>	<b>Выберите наиболее точно определение агрегации. Это</b>	<b>Балл</b>
1.	Процесс объединения элементов в одну систему	
2.	Включение объектов одного класса в состав другого класса	
3.	Отношение «часть-целое» между двумя равноправными объектами, когда один объект (контейнер) имеет ссылку на другой объект. Оба объекта могут существовать независимо: если контейнер будет уничтожен, то его содержимое — нет	
4.	Отношение «часть-целое» между двумя объектами, когда один объект (контейнер) имеет ссылку на другой объект, при этом включаемый объект может существовать только как часть контейнера	
<b>Вопрос 11</b>	<b>Какие требования описывает такие характеристики системы, как надежность, особенности поставки, определенный уровень качества (атрибуты качества)?</b>	<b>Балл</b>
1.	Системные	
2.	Пользовательские	
3.	Нефункциональные	
<b>Вопрос 12</b>	<b>Пользовательские требования (User requirements) к программному обеспечению</b>	<b>Балл</b>
1.	описывают цели/задачи пользователей системы, которые должны достигаться/выполняться пользователями при помощи создаваемого ПО	
2.	определяют высокоуровневые цели организации или клиента (потребителя) - заказчика разрабатываемого ПО	
3.	Это высокоуровневые требования к ПО, содержащему несколько или много взаимосвязанных подсистем и приложений	
<b>Вопрос 13</b>	<b>Выберите определение показателя качества: «Способность программной системы выполнять возложенные на нее функции при поступлении требований на их выполнение»?</b>	<b>Балл</b>
1.	Корректность	
2.	Эффективность	
3.	Надежность	
4.	Функциональность	

<b>Вопрос 14</b>	<b>Жизненный цикл проекта это</b>	<b>Балл</b>
1.	Набор обычно последовательных фаз проекта, количество и состав которых определяется потребностями управления проектом организацией или организациями, участвующими в проекте	
2.	Набор взаимосвязанных ресурсов и работ, благодаря которым входные воздействия преобразуются в выходные результаты	
3.	Элемент работ проекта	

<b>Вопрос 15</b>	<b>Для каскадной модели неверно следующее утверждение</b>	<b>Балл</b>
1.	каждая последующая фаза может начинаться после реализации половины работ предыдущей фазы	
2.	каждая последующая фаза начинается лишь тогда, когда полностью завершено выполнение предыдущей фазы	
3.	каждая фаза полностью документируется	




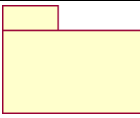
<b>Вопрос 16</b>	<b>FP-дизайн (FP-design, family pattern design) - это</b>	<b>Балл</b>
1.	декомпозиция структуры программного обеспечения в виде набора фрагментов или компонент	
2.	семейство архитектурных представлений, базирующихся на шаблонах	
3.	создание высоко-уровневой концепции, видения того, что из себя будет представлять программная система; данный вид дизайна является результатом процесса анализа требований и их трансформации в подходы к реализации	

<b>Вопрос 17</b>	<b>Какие техники предназначены для обеспечения качества кода (на фазе его конструирования)?</b>	<b>Балл</b>
1.	Разработка кода с первичностью тестов	
2.	Техники определения рисков	
3.	Использование основных положений стандарта ISO/IEC 12207	

<b>Вопрос 18</b>	<b>Какой метод тестирования заключается в проверке отдельных, изолированных и независимых частей ПО?</b>	<b>Балл</b>
1.	Модульное тестирование	
2.	Интеграционное тестирование	
3.	Метод «Черного ящика»	
4.	Системное тестирование	

<b>Вопрос 19</b>	<b>Модель жизненного цикла - это</b>	<b>Балл</b>
1.	Совокупность процессов, работ и задач, включающих в себя разработку, эксплуатацию и сопровождение ПП,	
2.	Структура, состоящая из процессов, работ и задач, включающих в себя разработку, эксплуатацию и сопровождение ПП, охватывающая жизнь системы от установления требований к ней до прекращения ее использования	
3.	непрерывный процесс, который начинается с момента принятия решения о необходимости его создания и заканчивается в момент его полного изъятия из эксплуатации	

<b>Вопрос 20</b>	<b>Когда стала использоваться при разработке ПО каскадная модель ЖЦ?</b>	<b>Балл</b>
1.	В 70-х годах XX века	
2.	В 80-х годах XX века	
3.	В 90-х годах XX века	

<b>Вопрос 21</b>	<b>Стандартным графическим обозначением актера на диаграммах использования является</b>				<b>Балл</b>
	1.	2.	3.	4.	
					

Вопрос 22	Какая диаграмма используется для моделирования аппаратной части системы, с которой связано ПО?	Балл
1.	Диаграмма вариантов использования	
2.	Диаграмма развертывания	
3.	Диаграмма компонентов	
4.	Диаграмма состояний	

Вопрос 23	Стандартным графическим обозначением отношения расширения на диаграммах использования является	Балл
1.	сплошной линией между актером и вариантом использования	
2.	пунктирной линией со стрелкой между актером и вариантом использования, которая помечается ключевым словом «extend»	
3.	сплошной линией со стрелкой в форме незакрашенного треугольника	
4.	пунктирной линией со стрелкой между актером и вариантом использования, которая помечается ключевым словом «include»	

<u>Вопрос 24</u>	Каскадная модель неприемлема при	Балл
1.	Разработке новой версии уже существующего ПО	
2.	Решении задач научно-вычислительного характера	
3.	Повторной разработке типового продукта	
4.	При разработке нового сложного плохо формализованного ПО	

<u>Вопрос 25</u>	Какое утверждение неверно (относительно применимости спиральной модели)	Балл
1.	Она предполагает возможность эволюции жизненного цикла, развитие и изменение программного продукта	
2.	Она позволяет контролировать источники проектных работ и соответствующих затрат.	
3.	Она проводит различия между разработкой нового продукта и расширением (или сопровождением) существующего	
4.	Она позволяет решать интегрированные задачи системной разработки	

Итоговая оценка \_\_\_\_\_ (по 5-балльной шкале) \_\_\_\_\_

**Вариант 3**  
**Тест по дисциплине «Программная инженерия»**  
**Студент \_\_\_\_\_ группы \_\_\_\_\_**

<b>Вопрос 1</b>	<b>Какой комитет занимается разработкой стандартов по радиоэлектронике и электротехнике?</b>	<b>Балл</b>
1.	Международная организация по стандартизации (ИСО, ISO)	
2.	Международная электротехническая комиссия (МЭК, IEC)	
3.	Ни одна из этих организаций	
4.	Институт инженеров по электронике и электротехнике (IEEE)	
<b>Вопрос 2</b>	<b>Программная инженерия (Software Engineering) является отраслью</b>	<b>Балл</b>
1.	Информатики (computer science)	
2.	Системотехники (system engineering)	
3.	Системного программирования (system )	
<b>Вопрос 3</b>	<b>Основными принципами какой парадигмы программирования являются инкапсуляция, наследование, полиморфизм?</b>	<b>Балл</b>
1.	Модульное программирование	
2.	Объектно-ориентированное программирование	
3.	Процедурно-ориентированное программирование	
4.	Логическое программирование	
<b>Вопрос 4</b>	<b>В основе структурного подхода к разработке программных систем лежит</b>	<b>Балл</b>
1.	Алгоритмическая декомпозиция	
2.	Объектно-ориентированная декомпозиция	
3.	Функциональная декомпозиция	
4.	Процедурная декомпозиция	
<b>Вопрос 5</b>	<b>Функциональные требования (functional requirements) к программному обеспечению</b>	<b>Балл</b>
	определяют функциональность ПО, которую разработчики должны построить, чтобы пользователи смогли выполнить свои задачи в рамках бизнес-требований	
	Это по существу спецификация вариантов использования ПО	
	Это высокоуровневые требования к ПО, содержащему несколько или много взаимосвязанных подсистем и приложений	
<b>Вопрос 6</b>	<b>Требования к программному обеспечению – это</b>	<b>Балл</b>
1.	Общие свойства, которыми должно обладать ПО	
2.	Совокупность атрибутов, свойств или качеств ПО	
3.	Свойства, которыми должно обладать ПО для адекватного определения функций, условий и ограничений выполнения ПО, а также объемов данных, технического обеспечения и среды функционирования	
<b>Вопрос 7</b>	<b>К какому показателю качества относятся время выполнения кода, загруженность процессора, объем требуемой памяти, время отклика и т.п.?</b>	<b>Балл</b>
1.	Тестируемость	
2.	Эффективность	
3.	Надежность	
4.	Мобильность	

<b>Вопрос 8</b>	<b>Какие стандарты при разработке имеют силу закона?</b>	<b>Балл</b>
1.	Корпоративные	
2.	Отраслевые	
3.	Государственные	
4.	Международные	
<b>Вопрос 9</b>	<b>Жизненный цикл проекта это</b>	<b>Балл</b>
4.	Набор обычно последовательных фаз проекта, количество и состав которых определяется потребностями управления проектом организацией или организациями, участвующими в проекте	
5.	Набор взаимосвязанных ресурсов и работ, благодаря которым входные воздействия преобразуются в выходные результаты	
6.	Элемент работ проекта	
<b>Вопрос 10</b>	<b>В блок-схеме связи обозначаются (при лексиграфическом обходе)</b>	<b>Балл</b>
1.	стрелочками	
2.	линиями	
3.	прямоугольниками	
4.	кружочками	
<b>Вопрос 11</b>	<b>I-дизайн (I-design, invention) - это</b>	<b>Балл</b>
1.	декомпозиция структуры программного обеспечения в виде набора фрагментов или компонент	
2.	семейство архитектурных представлений, базирующихся на шаблонах	
3.	создание высоко-уровневой концепции, видения того, что из себя будет представлять программная система; данный вид дизайна является результатом процесса анализа требований и их трансформации в подходы к реализации	
<b>Вопрос 12</b>	<b>Выберите неверное утверждение</b>	<b>Балл</b>
1.	Диаграмма вариантов использования используется для определения общих границ и контекста моделируемой предметной области на начальных этапах проектирования системы	
2.	Диаграмма вариантов использования используется для специфицирования требуемого поведения субъекта.	
3.	Диаграмма вариантов использования используется как исходная концептуальная модель системы для ее последующей детализации в форме логических и физических моделей	
4.	Диаграмма вариантов использования определяет внутреннюю структуру сущностей разрабатываемой системы	
<b>Вопрос13</b>	<b>Какие характеристики качества в модель качества МакКола связаны с функционированием ПО?</b>	<b>Балл</b>
1.	Корректность, надежность, функциональность	
2.	Сопровождаемость, оцениваемость, гибкость	
3.	Переиспользуемость, переносимость, способность к взаимодействию	
<b>Вопрос 14</b>	<b>Какой метод тестирования позволяет проверить связи и способы взаимодействия (интерфейсов) компонентов друг с другом?</b>	<b>Балл</b>
1.	Модульное	
2.	Интеграционное тестирование	
3.	Метод «Черного ящика»	
4.	Системное тестирование	



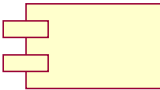
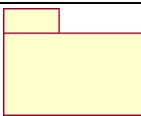
<b>Вопрос 15</b>	<b>Какой вид тестирования проверяет поведение системы на предмет удовлетворения требований заказчика?</b>	<b>Балл</b>
1.	Интеграционное тестирование	
2.	Приёмочное тестирование	
3.	Функциональное тестирование	
4.	Установочное тестирование	

<b>Вопрос 16</b>	<b>Модель жизненного цикла - это</b>	<b>Балл</b>
4.	Совокупность процессов, работ и задач, включающих в себя разработку, эксплуатацию и сопровождение ПП,	
5.	Структура, состоящая из процессов, работ и задач, включающих в себя разработку, эксплуатацию и сопровождение ПП, охватывающая жизнь системы от установления требований к ней до прекращения ее использования	
6.	непрерывный процесс, который начинается с момента принятия решения о необходимости его создания и заканчивается в момент его полного изъятия из эксплуатации	

<b>Вопрос 17</b>	<b>Какая модель ЖЦ не относится к фундаментальным (в соответствии с международными стандартами)?</b>	<b>Балл</b>
5.	Каскадная	
6.	Инкрементная	
7.	Модель быстрого прототипирования	
8.	Эволюционная	

<b>Вопрос 18</b>	<b>Какие процессы ЖЦ ПО не относятся к организационным?</b>	<b>Балл</b>
1.	Сопровождение	
2.	Усовершенствование	
3.	Управление	
4.	Обучение	

<b>Вопрос 19</b>	<b>Какие диаграммы относятся к структурным диаграммам?</b> а) Диаграмма компонентов b) Диаграмма классов c) Диаграмма развертывания d) Диаграмма состояний	<b>Балл</b>
1.	a) b)	
2.	a) d)	
3.	a) c)	
4.	b) d)	

<b>Вопрос 20</b>	<b>Стандартным графическим обозначением компонента на диаграммах является</b>	<b>Балл</b>
	1.  2.  3.  4. 	

<b>Вопрос 21</b>	<b>Что относится к поведенческим нотациям проектирования?</b>	<b>Балл</b>
1.	Язык UML (Unified Modeling Language)	
2.	Диаграммы «сущность-связь» ERD (Entity-Relation Diagrams)	
3.	Диаграммы потоков данных (Data Flow)	

<b><u>Вопрос 22</u></b>	<b>Стандартным графическим обозначением отношения ассоциации на диаграммах использования является</b>	<b>Балл</b>
1.	сплошной линией между актером и вариантом использования	
2.	пунктирной линией со стрелкой между актером и вариантом использования, которая помечается ключевым словом «extend»	
3.	сплошной линией со стрелкой в форме незакрашенного треугольника	
4.	пунктирной линией со стрелкой между актером и вариантом использования, которая помечается ключевым словом «include»	

<b><u>Вопрос 23</u></b>	<b>Какая модель жизненного цикла MSF ориентирована на «вехи»</b>	<b>Балл</b>
1.	Microsoft Solution Framework (MSF)	
2.	Rational Unified Process (RUP)	
3.	Extreme Programming (XP)	

<b><u>Вопрос 24</u></b>	<b>Что такое точка конвергенции (bug convergence)</b>	<b>Балл</b>
1.	При ее достижении становится заметен существенный прогресс в устранении ошибок, то есть скорость устранения ошибок начинает превосходить скорость их обнаружения	
2.	Это момент времени, когда впервые все выявленные ошибки оказываются устраненными	
3.	Это момент времени, когда разработка ПО завершена	

<b><u>Вопрос 25</u></b>	<b>Какая переменная не входит в треугольник компромиссов?</b>	<b>Балл</b>
1.	Время	
2.	Зарплата сотрудников	
3.	Ресурсы	
4.	Возможности	

Итоговая оценка \_\_\_\_\_ (по 5-балльной шкале) ) \_\_\_\_\_



**Вариант 4**  
**Тест по дисциплине «Программная инженерия»**

Студент \_\_\_\_\_ группы \_\_\_\_\_

<b>Вопрос 1</b>	<b>Какой комитет занимается разработкой стандартов по радиоэлектронике и электротехнике?</b>	<b>Балл</b>
1.	Международная организация по стандартизации (ИСО, ISO)	
2.	Международная электротехническая комиссия (МЭК, IEC)	
3.	Ни одна из этих организаций	
4.	Институт инженеров по электронике и электротехнике (IEEE)	

<b>Вопрос 2</b>	<b>Дайте определение парадигмы программирования</b>	<b>Балл</b>
	Это совокупность идей и понятий, определяющая стиль написания программ	
	Это новая модель конструирования программ и взаимодействия ее с данными	
	Это семейство обозначений (нотаций), разделяющих общий способ (методику) реализаций программ	
	Это способ размышления о компьютерных системах	

<b>Вопрос 3</b>	<b>В какие годы сформировалась концепция структурного проектирования?</b>	<b>Балл</b>
1.	В 50-е годы XX века	
2.	В 60-е годы XX века	
3.	В 70-е годы XX века	
4.	В 80-е годы XX века	

<b>Вопрос 4</b>	<b>Суть инкапсуляции в объектном подходе</b>	<b>Балл</b>
1.	Это процесс переноса части объектов в другой модуль («капсулу»)	
2.	Это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение	
3.	Скрытие деталей внутренней реализации	

<b>Вопрос 5</b>	<b>Сколько входов в цикл возможно при написании блок схемы?</b>	<b>Балл</b>
1.	Неограниченное количество	
2.	Один	
3.	Два	

<b>Вопрос 6</b>	<b>Какие характеристики системы определяются нефункциональными требованиями?</b>	<b>Балл</b>
1.	Легкость и простота использования	
2.	Эффективность и устойчивость к сбоям	
3.	Легкость перемещения	
4.	Целостность	
5.	Ни одна из перечисленных характеристик	
6.	Все из перечисленных характеристик	

<b>Вопрос 7</b>	<b>Требования к программному обеспечению – это</b>	<b>Балл</b>
	Общие свойства, которыми должно обладать ПО	
	Совокупность атрибутов, свойств или качеств ПО	
	Свойства, которыми должно обладать ПО для адекватного определения функций, условий и ограничений выполнения ПО, а также объемов данных, технического обеспечения и среды функционирования	

<b>Вопрос 8</b>	<b>Что такое предметная область?</b>	<b>Балл</b>
	Та часть реального мира, которая имеет существенное значение или непосредственное отношение к процессу функционирования программы	
	Некоторая абстрактная единица, которая обладает функциональностью, т. е. может выполнять определенные действия, связанные с решением поставленных задач	
	Это часть программного кода, которая предполагает построение программного обеспечения (ПО) из отдельных компонентов - физически отдельно существующих частей ПО, взаимодействующих между собой	
	Это набор «интеллектуальных пакетов программ»	



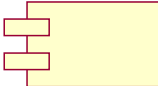

<b>Вопрос 9</b>	<b>Какие стандарты в разработке ПО обязательны для исполнения?</b>	<b>Балл</b>
1.	Корпоративные (отраслевые)	
2.	Государственные	
3.	Международные	

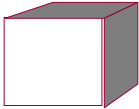

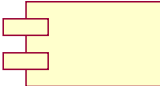
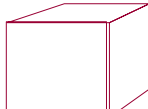
<b>Вопрос 10</b>	<b>К какому показателю качества соотносится совокупность свойств программного средства, характеризующая усилия, необходимые для его проверки после проведения какого-либо видоизменения</b>	<b>Балл</b>
1.	Мобильность	
2.	Тестируемость	
3.	Эффективность	
4.	Надежность	

<b>Вопрос 11</b>	<b>Выберите наиболее точное определение жизненного цикла программного обеспечения</b>	<b>Балл</b>
1.	Непрерывный процесс, который начинается с момента принятия решения о необходимости создания программного обеспечения и заканчивается в момент его полного изъятия из эксплуатации	
2.	Набор взаимосвязанных работ, которые преобразуют исходные данные в выходные результаты	
3.	Процесс определения того, отвечает ли текущее состояние разработки, достигнутое на данном этапе, требованиям этого этапа	
4.	Набор фаз (этапов, стадий) проекта по созданию ПО, в которых выполняются отдельные процессы, разбитые на операции и задачи	

<b>Вопрос 12</b>	<b>При использовании какой модели могут проявляться следующие ее недостатки: 1) невозможно вернуться на одну или две фазы назад, чтобы исправить какую-либо проблему; 2) интеграция компонентов, на которой обычно выявляется большая часть ошибок, выполняется в конце разработки, что сильно увеличивает стоимость устранения ошибок?</b>	<b>Балл</b>
1.	Каскадная модель	
2.	Итерационная модель	
3.	Спиральная модель	
4.	V-образная модель	

<b>Вопрос 13</b>	<b>Какие характеристики качества в модель качества МакКола связаны с внедрением ПО?</b>	<b>Балл</b>
1.	Корректность, надежность, функциональность	
2.	Сопровождаемость, оцениваемость, гибкость	
3.	Переиспользуемость, переносимость, способность к взаимодействию	

Вопрос 14	Модель зрелости (Capability Maturity Models - CMM) используется				Балл
1.	Для оценивания и совершенствования процессов программной инженерии				
2.	Для описание результативности процесса разработки ПО				
3.	Для управления процессом разработки ПО				
Вопрос 15	При использовании какого метода тестирования код программы недоступен тестирующим?				Балл
	Модульное тестирование				
	Метод «Белого ящика»				
	Метод «Черного ящика»				
	Метод «Серого ящика»				
Вопрос 16	Какой вид тестирования проверяет структуру системы на различных наборах данных, а также работу системы на различных конфигурациях аппаратуры и оборудования?				Балл
1.	Интеграционное тестирование				
2.	Конфигурационное тестирование				
3.	Функциональное тестирование				
4.	Стресс-тестирование (нагрузочное)				
Вопрос 17	Когда стала активно использоваться при разработке ПО спиральная модель ЖЦ?				Балл
1.	В 70-х годах XX века				
2.	В 80-х годах XX века				
3.	В 90-х годах XX века				
Вопрос 18	Какая диаграмма используется для моделирования поведения системы?				Балл
	Диаграмма состояний				
	Диаграмма развертывания				
	Диаграмма компонентов				
	Диаграмма классов				
Вопрос 19	Какие диаграммы относятся к структурным диаграммам? Диаграмма компонентов Диаграмма классов Диаграмма развертывания Диаграмма состояний				Балл
1.	a) b)				
2.	a) d)				
3.	a) c)				
4.	b) d)				
Вопрос 20	Стандартным графическим обозначением компонента на диаграммах является				Балл
	1.	2.	3.	4.	
					

<b>Вопрос 21</b>	<b>Стандартным графическим обозначением вычислителя на диаграммах развертывания является</b>				<b>Балл</b>
	1.	2.	3.	4.	
					

<u>Вопрос 22</u>	Стандартным графическим обозначением отношения включения на диаграммах использования является	Балл
1.	сплошной линией между актером и вариантом использования	
2.	пунктирной линией со стрелкой между актером и вариантом использования, которая помечается ключевым словом «extend»	
3.	сплошной линией со стрелкой в форме незакрашенного треугольника	
4.	пунктирной линией со стрелкой между актером и вариантом использования, которая помечается ключевым словом «include»	

<b>Вопрос 23</b>	<b>Что такое точка достижения нуля (zero-bug bounce)</b>	<b>Балл</b>
	При ее достижении становится заметен существенный прогресс в устранении ошибок, то есть скорость устранения ошибок начинает превосходить скорость их обнаружения	
	Это момент времени, когда впервые все выявленные ошибки оказываются устраненными	
	Это момент времени, когда разработка ПО заверше	

<b>Вопрос 24</b>	<b>С какой ролью нельзя совмещать тестирование?</b>	<b>Балл</b>
	Разработчика	
	Бизнес-аналитика	
	Менеджера проекта	
	Релиз-менеджера	

<b>Вопрос 25</b>	<b>В какой модели ЖЦ ПО каждый виток представляет собой фазу разработки?</b>	<b>Балл</b>
1.	Каскадной	
2.	Спиральной	
3.	Итерационной	
4.	V-образной	

Итоговая оценка \_\_\_\_\_ (по 5-балльной шкале) ) \_\_\_\_\_