МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Государственное образовательное учреждение высшего профессионального образования «Кемеровский государственный университет»
Кафедра ЮНЕСКО по новым информационным технологиям

А.М. Гудов, С.Ю. Завозкин, С.Н. Трофимов

Технология разработки программного обеспечения

Учебное пособие

Кемерово, 2009

Оглавление

Предисловие

Главной целью представленного пособия является освоение базовых знаний по вопросам проектирования и разработки информационных систем.

Объектами изучения в данной дисциплине являются: технологии проектирования, модели и методы поддержки жизненного цикла программного обеспечения; средства и методы создания и реализации проектов по созданию программных систем.

Основными задачами данного пособия и соответсвующего лабораторного практикума являются:

- знакомство с основными этапами жизненного цикла программного обеспечения;
- знакомство с технологиями функционального и объектно-ориентированного проектирования;
- приобретение навыков работы со средствами автоматизации разработки ПО;
- приобретение навыков по созданию программного средства с использованием базы данных;
- подготовка студентов к изучению других дисциплин по информационным технологиям.

Курс, поддерживаемый этим пособием, занимает особое место в учебном плане среди дисциплин математических специальностей по его значению. Вместе с курсами по программированию, курс «Технологии разработки программного обеспечения» составляет основу образования студента в части информационных технологий. Курс рассчитан на студентов-математиков, имеющих подготовку по математике и информатике в объеме программы средней школы. В течение преподавания курса предполагается, что студенты знакомы с основными понятиями процедурного и объектно-ориентированного программирования, логики, информатики, которые

читаются на математических и естественнонаучных факультетах перед изучением данной дисциплины.

Пособие включает следующие разделы: технологии, модели и процессы создания ПО, основы создания ПО, разработка требований к ПО, управление проектами по созданию и внедрению ПО, управление персоналом при реализации проектов, оценка стоимости программного продукта, управление качеством созданных программных систем, создание спецификации программной системы с использованием элементов объектного проектирования.

Особенностью пособия является то, что его можно применять как основу при изучении теоретического материала и получения навыков проектирования и разработки приложения с использованием CASE-систем на лабораторных занятиях или в рамках самостоятельной работы студента. Пособие представлено в двух вариантах: в виде печатного издания и в качестве набора HTML-страниц, что существенно упрощает его использование при изучении соответствующей дисциплины с использованием дистанционных технологий.

Учебное пособие предназначено для студентов, изучающих технологии создания ПО, и специалистов по программному обеспечению, работающих в различных областях разработки программных систем. Его можно использовать как основу базового курса по технологии разработки ПО или в качестве материала для таких курсов, как углубленные технологии программирования, спецификации ПО, разработка и управление программными системами.

Глава 1. Технологии, модели и процессы создания ПО

Терминология

Программное обеспечение (ПО) – компьютерные программы и соответствующая документация. Разрабатывается по частному заказу или для продажи на рынке ПО.

Инженерия ПО – инженерная дисциплина, охватывающая все аспекты разработки ПО.

Системотехника (технология создания вычислительных систем) – дисциплина, охватывающая все аспекты создания и модернизации сложных вычислительных систем, где программное обеспечение играет ведущую роль. Сюда можно отнести технологии создания аппаратных средств, создание вычислительных процессов, развертывание всей системы, а также технологию создания непосредственно ПО.

Процесс создания ПО – совокупность процессов, приводящих к созданию программного продукта.

Фундаментальные процессы, присущие любому проекту создания ПО:

- Разработка спецификации требований на ПО (Определяют функциональные характеристики системы и обязательны для выполнения).
- Создание программного обеспечения (создание ПО согласно спецификации).
- Аттестация ПО (Созданное ПО должно пройти аттестацию для подтверждения соответствию требованиям заказчика).
- Модернизация ПО (совершенствование ПО согласно измененным требованиям потребителя).

Модель процесса создания ПО – последовательность этапов, необходимых для разработки создаваемого ПО.

Рисунок 1 – Распределения затрат при разработке ПО

Модели процесса разработки ПО:

1. Каскадная модель
2. Эволюционная модель
3. Формальное преобразование
4. Сборка программных продуктов из ранее созданных компонентов (модель сборки)
5. Итерационная (спиральная) модель

Методы создания ПО представляют собой структурный подход к созданию ПО, который способствует производству ПО эффективным, с экономической точки зрения, способом.

Все методы основаны на использовании моделей системы в качестве спецификации ее структуры:

1. **Функционально-ориентированные** (структурный анализ, JSD, 70-е годы) основаны на определении основных функциональных компонент системы.

2. **Объектно-ориентированные** (Booch, Rumbaugh) используют подходы, основанные на использовании унифицированного языка моделирования UML.

Computer-Aided Software Engineering – автоматизированная разработка ПО.

Процессы создания ПО

Базовые процессы создания ПО:

- Разработка спецификации.
- Проектирование и реализация.
- Аттестация.
- Эволюция.

Жизненный цикл ПО – совокупность процессов, протекающих от момента принятия решения о создании ПО до его полного вывода из эксплуатации.

Каскадная модель

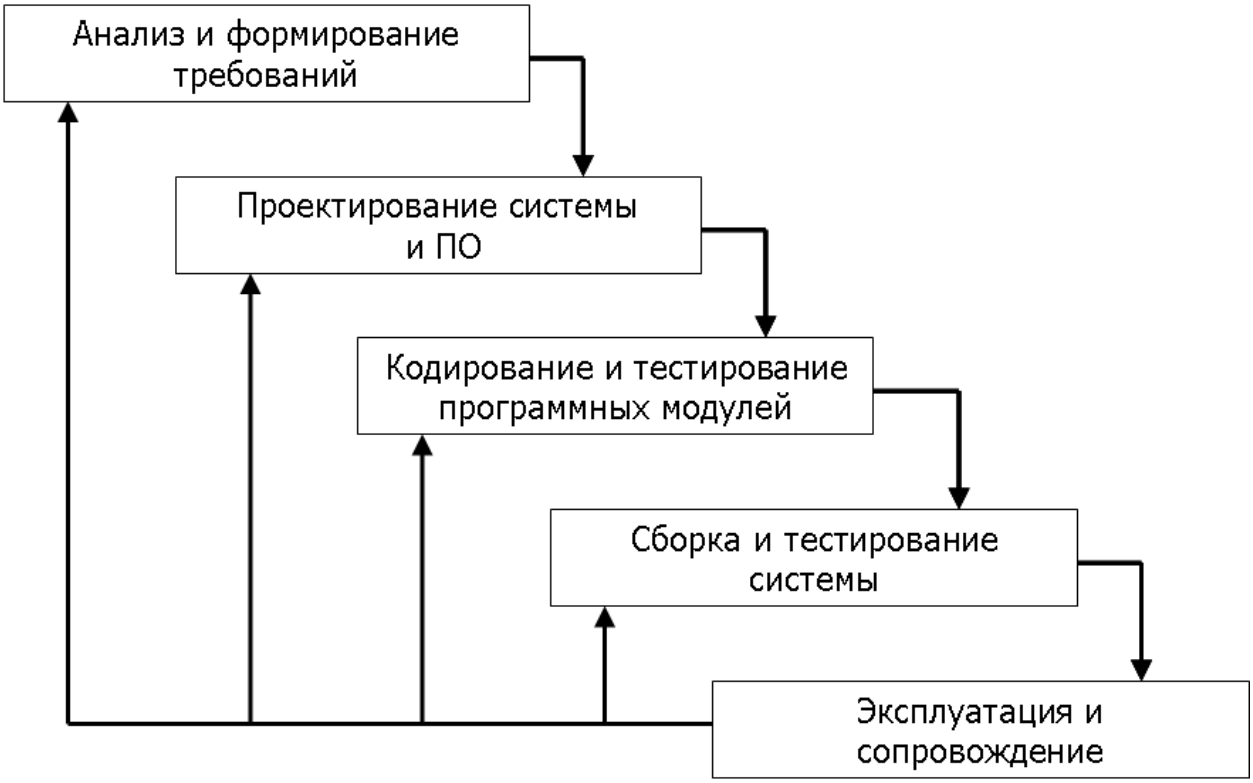


Рисунок 2 – Каскадная модель создания ПО

Достоинства:

- Документирование каждого этапа.

Недостатки:

- «Негибкое» разбиение процесса создания на отдельные этапы.

Применение:

- Требования сформулированы достаточно четко.
- Повсеместно для разработки небольших систем, входящих в состав крупного проекта.

Эволюционная модель

Прототип – действующий программный модуль, реализующий отдельные функции создаваемого ПО.

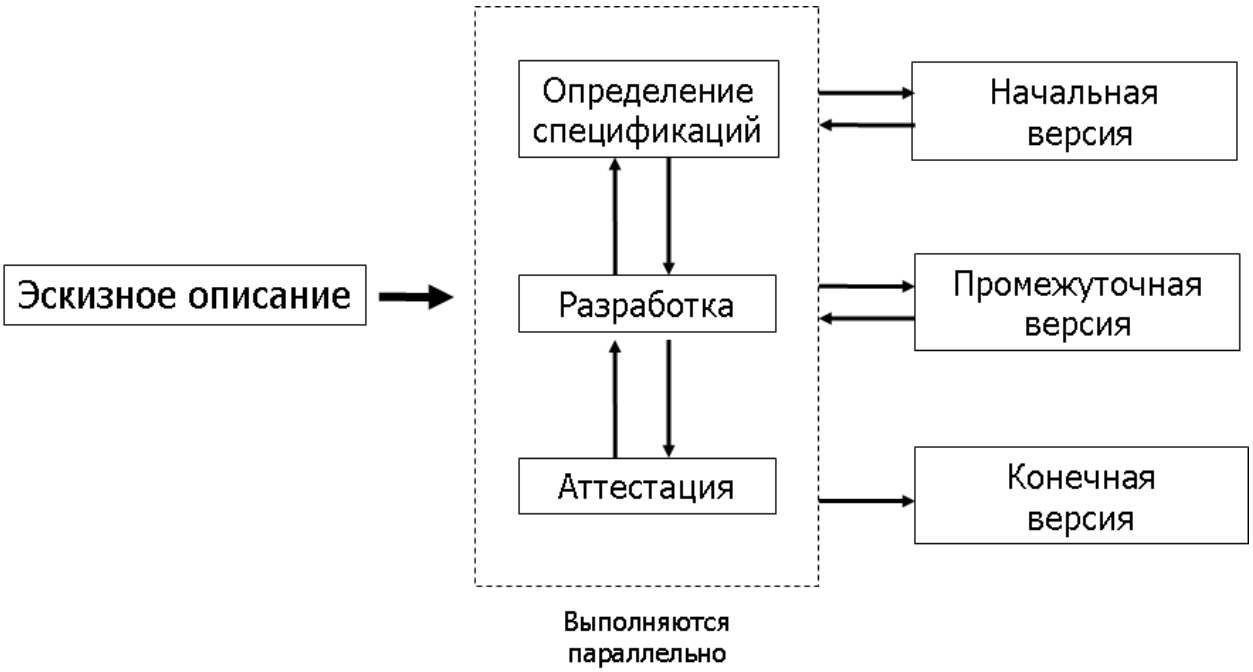


Рисунок 3 – Эволюционная модель создания ПО

Достоинства:

- Спецификация разрабатывается постепенно, по мере требования заказчика.

Недостатки:

- Многие этапы создания ПО не документированы.
- Система часто получается плохо структурированной.
- Требуются специальные средства и технологии разработки ПО.

Применение:

- Разработка небольших систем (<100 000 строк) или средних (<500 000 строк) с относительно коротким сроком жизни.

Формальная разработка

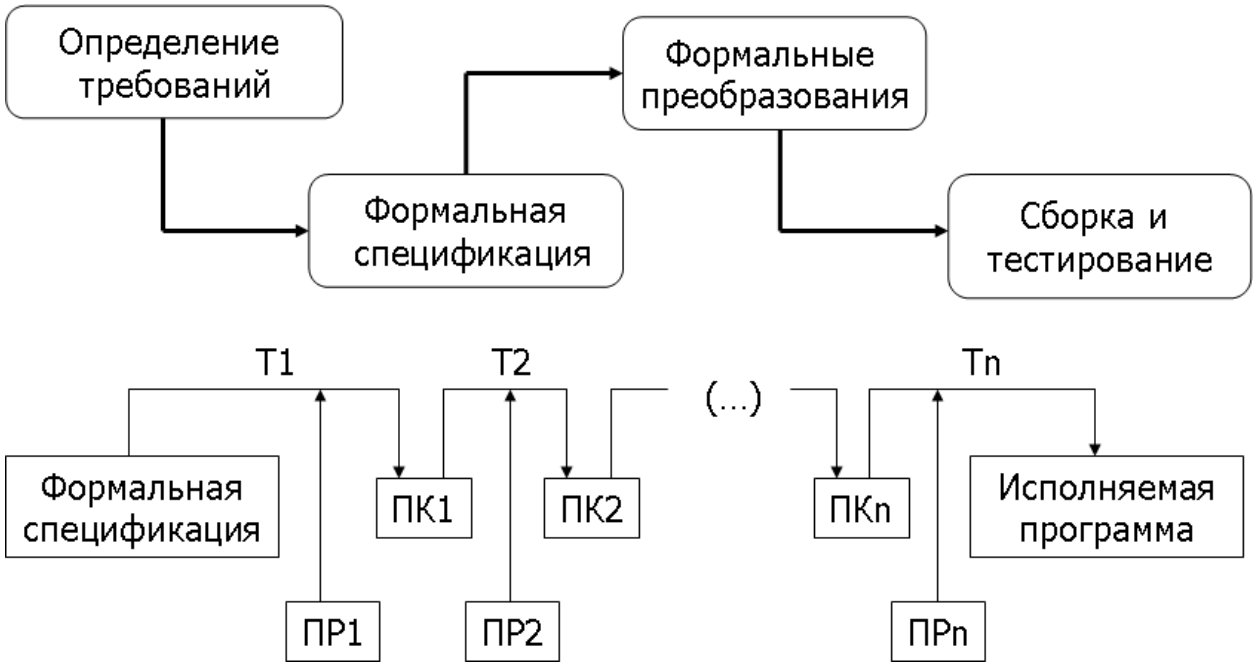


Рисунок 4 - Процесс формальных преобразований

Преимущества:

- Точное соответствие программы спецификации.
- Отказ от тестирования отдельных модулей.
- Тестирование всей системы только после ее сборки.

Недостатки:

- Требуют специальных знаний и опыта использования.
- Не дают существенного выигрыша в стоимости разработки.
- Большинство сложных систем с трудом поддаются формальному описанию.

Модель пошаговой разработки

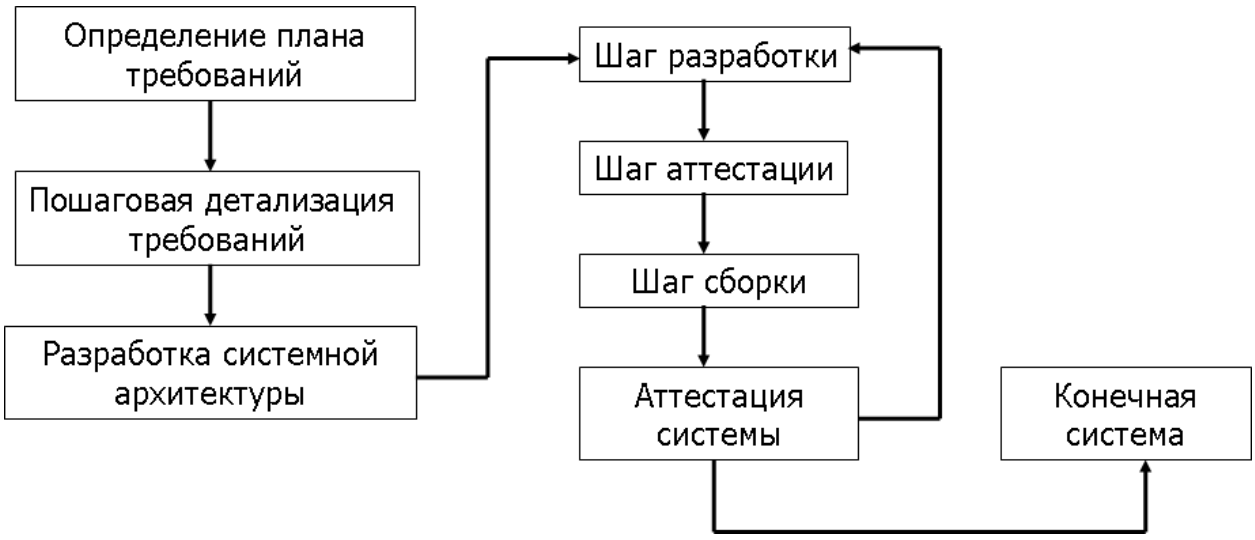


Рисунок 5 – Модель пошаговой разработки

На каждом шаге отсутствует требование использования одного и того же подхода к процессу разработки!

Достоинства:

- Нет необходимости ждать полного завершения разработки системы.
- Можно использовать компоненты, полученные на первых шагах, как прототипы.
- Уменьшается риск общесистемных ошибок.
- Системные сервисы с высоким приоритетом разрабатываются первыми, а все последующие интегрируются с ними. Это позволяет снизить вероятность программных ошибок в особо важных частях системы.

Недостатки:

- Компоненты, получаемые на каждом шаге, имеют небольшой размер.
- Сложно определить на первых этапах общесистемные функции.
- Невозможно сразу определить набор базовых свойств, которые зачастую разрабатываются совместно с другими частями системы.

Спиральная модель

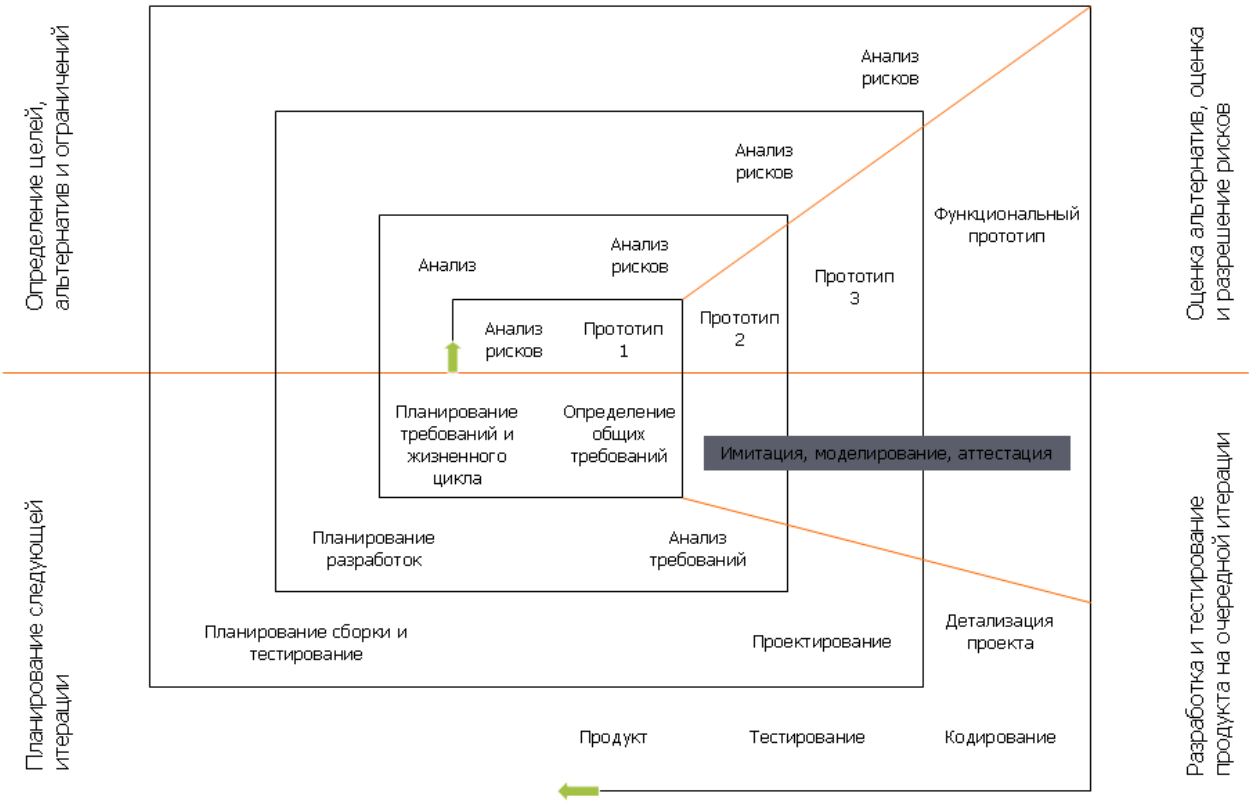


Рисунок 6 – Спиральная модель

Достоинства:

- Нет фиксированных этапов.
- Эта модель может включать в себя любые другие модели на каждом витке спирали:
  - прототипирование может использоваться при нечетком определении требований;
  - Каскадная модель в случае последовательного выполнения некоторых этапов;
  - Модель формальных преобразований – если четко сформулированы требования.

Недостатки:

- Сложена автоматизация процессов разработки.

– Огромная роль при разработке системы отводится управлению проектом.

Базовые процессы создания ПО

Разработка спецификации ПО – определение сервисов, которыми будет обладать создаваемое ПО, а также ограничений, налагаемых на функциональные возможности и разработку ПО.

Результат процесса определения требований – документация, формализующая требования, предъявляемые к системе.

Два уровня детализации:

– Требования, предъявляемые конечными пользователями;

– Системная спецификация для разработчиков.

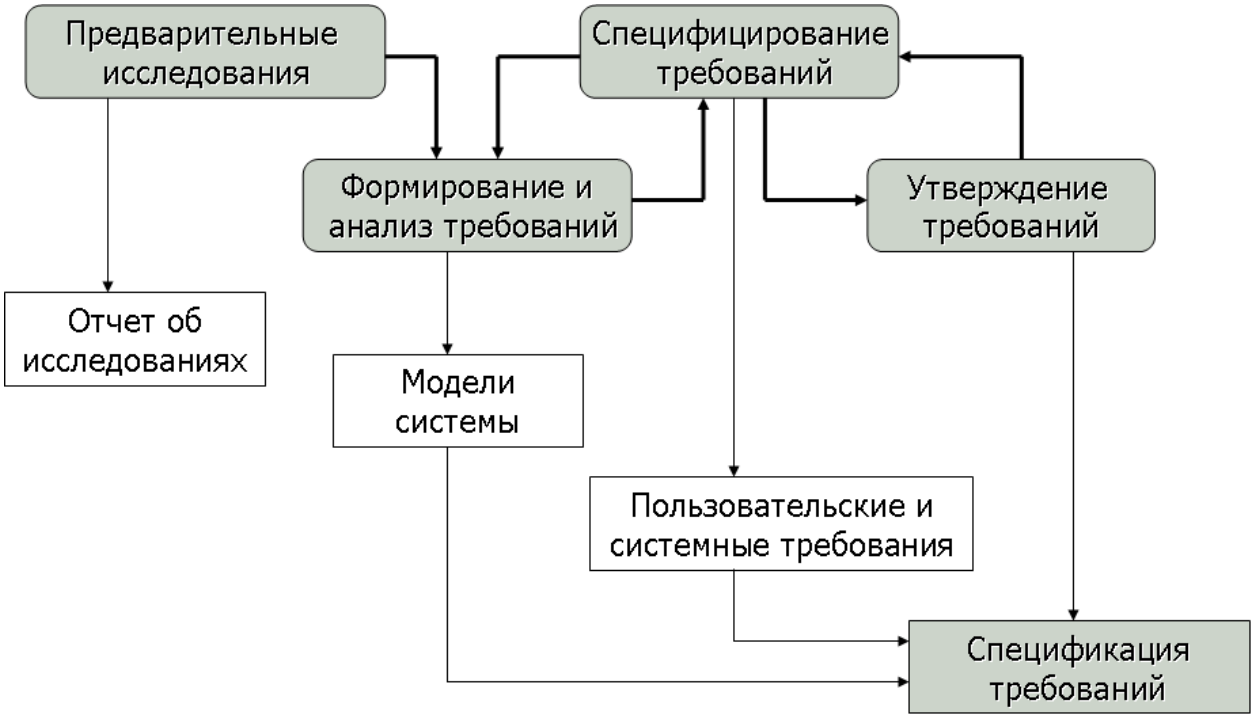


Рисунок 7 – Процесс спецификации требований

Реализация ПО – процесс перевода системной спецификации в работоспособную систему. Включает в себя процессы проектирования и программирования.

Процесс проектирования включает в себя определение структуры ПО, данных, интерфейсов взаимодействия системных компонентов, используемые алгоритмы. Проектирование предполагает последовательную формализацию и детализацию создаваемого ПО.

Результат каждого этапа проектирования – спецификация, необходимая для выполнения следующего этапа.

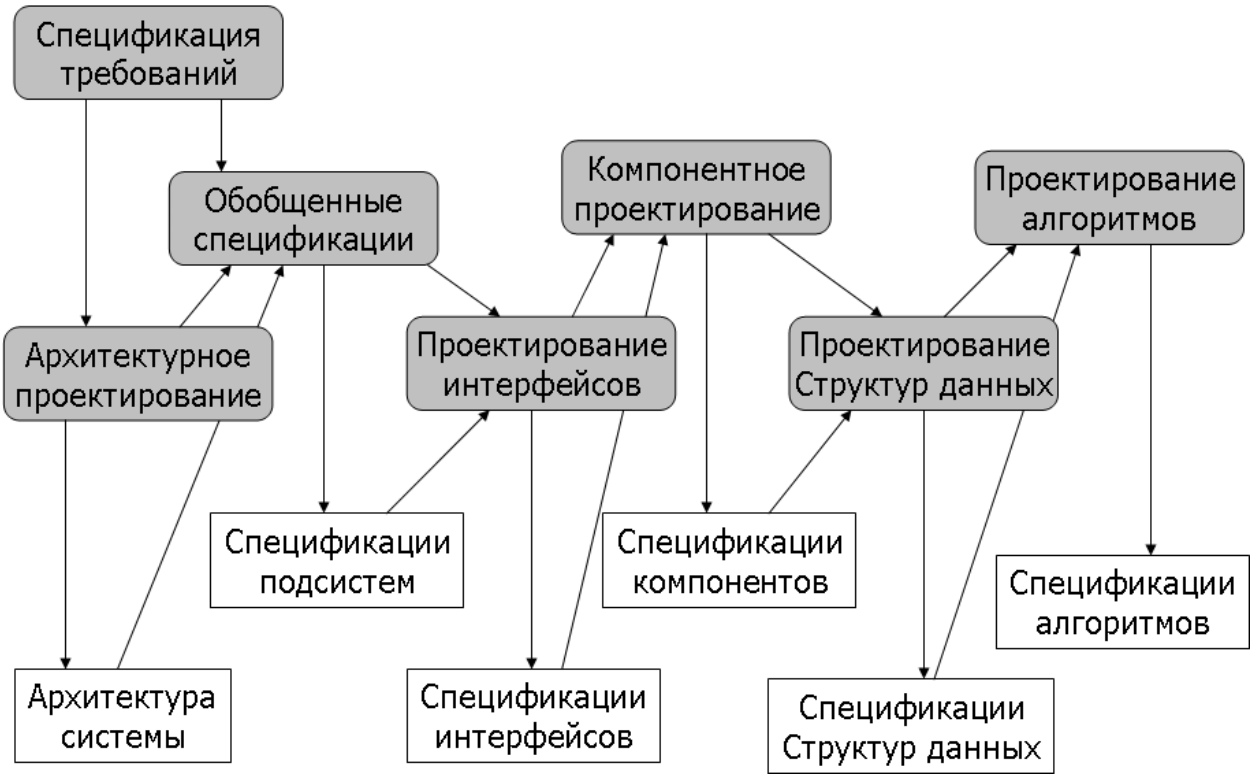


Рисунок 8 – процесс формирования спецификации требований

Методы проектирования – множество формализованных нотаций и нормативных документов для проектирования ПО.

Структурные методы поддерживают модели системы:

- Модель потоков данных;
- Модель «сущность-связь»;
- Структурная модель;
- Объектно-ориентированные иерархическая модель системы, модель отношений между объектами, модель взаимодействия объектов;

– Диаграммы переходов или сценарии жизни сущностей.

Программирование и отладка:

Тестирование – процесс установления программных ошибок.

Отладка – установление местоположения ошибок и их устранение.

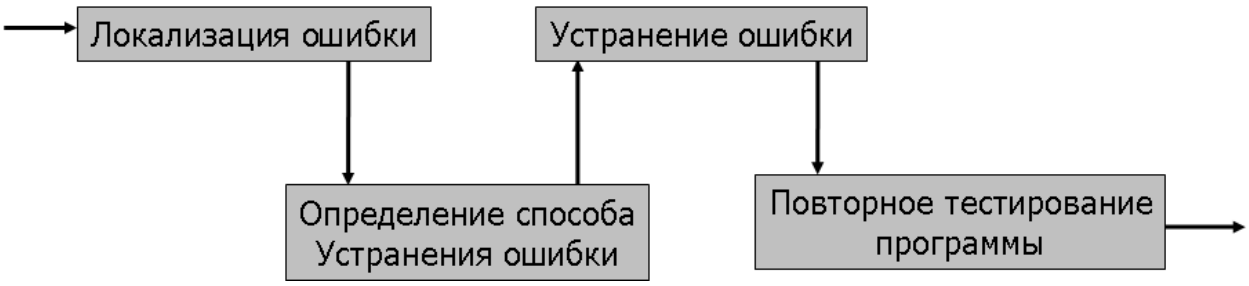


Рисунок 9 – Процесс тестирования

Аттестация и верификация – процесс установления соответствия ПО ее спецификации, а также ожиданиям и требованиям пользователей и заказчика.

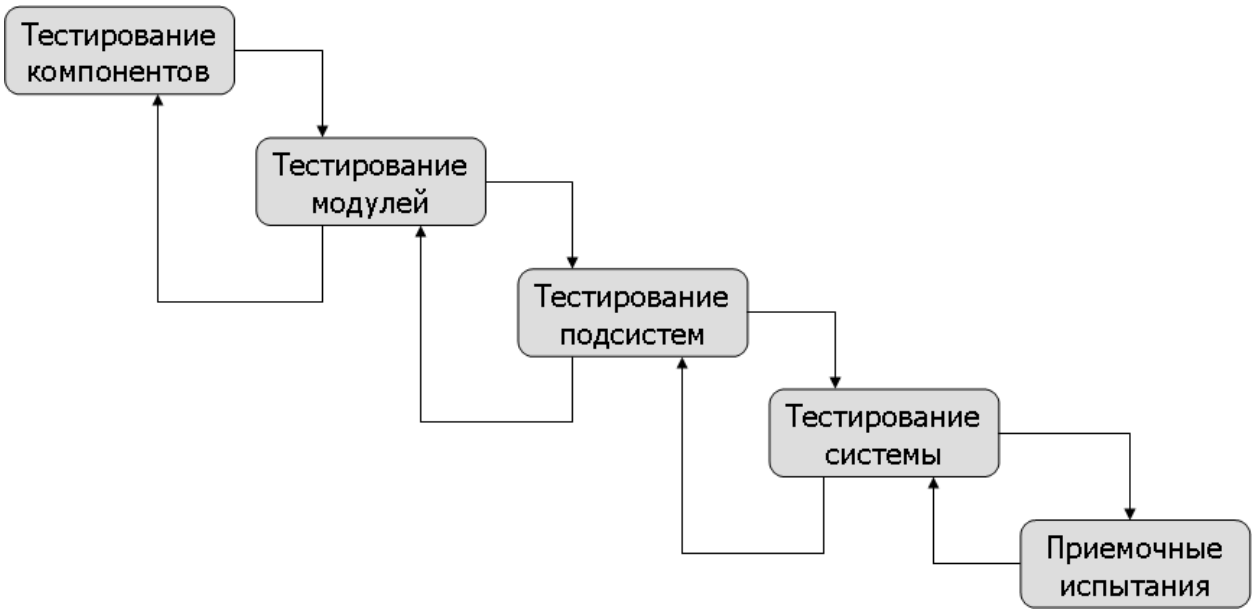


Рисунок 10 – Процесс аттестации и верификации

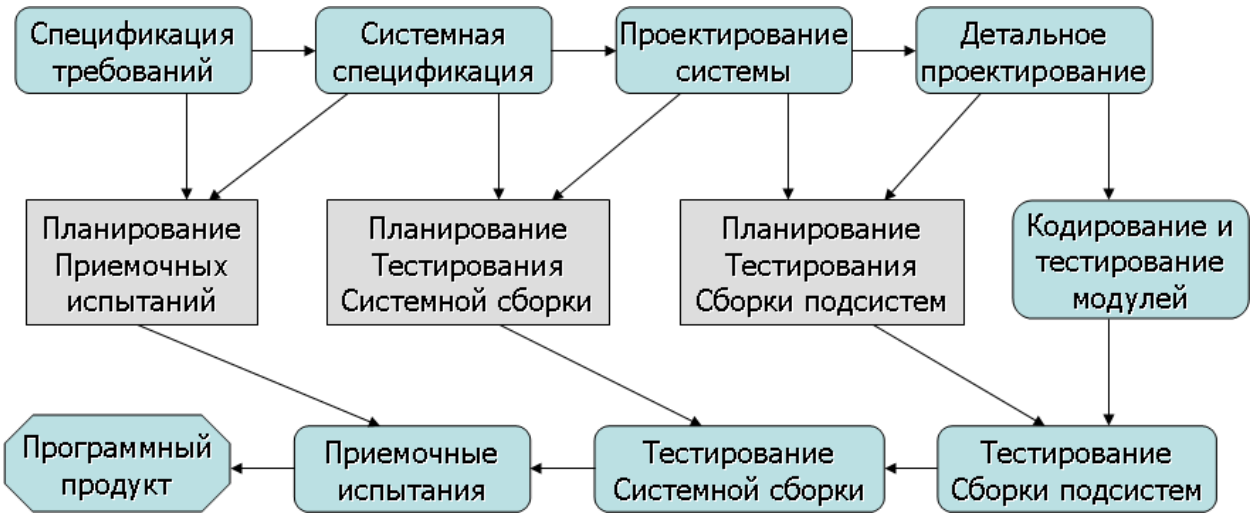


Рисунок 11 - Этапы тестирования

Сопровождение системы – это внесение изменений в систему, которая находится в эксплуатации.

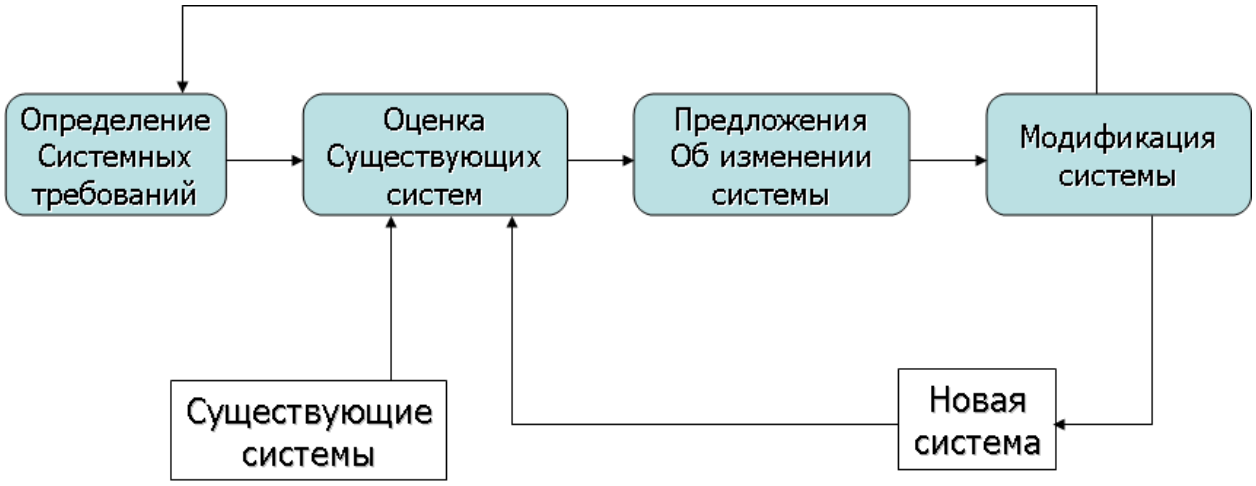


Рисунок 12 - Эволюция систем

Вопросы для обсуждения

1. Почему в процессе определения требований необходимо различать разработку пользовательских требований и разработку системных требований?
2. Каковы пять основных компонентов любых методов проектирования?
3. Разработайте модель процесса тестирования исполняемой программы.

Глава 2. Разработка требований к ПО

Анализ осуществимости

Разработка требований — это процесс, включающий мероприятия, необходимые для создания и утверждения документа, содержащего спецификацию системных требований.

Различают четыре основных этапа процесса разработки требований:

- анализ технической осуществимости создания системы,
- формирование и анализ требований,
- специфицирование требований и создание соответствующей документации,
- аттестация этих требований.

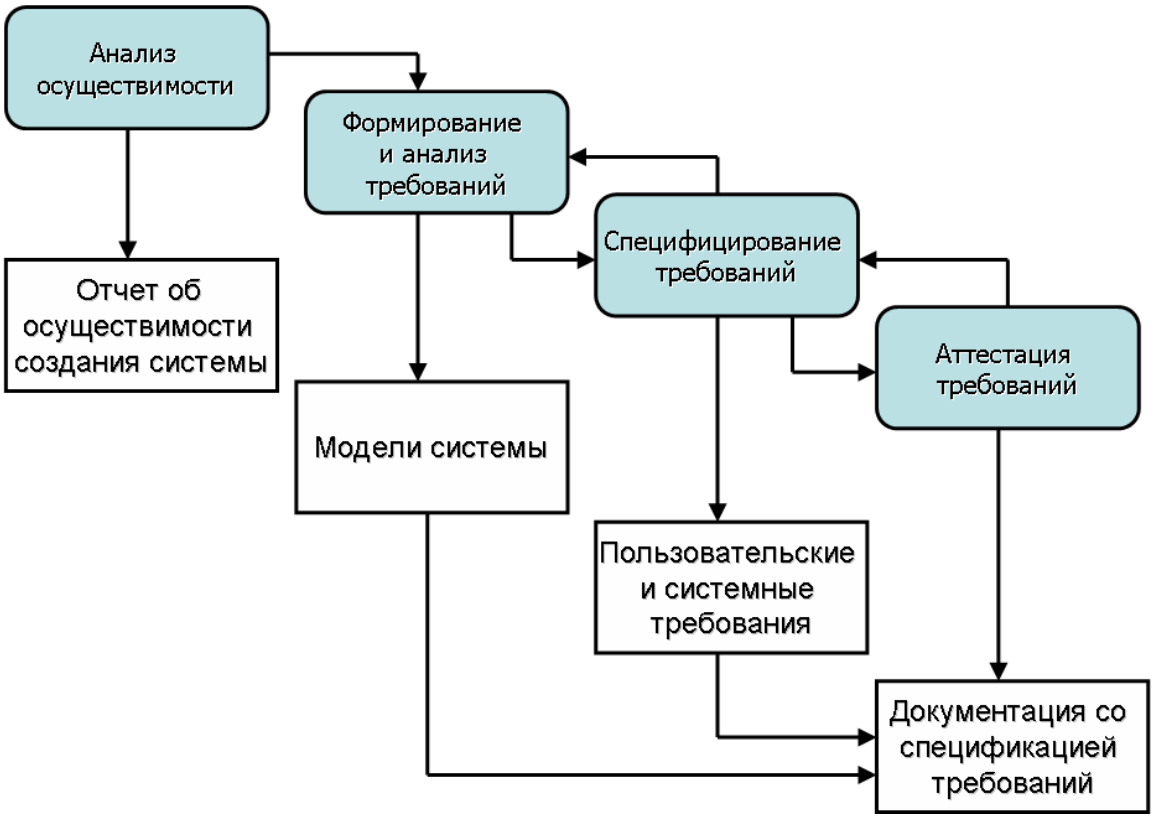


Рисунок 13 – Процесс формирования требований

Анализ осуществимости должен осветить следующие вопросы:

1. Отвечает ли система общим и бизнес-целям организации-заказчика и организации-разработчика?
2. Можно ли реализовать систему, используя существующие на данный момент технологии и не выходя за пределы заданной стоимости?
3. Можно ли объединить систему с другими системами, которые уже эксплуатируются?

Выполнение анализа осуществимости включает сбор и анализ информации о будущей системе, написание соответствующего отчета. Например, эту информацию можно получить, ответив на следующие вопросы:

1. Что произойдет с организацией, если система не будет введена в эксплуатацию?
2. Какие текущие проблемы существуют в организации и как новая система поможет их решить?
3. Каким образом система будет способствовать целям бизнеса?
4. Требует ли разработка системы технологии, которая до этого не использовалась в организации?

После обработки собранной информации готовится отчет по анализу осуществимости создания системы.

На этапе **формирования и анализа требований** команда разработчиков ПО работает с заказчиком и конечными пользователями системы для выяснения области применения, описания системных сервисов, определения режимов работы системы и ее характеристик выполнения, аппаратных ограничений и т.д.

Процесс формирования и анализа требований достаточно сложен по ряду причин:

– На требования к системе могут влиять политические факторы.

– Лица, участвующие в формировании требований, выражают в этих требованиях собственные точки зрения, основываясь на личном опыте работы.

– Лица участвующие в формировании требований, имеют различные предпочтения и могут выражать их разными способами. Разработчики должны определить все потенциальные источники требований и выделить общие и противоречивые требования.

– Экономическая и бизнес-обстановка, в которой происходит формирование требований, неизбежно будет меняться в ходе выполнения этого процесса.

– Лица, участвующие в формировании требований, часто не знают конкретно, чего они хотят от компьютерной системы.

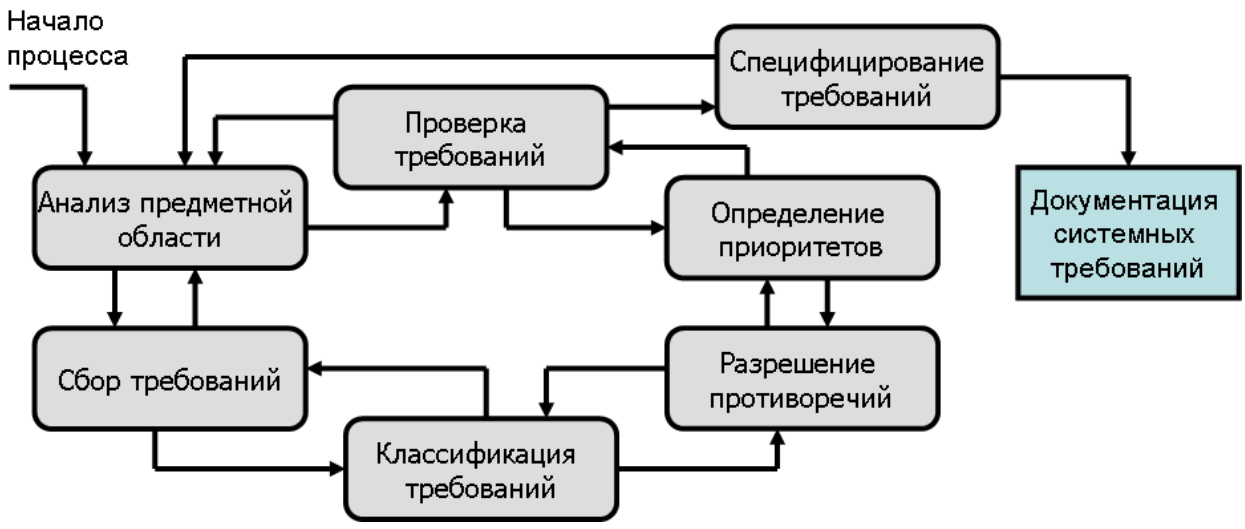


Рисунок 14 – Процесс формирования и анализа требований

Процесс формирования и анализа требований проходит через ряд этапов:

– Анализ предметной области. Аналитики должны изучить предметную область, где будет эксплуатироваться система.

- Сбор требований. Это процесс взаимодействия с лицами, формирующими требования. Во время этого процесса продолжается анализ предметной области.
- Классификация требований. На этом этапе бесформенный набор требований преобразуется в логически связанные группы требовании.
- Разрешение противоречий. Без сомнения, требования многочисленных лиц, занятых в процессе формирования требований, будут противоречивыми. На этом этапе определяются и разрешаются противоречия такого рода.
- Назначение приоритетов. В любом наборе требований одни из них будут более важны, чем другие. На этом этапе совместно с лицами, формирующими требования, определяются наиболее важные требования.
- Проверка требований. На этом этапе определяется их полнота, последовательность и непротиворечивость.

Распространены три подхода к формированию требований: метод, основанный на множестве **опорных точек зрения**, **сценарии** и **этнографический** метод.

Другие подходы, которые могут использоваться в процессе разработки требований, — это методы структурного анализа и методы прототипирования.

Не существует универсального подхода к формированию и анализу требований. Обычно для разработки требований одновременно используется несколько подходов.

Метод опорных точек зрения

Различные точки зрения на проблему позволяют увидеть ее с разных сторон. Однако эти взгляды не являются полностью независимыми и обычно перекрывают друг друга, а потому могут служить основой общих требований.

Подход с использованием различных **опорных** точек зрения к разработке требований признает эти точки зрения и использует их в качестве основы построения и организации как процесса формирования требований, так и непосредственно самих требований.

Различные методы предлагают разные трактовки выражения "точка зрения". Точки зрения можно трактовать следующим образом:

- Как источник информации о системных данных. В этом случае на основе опорных точек зрения строится модель создания и использования данных в системе. В процессе формирования требований отбираются все такие точки зрения, на их основе определяются данные, которые будут созданы или использованы при работе системы, и способы обработки этих данных.
- Как получатели системных сервисов. В этом случае точки зрения являются внешними (относительно системы) получателями системных сервисов. Точки зрения помогают определить данные, необходимые для выполнения системных сервисов или их управления.
- Как получатели системных сервисов. В этом случае точки зрения являются внешними (относительно системы) получателями системных сервисов. Точки зрения помогают определить данные, необходимые для выполнения системных сервисов или их управления.

Наиболее эффективным подходом к анализу интерактивных систем является использование внешних опорных точек зрения. Эти точки зрения взаимодействуют с системой, получая от нее сервисы и продуцируя данные и управляющие сигналы.

Этот тип точек зрения имеет ряд преимуществ:

1. Точки зрения, внешние к системе, — естественный способ структурирования процесса формирования требований.
2. Сравнительно просто решить, какие точки зрения следует оставить в качестве опорных: они должны отображать какой-либо способ взаимодействия с системой.
3. Данный подход полезен для создания нефункциональных требований, с которыми можно связать какой-либо сервис.

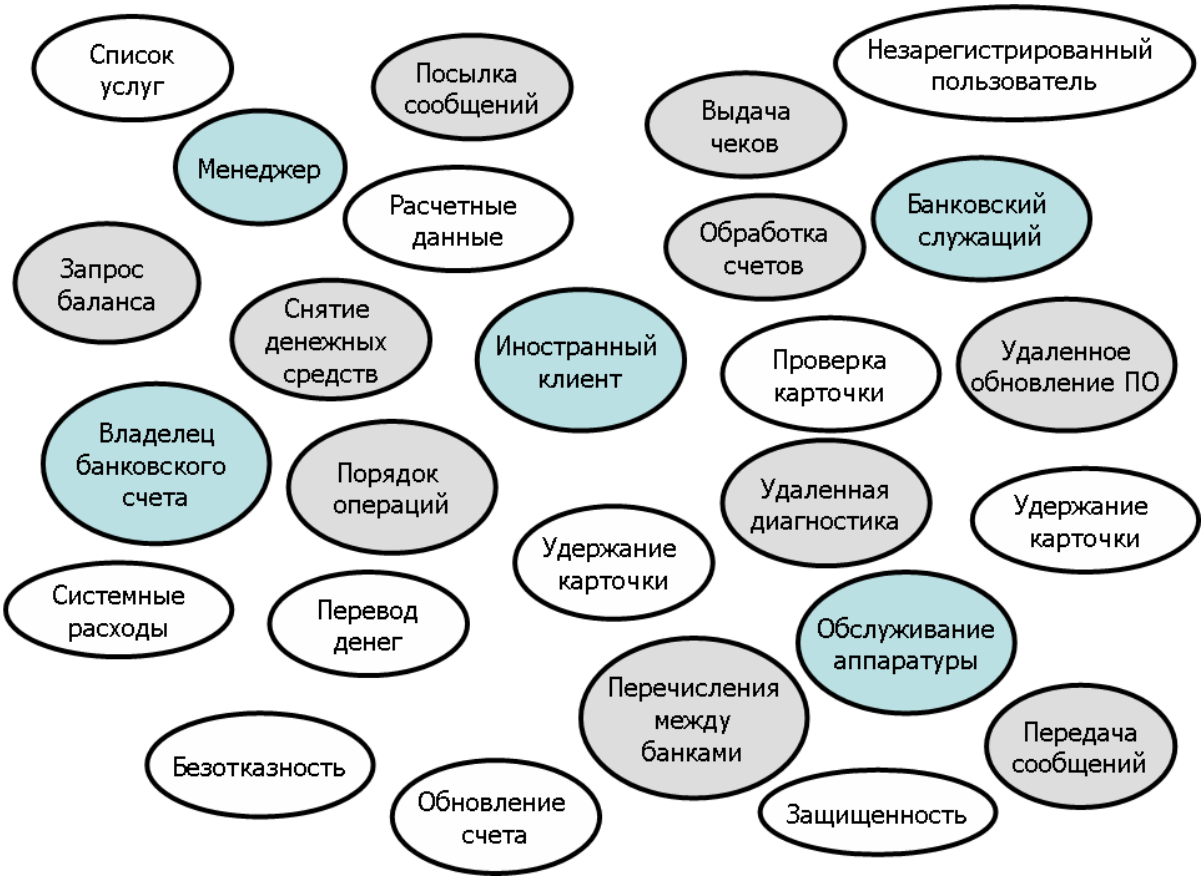


Рисунок 15 - Диаграмма идентификации точек зрения

Таблица 1 - Сервисы, соотнесенные с точками зрения

ВЛАДЕЛЕЦ СЧЕТА	ИНОСТРАННЫЙ КЛИЕНТ	КАССИР БАНКА
Список сервисов	Список сервисов	Список сервисов
Выдача денег Запрос баланса Выдача чеков Посылка сообщения Список транзакций Порядок операций Перевод денег	Выдача денег Запрос баланса	Выполнение диагностики Зачисление денег Обработка счетов Посылка сообщения

Информация, извлеченная из точек зрения, используется для заполнения форм шаблонов точек зрения и организации точек зрения в иерархию наследования. Сервисы, данные и управляющая информация наследуются подмножеством точек зрения.

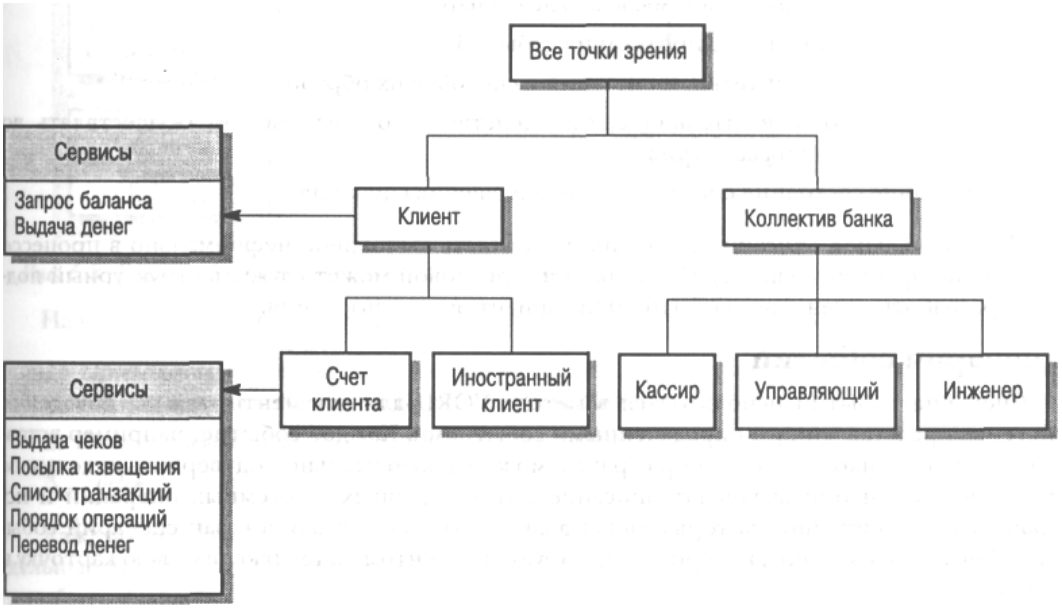


Рисунок 16 – Диаграмма пользователей, соотнесенная с возможными сервисами

Сценарии особенно полезны для детализации уже сформулированных требований, поскольку описывают последовательность интерактивной работы пользователя с системой. Каждый сценарий описывает одно или несколько возможных взаимодействий.

Сценарий начинается с общего описания, затем постепенно детализируется для создания полного описания взаимодействия пользователя с системой.

В большинстве случаев сценарий включает следующее:

- Описание состояния системы после завершения сценария.
- Информацию относительно других действий, которые можно осуществлять во время выполнения сценария.
- Описание исключительных ситуаций и способов их обработки.
- Описание нормального протекания событий.
- Описание состояния системы в начале сценария.

Сценарии событий используются для документирования поведения системы, представленного определенными событиями. Сценарии включают описание потоков данных, системных операций и исключительных ситуаций, которые могут возникнуть

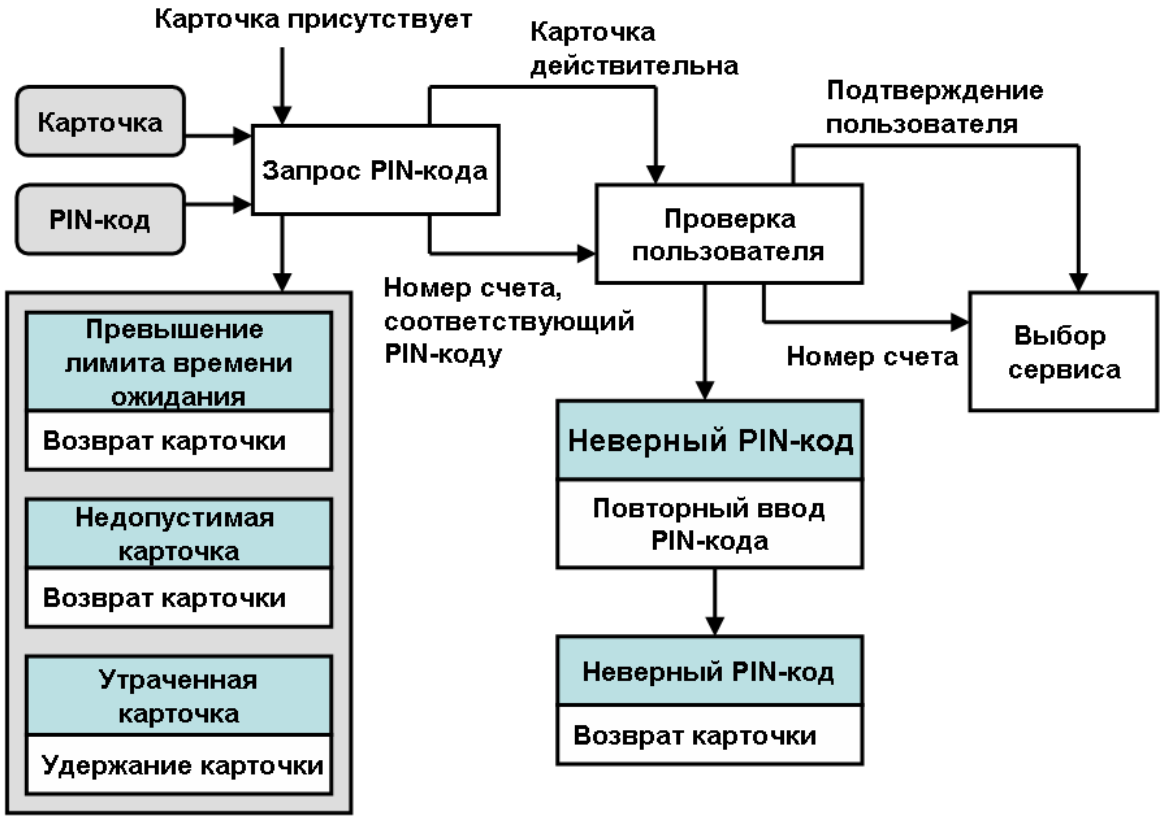


Рисунок 17 – Диаграмма сценариев

Условные обозначения:

1. Данные, поступающие в систему или исходящие из нее, представлены в эллипсах.
2. Управляющая информация показана стрелками в верхней части прямоугольников.
3. Внутрисистемные данные показаны справа от прямоугольников.
4. Исключительные ситуации показаны в нижней части прямоугольников.
5. Имя следующего события, ожидаемого после завершения сценария, приводится в затененном прямоугольнике.

Варианты использования (use-case) — это методика формирования требований, основанная на сценариях. Они стали основой нотаций в языке моделирования UML при описании объектных моделей систем.

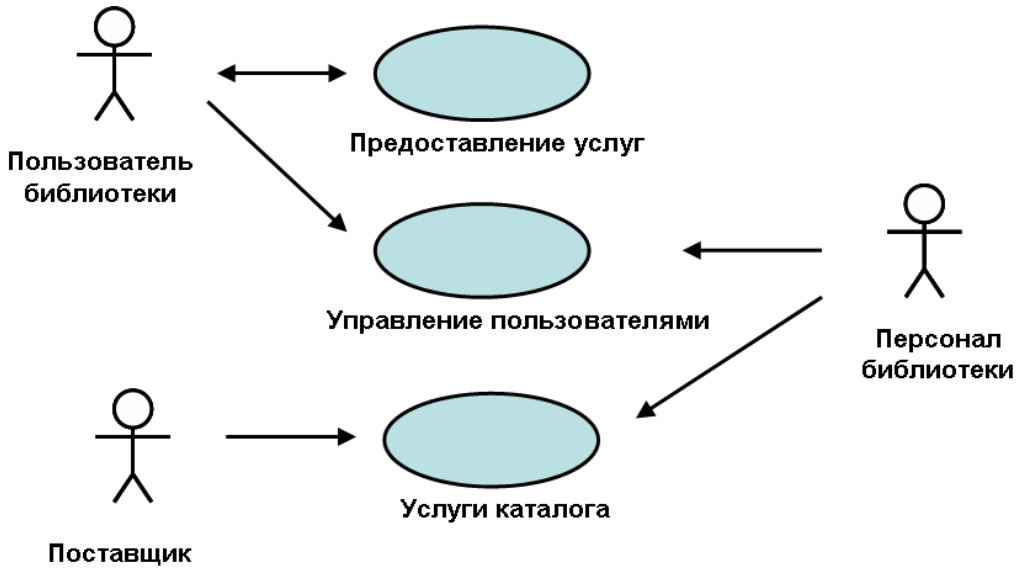


Рисунок 18 – Диаграмма вариантов использования

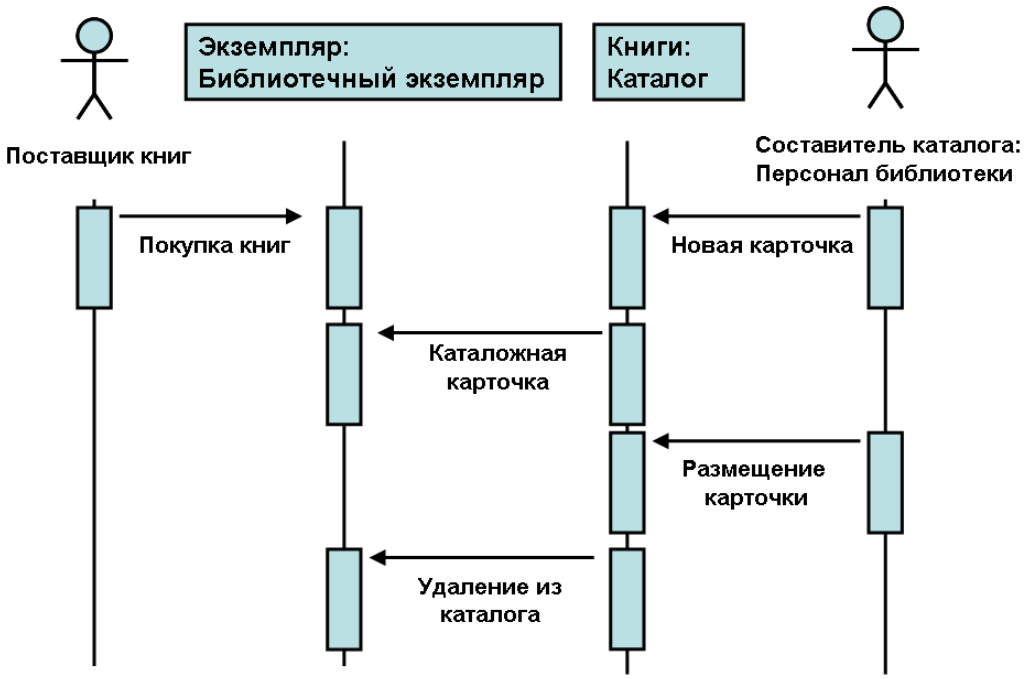


Рисунок 19 – Диаграмма последовательности

Этнографический подход

Этнографический подход к формированию системных требований используется для понимания и формирования социальных и организационных аспектов эксплуатации системы. Разработчик требований погружается в рабочую среду, где будет использоваться система. Его ежедневная работа связана с наблюдением и протоколированием реальных

действий, выполняемых пользователями системы. Значение этнографического подхода заключается в том, что он помогает обнаружить неявные требования к системе, которые отражают реальные аспекты ее эксплуатации, а не формальные умозрительные процессы.

Этнографический подход позволяет детализировать требования для критических систем, чего не всегда можно добиться другими методами разработки требований. Однако, поскольку этот метод ориентирован на конечного пользователя, он не может охватить все требования предметной области и требования организационного характера.

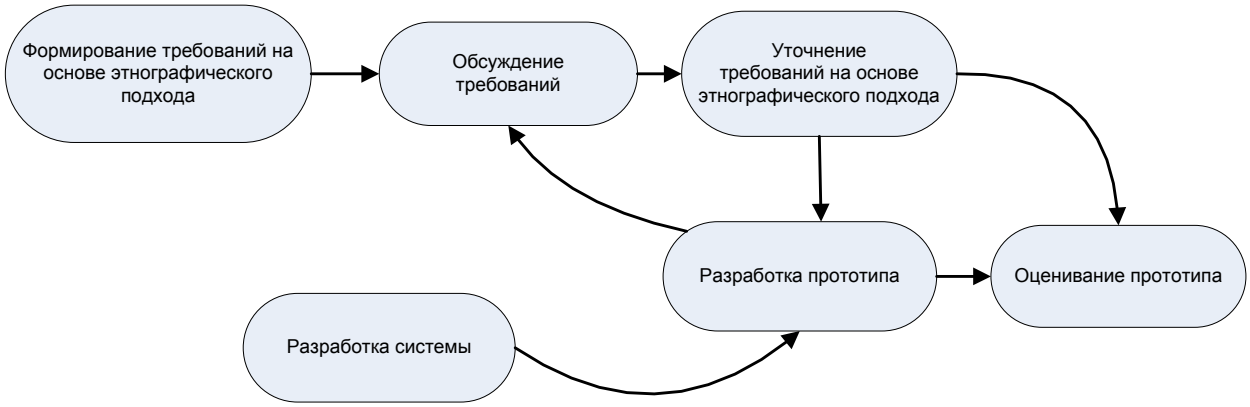


Рисунок 20 – Процесс разработки требований согласно этнографическому подходу

Во время процесса **аттестации** должны быть выполнены различные типы проверок документации требований:

1. Проверка правильности требований.
2. Проверка на непротиворечивость.
3. Проверка на полноту.
4. Проверка на выполнимость.

Существует ряд методов аттестации требований:

1. Обзор требований.
2. Прототипирование.

3. Генерация тестовых сценариев.
4. Автоматизированный анализ непротиворечивости.

Управление требованиями — это процесс управления изменениями системных требований. Процесс управления требованиями выполняется совместно с другими процессами разработки требований. Начало этого процесса планируется на то же время, когда начинается процесс первоначального формирования требований, непосредственно процесс управления требованиями должен начаться сразу после того, как черновая версия спецификации требований будет готова.

С точки зрения разработки требования можно разделить на два класса:

1. Постоянные требования.
2. Изменяемые требования.

Планирование управления требованиями

1. Идентификация требований.
2. Управление процессом внесения изменений.
3. Стратегия оперативного контроля.
   - Информация об источнике требования
   - Информация о требованиях
   - Информация о структуре системы
4. Поддержка CASE-средств.

Процесс управления изменениями состоит из трех основных этапов:

1. Анализ проблем изменения спецификации.
2. Анализ изменений и расчет их стоимости.
3. Реализация изменений.

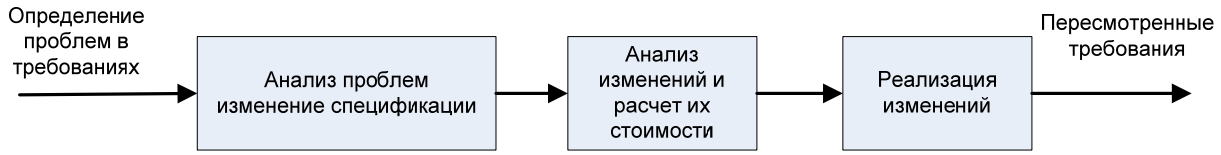


Рисунок 21 – Процесс управления требованиями

Вопросы для обсуждения

1. Предложите, кто бы мог участвовать в формировании требований для университетской системы регистрации студентов.
2. Разрабатывается система ПО для автоматизации библиотечного каталога. Эта система будет содержать информацию относительно всех книг в библиотеке и будет полезна библиотечному персоналу, абонентам и читателям. Система должна иметь средства просмотра каталога, средства создания запросов и средства, позволяющие пользователям резервировать книги, находящиеся в данный момент на руках. Определите основные опорные точки зрения, которые необходимо учесть в спецификации системы.
3. Ваша компания использует стандартный метод анализа требований. В процессе работы вы обнаружили, что этот метод не учитывает социальные факторы, важные для системы, которую вы анализируете. Ваш руководитель дал вам ясно понять, какому методу анализа нужно следовать. Обсудите, что вы должны делать в такой ситуации.

Формальные спецификации

Так называемые формальные методы разработки программных систем широко не используются. Многие компании, разрабатывающие ПО, не считают экономически выгодным применение этих методов в процессе разработки.

Термин "формальные методы" подразумевает ряд операций, в состав которых входят создание формальной спецификации системы, анализ и

доказательство спецификации, 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.

В инженерии ПО определены три уровня спецификации программного обеспечения. Это пользовательские и системные требования и спецификация структуры программной системы. Пользовательские требования наиболее обобщенные, спецификация структуры наиболее детальна. Формальные математические спецификации находятся где-то между системными требованиями и спецификацией структуры. Они не содержат деталей реализации системы, но должны представлять ее полную математическую модель.

Разработка спецификация и проектирование могут выполняться параллельно, когда информация от этапов разработки спецификации передается к этапам проектирования и наоборот.

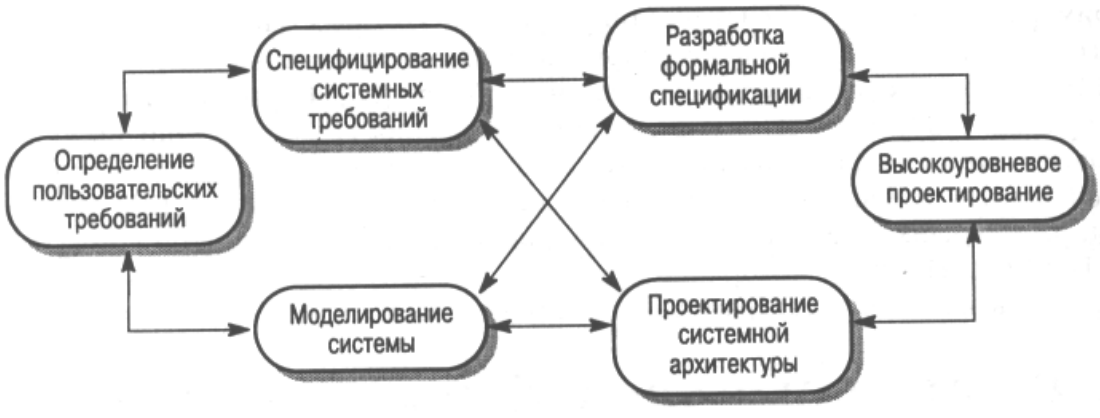


Рисунок 22 – Процесс разработки формальной спецификации

Создание формальной спецификации требует детального анализа системы, который позволяет обнаружить ошибки и несоответствия в спецификации неформальных требований. Проблемы в требованиях, которые остаются необнаруженными до последних стадий процесса разработки ПО, обычно требуют больших затрат на исправление.

Существует два основных подхода к разработке формальной спецификации, которые используются для написания детализированных спецификаций нетривиальных программных систем.

1. Алгебраический подход, при котором система описывается в терминах операций и их отношений.
2. Подход, ориентированный на моделирование, при котором модель системы строится с использованием математических конструкций, таких, как множества и последовательности, а системные операции определяются тем, как они изменяют состояния системы.

Для разработки формальных спецификаций последовательных и параллельных систем в настоящее время создано несколько языков, представленных ниже в таблице.

Таблица 2 - Языки разработки формальных спецификаций

Тип языка	***Последовательные системы***	***Параллельные системы***
Алгебраический	Larch, OBJ	Lotos
Основанный на моделях	Z, VDM, B	CSP, сети Петри

Большие системы обычно разбиваются на подсистемы, которые разрабатываются независимо друг от друга. Подсистемы могут использовать другие подсистемы, поэтому необходимой частью процесса специфицирования является определение интерфейсов подсистем. Если

интерфейсы определены и согласованы, подсистемы можно разрабатывать независимо друг от друга.

Интерфейс подсистемы часто определяется как набор абстрактных типов данных и объектов, при этом только через интерфейс доступны описание данных и операции над ними. Поэтому спецификацию интерфейса подсистемы можно рассматривать как объединение спецификаций компонентов, что в итоге и составит описание интерфейса подсистемы.

Точные спецификации интерфейсов подсистем необходимы для разработчиков, которые пишут программный код, обращающийся к сервисам других подсистем. Спецификации интерфейсов содержат информацию о том, какие сервисы доступны в других подсистемах и как получить к ним доступ. Ясный и однозначный интерфейс подсистем уменьшает вероятность ошибок во взаимоотношениях между ними.

Алгебраический подход первоначально был разработан для описания интерфейсов абстрактных типов данных, где типы данных определяются скорее спецификациями операций над данными, чем способом представления самих данных. Это очень похоже на определение классов объектов. Алгебраический подход к формальным спецификациям определяет абстрактный тип данных в терминах операций над данными.

Структура спецификации объекта состоит из четырех компонентов:

- Введение, где объявляется класс (sort) объектов. Класс — это общее название для множества объектов. Он обычно реализуется как тип данных. Введение может также включать объявление импорта (imports), где указываются имена спецификаций, определяющие другие классы. Импортирование спецификаций делает эти классы доступными для использования.
- Описательная часть, в которой неформально описываются операции, ассоциированные с классом. Это делает формальную спецификацию более простой для понимания. Формальная спецификация дополняет

это описание, обеспечивая однозначный синтаксис и семантику операций.

– Часть сигнатур, в которой определяется синтаксис интерфейса объектного класса или абстрактного типа данных. Здесь описываются имена операций, количество и типы их параметров, а также классы выходных результатов операции.
– Часть аксиом, где определяется семантика операций посредством создания ряда аксиом, которые характеризуют поведение абстрактного типа данных. Эти аксиомы связывают операции создания объектов класса с операциями, проверяющими их значения.

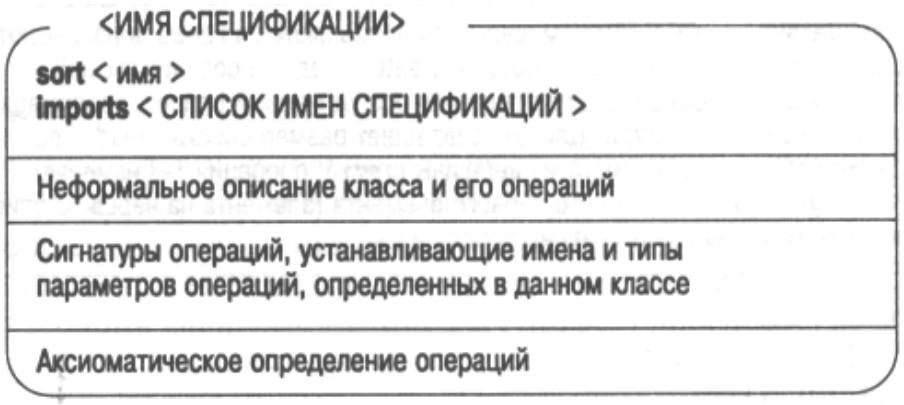


Рисунок 23 - Структура алгебраической спецификации

Процесс разработки формальной спецификации интерфейса подсистемы включает следующие действия.

1. *Структурирование спецификации.* Представление неформальной спецификации интерфейса в виде множества абстрактных типов данных или объектных классов. Также неформально определяются операции, ассоциированные с каждым классом.
2. *Именование спецификаций.* Задаются имена для каждой спецификации абстрактного типа, определяются параметры спецификаций (если они необходимы) и имена определяемых классов.

3. *Определение операций.* На основании списка выполняемых интерфейсом функций для каждой спецификации определяется связанный с ней набор операций. Необходимо предусмотреть операции по созданию экземпляров классов, по изменению значений экземпляров классов и по проверке этих значений. Вероятно, придется добавить новые функции к первоначально определенному списку функций интерфейса.
4. *Неформальная спецификация операций.* Написание неформальной спецификации для каждой операции, где должно быть указано, как операции воздействуют на определяемый класс.
5. *Определение синтаксиса операций.* Определение синтаксиса и параметров для каждой операции. Это часть сигнатуры формальной спецификации.
6. *Определение аксиом.* Определение семантики операций путем описания условий, которые должны выполняться для различных комбинаций операций.

Операции над абстрактным типом данных обычно относятся к одному из двух классов.

1. *Операции конструирования,* которые создают или изменяют объекты класса. Обычно их называют Create (Создать), Update (Изменить), Add (Добавить) или Cons (Конструирование).
2. *Операции проверки,* которые возвращают атрибуты класса. Обычно им дают имена, соответствующие именам атрибута, или имена, подобные Eval (Значение), Get (Получить) и т.п.

Хорошим эмпирическим правилом для написания алгебраической спецификации является создание аксиом для каждой операции конструирования с применением всех операций проверки. Это означает, что если есть m операций конструирования и n операций проверки, то должно быть определено m x n аксиом.

Операции конструирования, связанные с абстрактным типом данных, часто очень сложны и могут определяться через другие операции конструирования и проверки. Если операции конструирования определены посредством других операций, то необходимо определить операции проверки, используя более примитивные конструкции.

В спецификации списка операциями конструирования являются Create, Cons и Tail, которые создают списки. Операциями проверки являются Head и Length, которые используются для получения значений атрибутов списка. Операция Tail не является примитивной конструкцией, поэтому для нее можно не определять аксиомы с использованием операций Head и Length, но в таком случае Tail необходимо определить посредством примитивных конструкций.

При написании алгебраических спецификаций часто используется рекурсия. Результат операции Tail — список, сформированный из входного списка путем удаления верхнего элемента. Это определение подсказывает, как использовать рекурсию для построения данной операции. Операция определяется на пустых списках, затем рекурсивно переходит на непустые списки и завершается, когда результатом снова будет пустой список.

Иногда проще понять рекурсивные преобразования, используя короткий пример. Предположим, что есть список [5, 7], где элемент 5 — начало (вершина) списка, а элемент 7— конец списка. Операция Cons([5, 7], 9) должна возвратить список [5, 7, 9], а операция Tail, примененная к этому списку, должна возвратить список [7, 9]. Приведем последовательность рекурсивных преобразований, приводящую к этому результату.

Tail ([5,7,9]) =

= Tail (Cons ([5, 7], 9)) =

= Cons (Tail ([5, 7]), 9) =

= Cons (Tail (Cons ([5], 7)), 9) =

= Cons (Cons (Tail ([5]), 7), 9) =

= Cons (Cons (Tail (Cons ([], 5)), 7), 9) =

= Cons (Cons ([Create], 7), 9) =

= Cons ([7], 9) =

= [7, 9]

Здесь систематически использовались аксиомы для Tail, что привело к ожидаемому результату. Аксиому для операции Head можно проверить подобным способом.

Простые алгебраические методы подходят для описания интерфейсов, когда операции, ассоциированные с объектом, не зависят от состояния объекта. Тогда результаты любой операции не зависят от результатов предыдущих операций. Если это условие не выполняется, алгебраические методы могут стать громоздкими. Более того, я думаю, что алгебраические описания поведения систем часто искусственны и трудны для понимания.

Альтернативным подходом к созданию формальных спецификаций, который широко используется в программных проектах, является спецификация, основанная на моделях системы. Такие спецификации используют модели состояний системы. Системные операции определяются посредством изменений состояний системной модели. Таким образом определяется поведение системы.

Вопросы для обсуждения

1. Перед вами поставлена задача "продажи" методов формальной

спецификации организации, разрабатывающей программное обеспечение. Как вы будете объяснять преимущества формальной спецификации скептически настроенным разработчикам ПО?

2. Объясните, почему необходимо определять интерфейсы подсистем как можно точнее и почему алгебраическая спецификация наиболее подходит для специфицирования интерфейсов подсистем.
3. Абстрактный тип данных, представляющий стек, имеет следующие операции:
    a. New (Создать) — создает пустой стек;
    b. Push (Добавить) — добавляет элемент в вершину стека;
    c. Top (Вершина) — возвращает элемент на вершине стека;
    d. Retract (Удалить) — удаляет элемент из вершины стека и возвращает модифицированный стек;
    e. Empty (Пустой) — возвращает значение истины, если стек пустой.

Определите этот абстрактный тип данных, используя алгебраическую спецификацию.

4. Вы системный инженер и вас просят назвать наилучший способ разработки программного обеспечения для сердечного стимулятора, критического по обеспечению безопасности. Вы предлагаете разработать формальную спецификацию системы, но ваше предложение отвергнуто менеджером. Вы считаете, что его доводы не обоснованы и базируются на предубеждениях. Будет ли этичной разработка системы с использованием методов, которые вы считаете неподходящими?

Модели систем

Одной из широко используемых методик документирования системных требований является построение ряда моделей системы. Эти модели используют графические представления, показывающие решение как исходной задачи, для которой создается система, так и разрабатываемой

системы. Модели являются связующим звеном между процессом анализа исходной задачи и процессом проектирования системы.

Модели могут представить систему в различных аспектах:

- Внешнее представление, когда моделируется окружение или рабочая среда системы.
- Описание поведения системы, когда моделируется ее поведение.
- Описание структуры системы, когда моделируется системная архитектура или структуры данных, обрабатываемых системой.

Наиболее важным аспектом системного моделирования является то, что оно опускает детали. Модель является абстракцией системы и легче поддается анализу, чем любое другое представление этой системы. В идеале **представление** системы должно сохранять всю информацию относительно представляемого объекта. **Абстракция** является упрощением и определяется выбором наиболее важных характеристик системы.

Типы системных моделей, которые могут создаваться в процессе анализа систем:

- Модель обработки данных. Диаграммы потоков данных показывают последовательность обработки данных в системе.
- Композиционная модель. Диаграммы "сущность-связь" показывают, как системные сущности составляются из других сущностей.
- Архитектурная модель. Эти модели показывают основные подсистемы, из которых строится система.
- Классификационная модель. Диаграммы наследования классов показывают, какие объекты имеют общие характеристики.

– Модель "стимул-ответ". Диаграммы изменения состояний показывают, как система реагирует на внутренние и внешние события.

Модели системного окружения

При анализе будущей системы необходимо определить границы между системой и ее окружением, специфицировать само рабочее окружение и связи между ним и системой. Обычно на этом этапе строится простая структурная модель.

Структурные модели высокого уровня обычно являются простыми блок-схемами, где каждая подсистема представлена именованным прямоугольником, а линии показывают, что существуют некоторые связи между подсистемами.

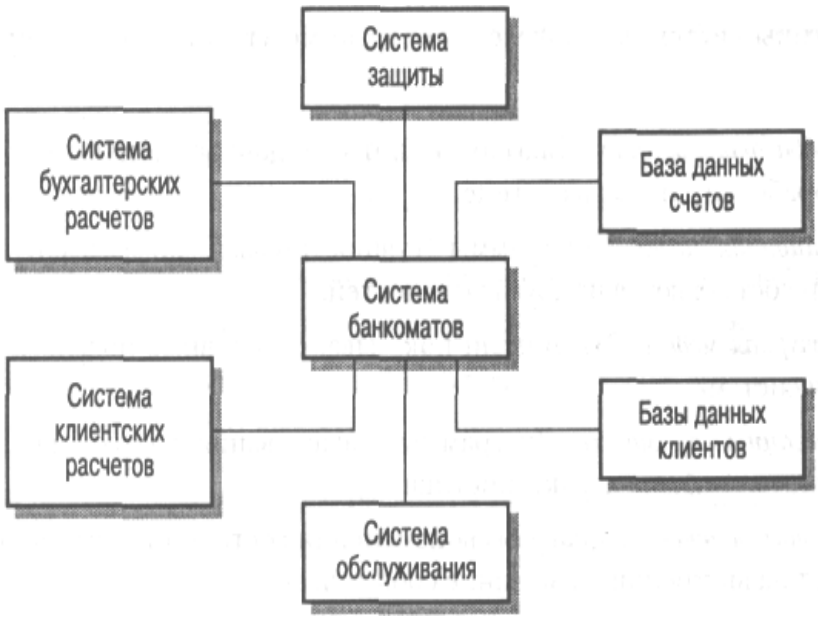


Рисунок 24 – Пример модели окружения

Простые структурные модели обычно дополняются моделями других типов, например **моделями процессов**, которые показывают взаимодействия в системе, или **моделями потоков данных**, которые показывают

последовательность обработки и перемещения данных внутри системы и между другими системами в окружающей среде.

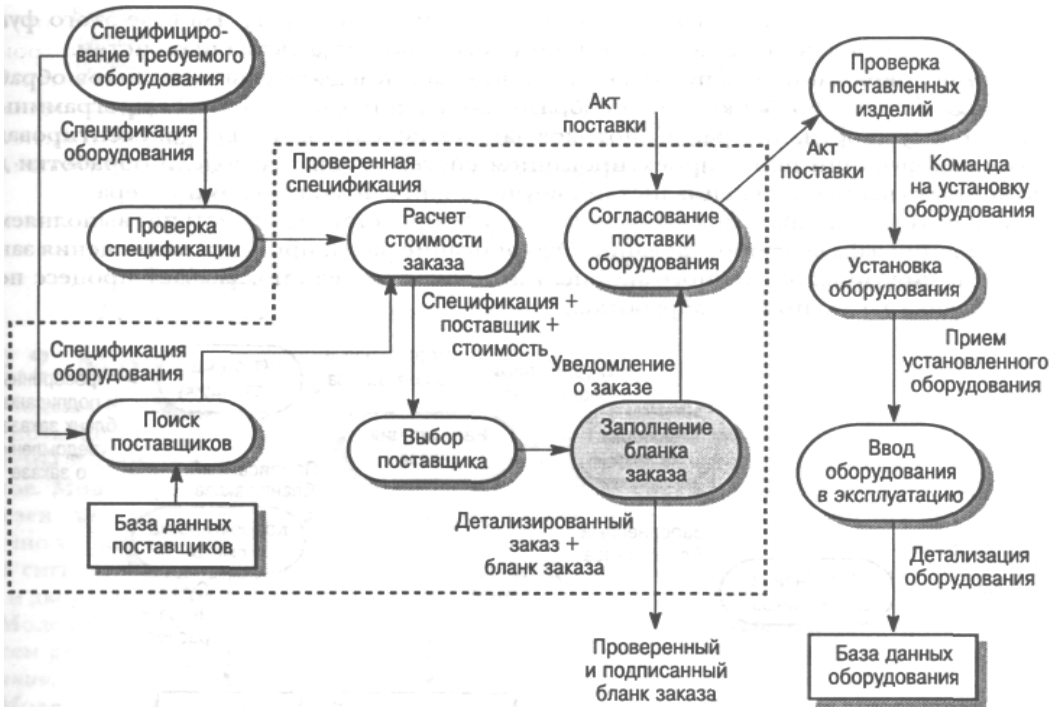


Рисунок 25 - Модель процесса приобретения оборудования

Поведенческие модели

Эти модели используются для описания общего поведения системы. Обычно рассматривают два типа поведенческих моделей **— модель потоков данных** и **модель конечного автомата**. Эти модели можно использовать отдельно или совместно, в зависимости от типа разрабатываемой системы.

Модели потока данных — это интуитивно понятный способ показа последовательности обработки данных внутри системы. Нотации, используемые в этих моделях, описывают обработку данных с помощью системных функций, а также хранение и перемещения данных между системными функциями.

В диаграммах потоков данных используются следующие обозначения: закругленные прямоугольники соответствуют этапам обработки данных;

стрелки, снабженные примечаниями с названием данных, представляют потоки данных; прямоугольники соответствуют хранилищам или источникам данных.

Рисунок 26 – Модель потоков данных

Модели потоков данных показывают функциональную структуру системы, где каждое преобразование данных соответствует одной системной функции. Иногда модели потоков данных используют для описания потоков данных в рабочем окружении системы. Такая модель показывает, как различные системы и подсистемы обмениваются информацией. Подсистемы окружения не обязаны быть простыми функциями.

Рисунок 27 - Диаграмма потоков данных комплекса CASE-средств

Модели конечных автоматов используются для моделирования поведения системы, реагирующей на внутренние или внешние события.

Такая модель показывает состояние системы и события, которые служат причиной перехода системы из одного состояния в другое.

Модели конечных автоматов являются неотъемлемой частью методов проектирования систем реального времени. Такие модели определяются **диаграммами состояний**, которые стали основой системы нотаций в языке моделирования UML.

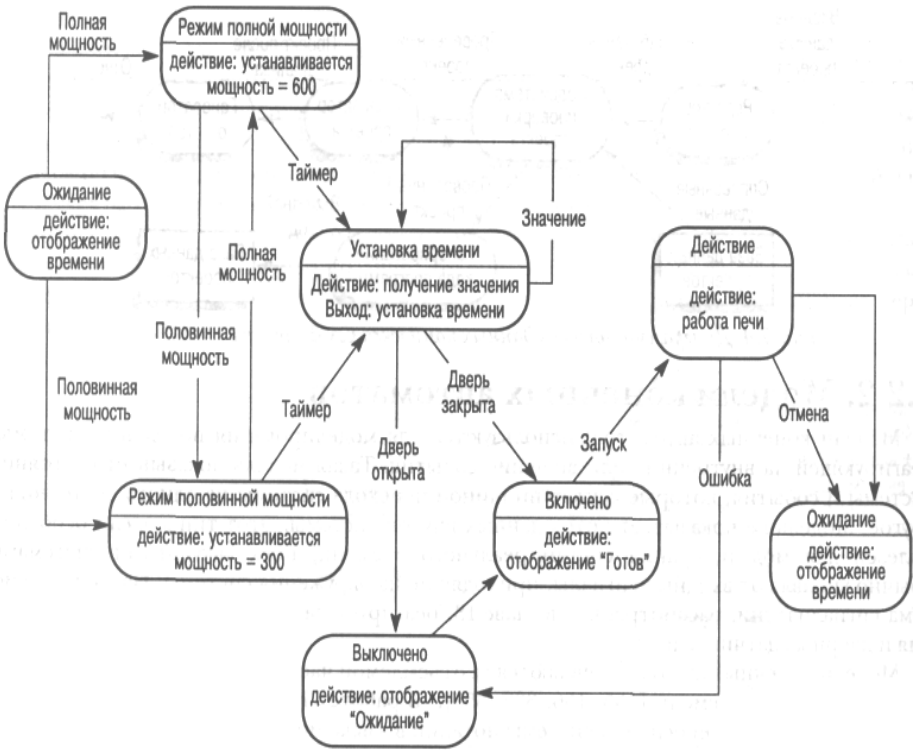


Рисунок 28 – Модель конечного автомата

Наиболее широко используемой методологией **моделирования данных** является моделирование типа **"сущность-связь".** Для описания структуры обрабатываемой информации модели данных часто используются совместно с моделями потоков данных.

Проекты структуры ПО представляются ориентированными графами. Они состоят из набора узлов различных типов, соединенных дугами, отображающими связи между структурными узлами.

Подобно всем графическим моделям, модели "сущность-связь" недостаточно детализированы, поэтому они обычно дополняются более

подробным описанием объектов, связей и атрибутов, включенных в модель. Эти описания собираются в словари данных или репозитории.

Объектные модели, могут использоваться как для представления данных, так и для процессов их обработки. В этом отношении они объединяют модели потоков данных и семантические модели данных. Они также полезны для классификации системных сущностей и могут представлять сущности, состоящие из других сущностей.

Класс объектов - это абстракция множества объектов, которые определяются общими атрибутами и сервисами (операциями).

Объекты - это исполняемые сущности с атрибутами и сервисами класса объектов. Объекты представляют собой реализацию класса. На основе одного класса можно создать много различных объектов.

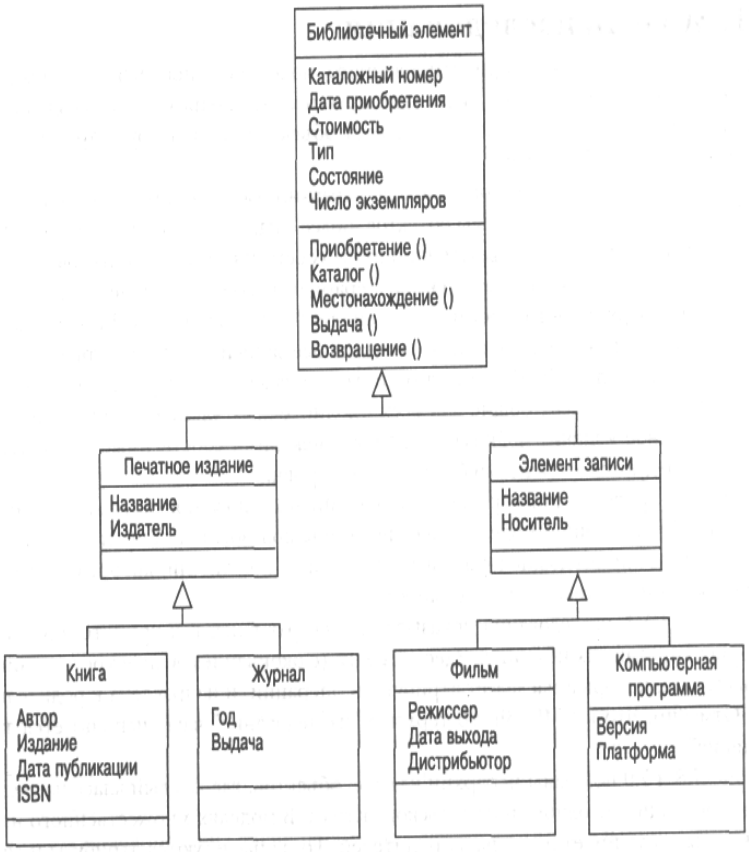


Рисунок 29 – Модель классов

Важным этапом объектно-ориентированного моделирования является определение классов объектов, которые затем систематизируются. Это подразумевает создание схемы классификации, которая показывает, как классы объектов связаны друг с другом посредством общих атрибутов и сервисов.

Схема классификации организована в виде иерархии наследования, на вершине которой представлены наиболее общие классы объектов. Более специализированные объекты наследуют их атрибуты и сервисы. Эти объекты могут иметь собственные атрибуты и сервисы.

В нотации UML наследования показываются сверху вниз. Здесь стрелка (с окончанием в виде треугольника) выходит из класса, который наследует атрибуты и операции, и направлена к родительскому классу.

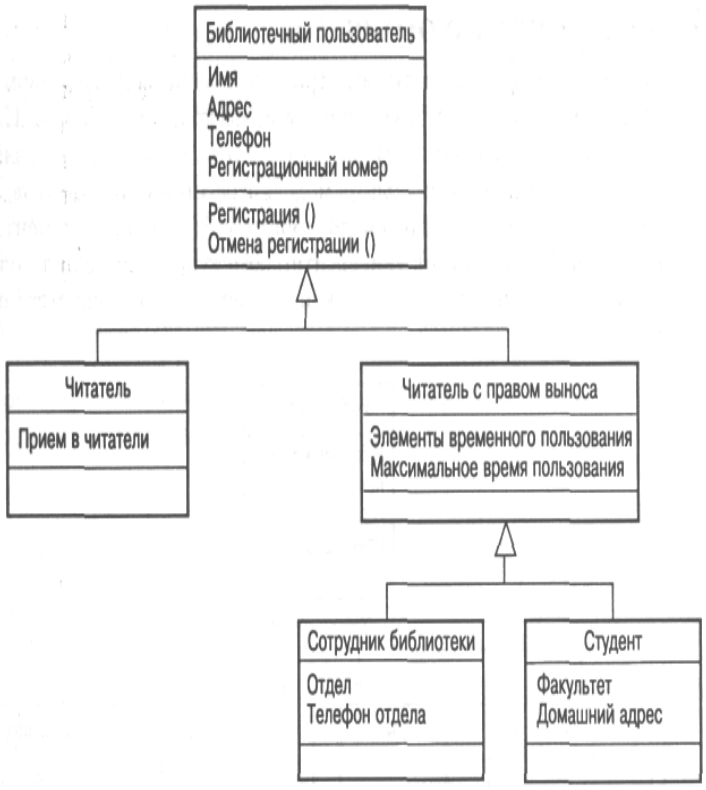


Рисунок 30 - Модели наследования

Объекты могут создаваться из нескольких объектов. Такой объект агрегируется из совокупности других объектов. Классы, представляющие такие объекты, можно смоделировать, используя модель агрегирования объектов.

В UML для показа агрегирования объектов используется связь с окончанием ромбовидной формы.

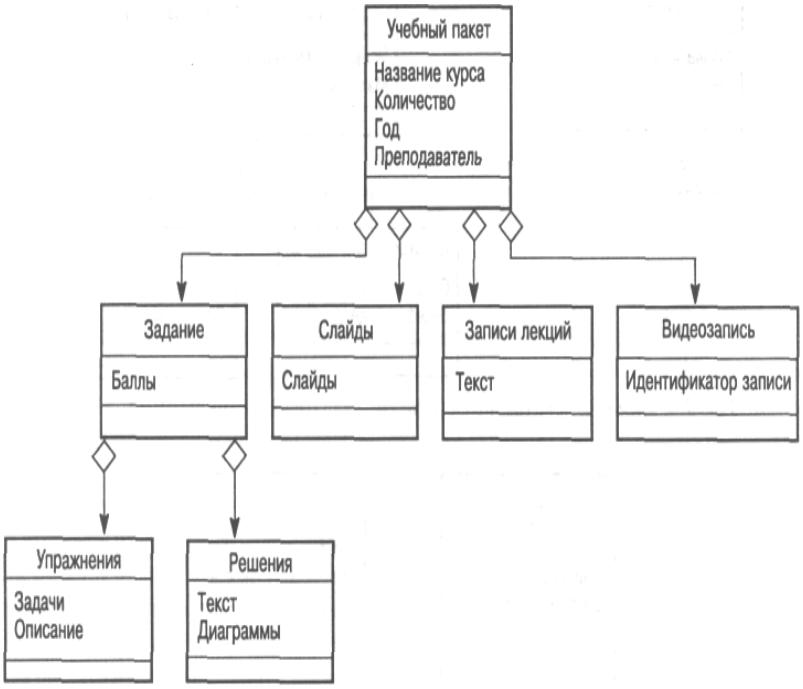


Рисунок 31 - Агрегирование объектов

Модели поведения объектов показывают операции, выполняемые объектами.

В UML поведение объектов моделируется посредством сценариев, которые основаны на вариантах использования.

В верхней части расположены объекты. Операции обозначаются помеченными стрелками, а последовательность операций читается сверху вниз.

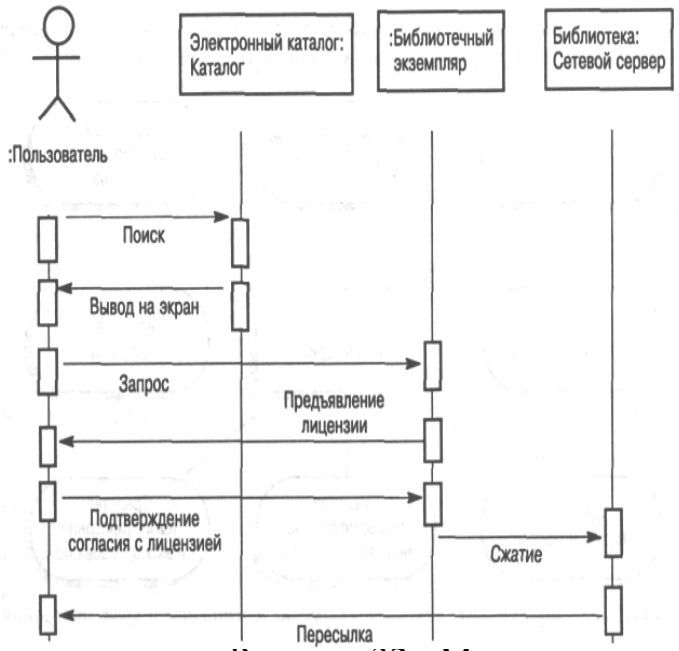


Рисунок 32 - Моделирование поведения объектов

CASE-средства проектирования

Инструментальные средства анализа и проектирования ПО созданы для поддержки моделирования систем на этапах анализа и проектирования процесса разработки программного обеспечения. Они поддерживают создание, редактирование и анализ графических нотаций, используемых в структурных методах.

Рисунок 33 – Структура CASE-системы

Средства, которые входят в пакет инструментальных средств:

- Редакторы диаграмм предназначены для создания диаграмм потоков данных, иерархий объектов, диаграмм "сущность-связь" и т.д.
- Средства проектирования, анализа и проверки выполняют проектирование ПО и создают отчет об ошибках и дефектах в системной архитектуре.
- Центральный репозиторий позволяет проектировщику найти нужный проект и соответствующую проектную информацию.
- Словарь данных хранит информацию об объектах, которые используются в структуре системы.
- Средства генерирования отчетов на основе информации из центрального репозитория автоматически генерируют системную документацию.
- Средства создания форм определяют форматы документов и экранных форм.
- Средства импортирования и экспортирования позволяют обмениваться информацией из центрального репозитория различным инструментальным средствам.
- Генераторы программного кода автоматически генерируют программы на основе проектов, хранящихся в центральном репозитории.

Задания для контроля

1. Разработайте модель рабочего окружения для информационной системы больницы. Модель должна предусматривать ввод данных о новых пациентах и систему хранения рентгеновских снимков.

2. Создайте модель обработки данных в системе электронной почты. Необходимо отдельно смоделировать отправку почты и ее получение.
3. Разработайте модель классов объектов для системы электронной почты. Если вы выполнили упражнение, опишите различия и сходства между моделью обработки данных и объектной моделью.
4. Нарисуйте модель конечного автомата управляющей системы для телефонного автоответчика, который регистрирует входные сообщения и показывает число принятых сообщений на дисплее. Система должна соединять владельца телефона с абонентом после ввода им последовательности чисел (телефонного номера абонента), а также, имея записанные сообщения, повторять их по телефону.

Глава 3. Реализация ПО

Архитектурное проектирование

Архитектурным проектированием называют первый этап процесса проектирования, на котором определяются подсистемы, а также структура управления и взаимодействия подсистем.

Целью архитектурного проектирования является описание архитектуры программного обеспечения. Модель системной архитектуры часто является отправной точкой для создания спецификации различных частей системы. В процессе архитектурного проектирования разрабатывается базовая структура системы, т.е. определяются основные компоненты системы и взаимодействия между ними.

Подсистема — это система (т.е. удовлетворяет "классическому" определению "система"), операции (методы) которой не зависят от сервисов, предоставляемых другими подсистемами. Подсистемы состоят из модулей и имеют определенные интерфейсы, с помощью которых взаимодействуют с другими подсистемами.

Модуль — это обычно компонент системы, который предоставляет один или несколько сервисов для других модулей. Модуль может использовать сервисы, поддерживаемые другими модулями. Как правило, модуль никогда не рассматривается как независимая система. Модули обычно состоят из ряда других, более простых компонентов.

Этапы, общие для всех процессов архитектурного проектирования:

1. Структурирование системы. Программная система структурируется в виде совокупности относительно независимых подсистем. Также определяются взаимодействия между подсистемами.

2. Моделирование управления. Разрабатывается базовая модель управления взаимоотношениями между частями системы.
3. Модульная декомпозиция. Каждая определенная на первом этапе подсистема разбивается на отдельные модули. Здесь определяются типы модулей и типы их взаимосвязей.

Результатом процесса архитектурного проектирования является документ, отображающий архитектуру системы. Он состоит из набора графических схем представлений моделей системы с соответствующим описанием. В описании должно быть указано, из каких подсистем состоит система и из каких модулей слагается каждая подсистема. Графические схемы моделей системы позволяют взглянуть на архитектуру с разных сторон.

Как правило, разрабатывается четыре архитектурные модели:

1. Статическая структурная модель, в которой представлены подсистемы или компоненты, разрабатываемые в дальнейшем независимо.
2. Динамическая модель процессов, в которой представлена организация процессов во время работы системы.
3. Интерфейсная модель, которая определяет сервисы, предоставляемые каждой подсистемой через общий интерфейс.
4. Модели отношений, в которых показаны взаимоотношения между частями системы, например поток данных между подсистемами.

Модели архитектуры могут зависеть от нефункциональных системных требований:

1. Производительность. Если критическим требованием является производительность системы, следует разработать такую архитектуру, чтобы за все критические операции отвечало как

можно меньше подсистем с максимально малым взаимодействием между ними. Чтобы уменьшить взаимодействие между компонентами, лучше использовать крупномодульные компоненты, а не мелкие структурные элементы.

2. Защищенность. В этом случае архитектура должна иметь многоуровневую структуру, в которой наиболее критические системные элементы защищены на внутренних уровнях, а проверка безопасности этих уровней осуществляется на более высоком уровне.
3. Безопасность. В этом случае архитектуру следует спроектировать так, чтобы за все операции, влияющие на безопасность системы, отвечало как можно меньше подсистем. Такой подход позволяет снизить стоимость разработки и решает проблему проверки надежности.
4. Надежность. В этом случае следует разработать архитектуру с включением избыточных компонентов, чтобы можно было заменять и обновлять их, не прерывая работу системы.
5. Удобство сопровождения. В этом случае архитектуру системы следует проектировать на уровне мелких структурных компонентов, которые можно легко изменять. Программы, создающие данные, должны быть отделены от программ, использующих эти данные. Следует также избегать структуры совместного использования данных.

На первом этапе процесса проектирования архитектуры система разбивается на несколько взаимодействующих подсистем. На самом абстрактном уровне архитектуру системы можно изобразить графически с помощью блок-схемы, в которой отдельные подсистемы представлены отдельными блоками. Если подсистему также можно разбить на несколько частей, на диаграмме эти части изображаются прямоугольниками внутри

больших блоков. Потоки данных и/или потоки управления между подсистемами обозначается стрелками. Такая блок-схема дает общее представление о структуре системы.

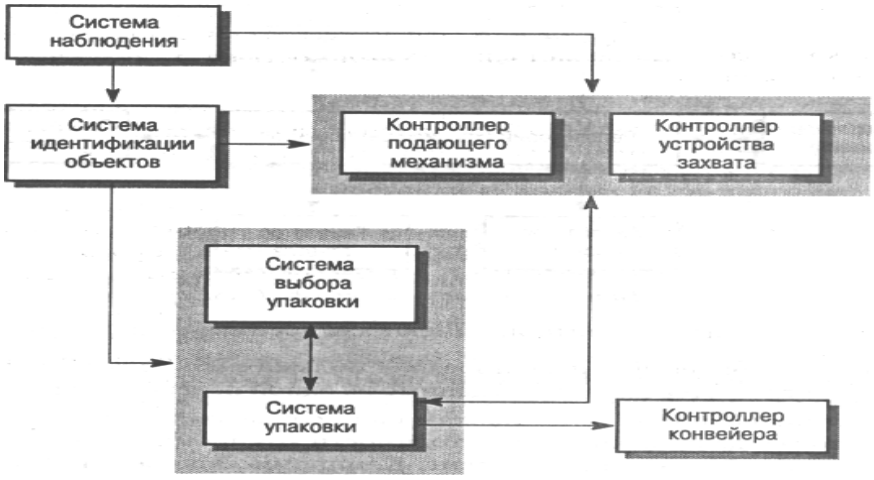


Рисунок 34 - Блок-схема системы управления автоматической упаковкой.

Для того чтобы подсистемы, составляющие систему, работали эффективнее, между ними должен идти обмен информацией. Обмен можно организовать двумя способами:

1. Все совместно используемые данные хранятся в центральной базе данных, доступной всем подсистемам. Модель системы, основанная на совместном использовании базы данных, часто называют моделью репозитория.
2. Каждая подсистема имеет собственную базу данных. Взаимообмен данными между подсистемами происходит посредством передачи сообщений.
3. Модель репозитория
4. Совместное использование больших объемов данных эффективно, поскольку не требуется передавать данные из одной подсистемы в другие.
5. Подсистемы должны быть согласованы с моделью репозитория данных. Это всегда приводит к необходимости компромисса между

требованиями, предъявляемыми к каждой подсистеме. Компромиссное решение может понизить их производительность. Если форматы данных новых подсистем не подходят под согласованную модель представления данных, интегрировать такие подсистемы сложно или невозможно.

6. Подсистемам, в которых создаются данные, не нужно знать, как эти данные используются в других подсистемах.
7. Поскольку в соответствии с согласованной моделью данных генерируются большие объемы информации, модернизация таких систем проблематична. Перевод системы на новую модель данных будет дорогостоящим и сложным, а порой даже невозможным.
8. В системах с репозиторием такие средства, как резервное копирование, обеспечение безопасности, управление доступом и восстановление данных, централизованы, поскольку входят в систему управления репозиторием.
9. К разным подсистемам предъявляются разные требования, касающиеся безопасности, восстановления и резервирования данных. В модели репозитория ко всем подсистемам применяется одинаковая политика.
10. Модель совместного использования репозитория прозрачна: если новые подсистемы совместимы с согласованной моделью данных, их можно непосредственно интегрировать в систему.
11. Сложно разместить репозитории на нескольких машинах, поскольку могут возникнуть проблемы, связанные с избыточностью и нарушением целостности данных.

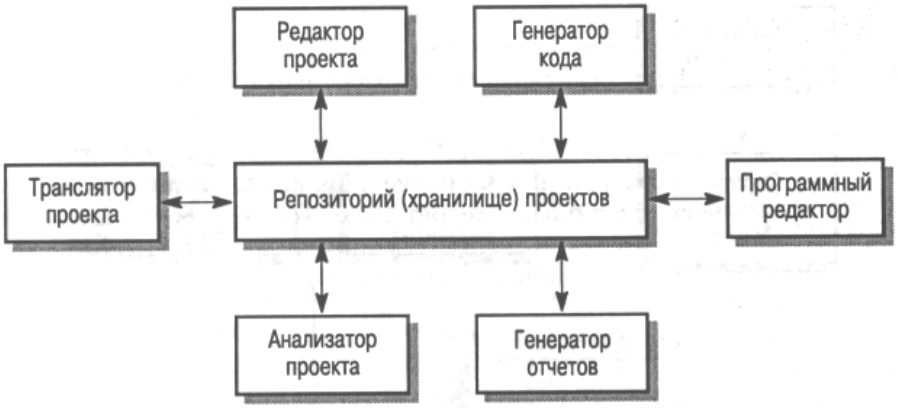


Рисунок 35 - Архитектура интегрированного набора CASE-средств

Модель клиент/сервер

Модель архитектуры клиент/сервер — это модель распределенной системы, в которой показано распределение данных и процессов между несколькими процессорами. Модель включает три основных компонента:

1. Набор автономных серверов, предоставляющих сервисы другим подсистемам.
2. Набор клиентов, которые вызывают сервисы, предоставляемые серверами. В контексте системы клиенты являются обычными подсистемами. Допускается параллельное выполнение нескольких экземпляров клиентской программы.
3. Сеть, посредством которой клиенты получают доступ к сервисам.

Наиболее важное преимущество модели клиент/сервер состоит в том, что она является распределенной архитектурой. Ее эффективно использовать в сетевых системах с множеством распределенных процессоров. В систему легко добавить новый сервер и интегрировать его с остальной частью системы или же обновить серверы, не воздействуя на другие части системы.

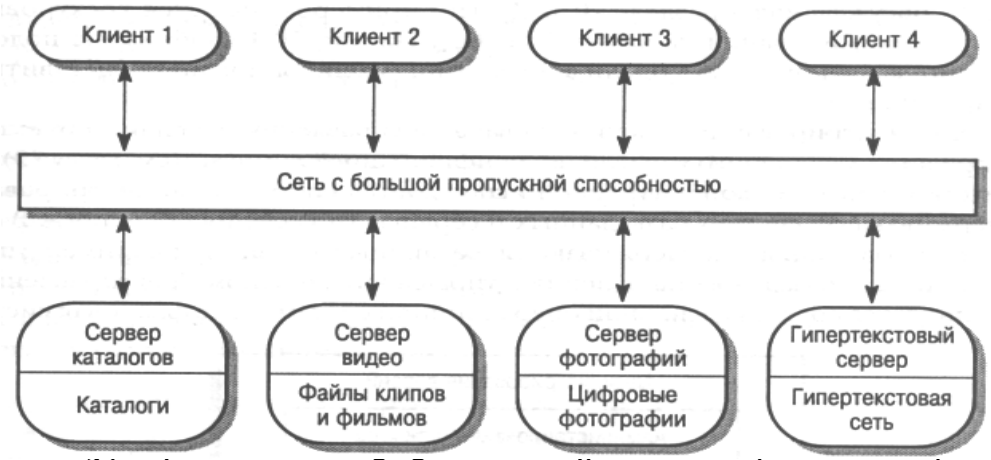


Рисунок 36 - Архитектура библиотечной системы фильмов и фотографий

Модель абстрактной машины

Модель архитектуры абстрактной машины (иногда называемая многоуровневой моделью) моделирует взаимодействие подсистем. Она организует систему в виде набора уровней, каждый из которых предоставляет свои сервисы. Каждый уровень определяет абстрактную машину, машинный язык которой (сервисы, предоставляемые уровнем) используется для реализации следующего уровня абстрактной машины.

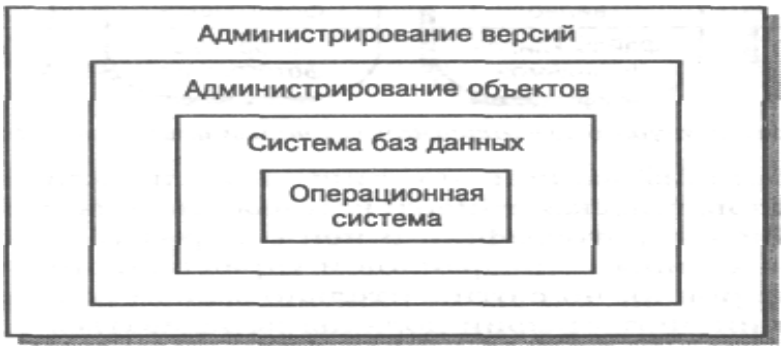


Рисунок 37 - Модель абстрактной машины для системы администрирования версий

В структурных моделях нет (и не должно быть) никакой информации по управлению. Однако разработчик архитектуры должен организовать подсистемы согласно некоторой модели управления, которая дополняла бы имеющуюся модель структуры. В моделях управления на уровне архитектуры проектируется поток управления между подсистемами.

Можно выделить два основных типа управления:

1. Централизованное управление. Одна из подсистем полностью отвечает за управление, запускает и завершает работу остальных подсистем.
2. Управление, основанное на событиях. Здесь вместо одной подсистемы, ответственной за управление, на внешние события может отвечать любая подсистема. События, на которые реагирует система, могут происходить либо в других подсистемах, либо во внешнем окружении системы.

Централизованное управление

В модели централизованного управления одна из систем назначается главной и управляет работой других подсистем. Такие модели можно разбить на два класса:

Модель вызова-возврата. Это известная модель организации вызова программных процедур "сверху вниз", в которой управление начинается на вершине иерархии процедур и через вызовы передается на более нижние уровни иерархии. Данная модель применима только в последовательных системах.

Модель диспетчера. Применяется в параллельных системах. Один системный компонент назначается диспетчером и управляет запуском, завершением и координированием других процессов системы. Процесс может протекать параллельно с другими процессами. Модель такого типа применима также в последовательных системах, где управляющая программа вызывает отдельные подсистемы в зависимости от значений некоторых переменных состояния.

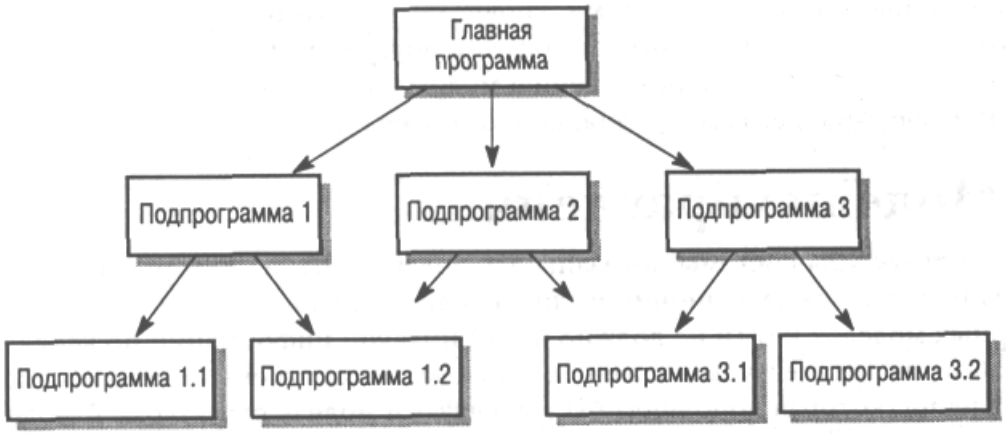


Рисунок 38 - Модель вызова-возврата

Рисунок 39 - Модель диспетчера для системы реального времени

Управление, основанное на событиях

В моделях централизованного управления, как правило, управление системой определяется значениями некоторых переменных ее состояния. В противоположность таким моделям существуют системы, управление которыми основано на внешних событиях.

Модели передачи сообщений. В этих моделях событие представляет собой передачу сообщения всем подсистемам. Любая подсистема, которая обрабатывает данное событие, отвечает на него.

Модели, управляемые прерываниями. Такие модели обычно используются в системах реального времени, где внешние прерывания регистрируются обработчиком прерываний, а обрабатываются другим системным компонентом.

Рисунок 40 - Модель управления, основанная на передаче сообщений

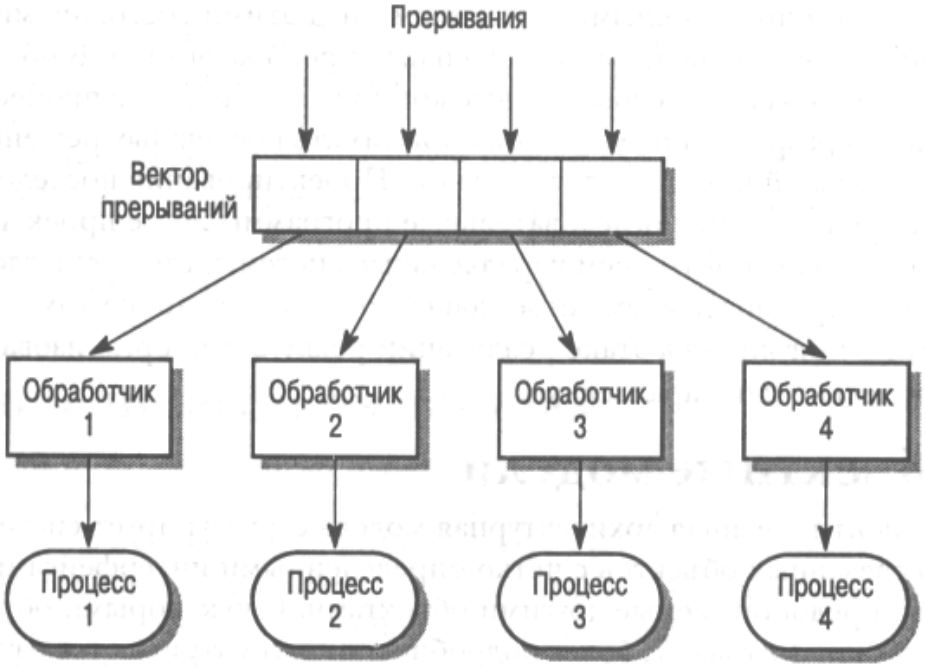


Рисунок 41 - Модель управления, основанная на прерываниях

После этапа разработки системной структуры в процессе проектирования следует этап декомпозиции подсистем на модули.

Распространены две модели, используемые на этапе модульной декомпозиции подсистем:

1. Объектно-ориентированная модель. Система состоит из набора взаимодействующих объектов.
2. Модель потоков данных. Система состоит из функциональных модулей, которые получают на входе данные и преобразуют их некоторым образом в выходные данные. Такой подход часто называется конвейерным.

В объектно-ориентированной модели модули представляют собой объекты с собственными состояниями и определенными операциями над этими состояниями. В модели потоков данных модули выполняют функциональные преобразования.

Объектные модели

Объектно-ориентированная архитектурная модель структурирует систему в виде совокупности слабо связанных объектов с четко определенными интерфейсами. Объекты вызывают сервисы, предоставляемые другими объектами.

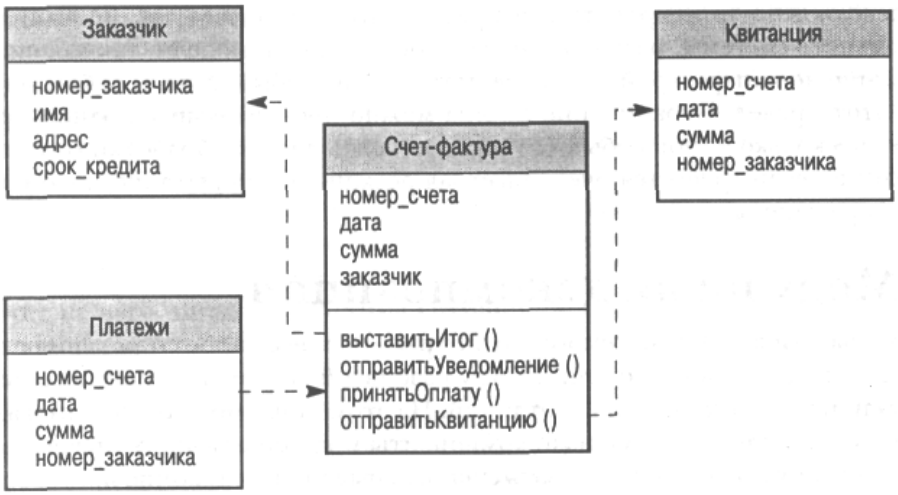


Рисунок 42 - Объектная модель системы обработки счетов

Модели потоков данных

Данные проходят через последовательность преобразований. Каждый шаг обработки данных реализован в виде преобразования. Данные, поступающие на вход системы, проходят через все преобразования и достигают выхода системы. Преобразования могут выполняться последовательно или параллельно. Обработка данных может быть пакетной или поэлементной.

Рисунок 43 - Модель потоков данных для системы обработки счетов

Наряду с основными моделями, используются архитектурные модели, характерные для конкретной предметной области приложения. Эти модели называются проблемно-**зависимыми архитектурами.**

Можно выделить два типа проблемно-зависимых архитектурных моделей:

1. Модели классов систем. Отображают классы реальных систем, вобрав в себя основные характеристики этих классов. Как правило, архитектурные модели классов встречаются в системах реального времени, например в системах сбора данных, мониторинга и т.д.

2. Базовые модели. Более абстрактны и предоставляют разработчикам информацию по общей структуре какого-либо типа систем.

Модели классов систем

Модель компилятора наиболее известный пример архитектурной модели класса систем. Компоненты, из которых состоит компилятор, можно организовывать в соответствии с разными архитектурными моделями. Можно применить архитектуру потоков данных, в которой таблица идентификаторов служит хранилищем совместно используемых данных.

Рисунок 44 – Модель компилятора

Однако такие модели оказываются менее эффективными, если компилятор интегрирован с другими языковыми средствами, например системой редактирования структур, интерактивным отладчиком, программой подготовки печатных документов и т.п. В этом случае компоненты системы можно организовать в соответствии с моделью репозитория.

Рисунок 45 – Модель репозитория

Базовые архитектуры

Базовые модели представляют собой идеализированную архитектуру, в которой отражены особенности, присущие системам, работающим в данной предметной области.

Примером базовой архитектуры может служить модель OSI.

Базовые модели обычно не рассматриваются в качестве методов реализации. Их основное назначение — служить эталоном для сравнения различных систем в какой-либо предметной области, т.е. базовая модель является стандартом при оценке различных систем.

Вопросы для обсуждения

1. Объясните, почему архитектуру системы необходимо разработать до окончания создания спецификации.
2. Предложите подходящую структурную модель для системы видеоконференций, управляемой компьютером, с возможностью одновременного просмотра компьютерных, аудио- и видеоданных несколькими участниками. Обоснуйте свой выбор.

3. Объясните, почему модель управления вызова-возврата обычно не подходит для систем реального времени, управляющих определенным процессом.
4. Предложите подходящую модель управления для набора инструментальных программных средств от разных производителей, которые должны работать совместно. Обоснуйте свой выбор.
5. Предположим, существует конкретная должность "архитектор программного обеспечения"; его роль состоит в проектировании системной архитектуры независимо от того, для какого заказчика выполняется данный проект. Какие трудности могут возникнуть при введении данной должности?

Проектирование с повторным использованием компонентов

В большинстве инженерных разработок процесс проектирования основан на повторном использовании уже имеющихся компонентов.

Повторное использование компонентов позволит существенно сократить расходы на разработку ПО. Только с помощью систематического повторного использования ПО можно уменьшить расходы на его создание и обслуживание, сократить сроки разработки систем и повысить качество программных продуктов.

Чтобы повторное использование ПО было эффективным, его необходимо учитывать на всех этапах процесса проектирования ПО или процесса разработки требований. Во время программирования возможно повторное использование на этапе подбора компонентов, соответствующих требованиям. Однако для **систематического** повторного использования необходим такой процесс проектирования, в ходе которого постоянно рассматривалась бы возможность повторного использования уже существующих архитектур, где система была бы явно организована из имеющихся компонентов ПО.

Метод проектирования ПО, основанный на повторном использовании, предполагает максимальное использование уже имеющихся программных объектов:

1. Повторно используемые приложения. Можно повторно использовать целые приложения либо путем включения их в систему без изменения других подсистем, либо с помощью разработки семейств приложений, работающих на разных платформах и адаптированных к требованиям конкретных заказчиков.
2. Повторно используемые компоненты. Можно повторно использовать компоненты приложений — от подсистем до отдельных объектов.
3. Повторно используемые функции. Можно повторно использовать программные компоненты, которые реализуют отдельные функции.

Очевидным преимуществом повторного использования ПО является снижение общей стоимости проекта, так как в целом требуется специфицировать, спроектировать, реализовать и проверить меньшее количество системных компонентов. Но снижение стоимости проекта — это только потенциальное преимущество повторного использования.

Таблица 3 - Преимущества повторного использования кода

Повышение надежности	Компоненты, повторно используемые в других системах, оказываются значительно надежнее новых компонентов. Они протестированы и проверены в разных условиях работы. Ошибки, допущенные при их проектировании и реализации,

	обнаружены и устранены еще при первом их применении.
Уменьшение проектных рисков	Для уже существующих компонентов можно более точно прогнозировать расходы, связанные с их повторным использованием, чем расходы, необходимые на их разработку. Такой прогноз — важный фактор администрирования проекта, так как позволяет уменьшить неточности при предварительной оценке сметы проекта.
Эффективное использование специалистов	Часть специалистов, выполняющих одинаковую работу в разных проектах, может заниматься разработкой компонентов для их дальнейшего повторного использования, эффективно применяя накопленные ранее знания.
Соблюдение стандартов	Некоторые стандарты можно реализовать в виде набора стандартных компонентов. Использование стандартного пользовательского интерфейса повышает надежность систем, так как, работая со знакомым интерфейсом, пользователи совершают меньше ошибок.

Ускорение разработки	Часто для успешного продвижения системы на рынке необходимо как можно более раннее ее появление, причем независимо от полной стоимости ее создания. Повторное использование компонентов ускоряет создание систем, так как сокращается время на их разработку и тестирование

Для успешного проектирования и разработки ПО с повторным использованием компонентов должны выполняться три основных условия:

1. Возможность поиска необходимых системных компонентов.
2. При повторном использовании необходимо удостовериться, что поведение компонентов предсказуемо и надежно. В идеале все компоненты должны быть сертифицированы, чтобы подтвердить соответствие определенным стандартам качества.
3. На каждый компонент должна быть соответствующая документация, цель которой — помочь разработчику получить нужную информацию о компоненте и адаптировать его к новому приложению.

Таблица 4 - Проблемы повторного использования

Повышение стоимости сопровождения системы	Недоступность исходного кода компонента может привести к увеличению расходов на сопровождение системы, так как повторно используемые системные элементы могут со временем оказаться не совместимыми с изменениями, производимыми в системе.

Недостаточная инструментальная поддержка	CASE-средства не поддерживают разработку ПО с повторным использованием компонентов.
Синдром "изобретения велосипеда"	Некоторые разработчики предпочитают переписать компоненты, так как полагают, что смогут при этом их усовершенствовать. Кроме того, многие считают, что создание программ "с нуля" перспективнее и "благороднее" повторного использования написанных другими программ.
Содержание библиотеки компонентов	Заполнение библиотеки компонентов и ее сопровождение может стоить дорого. В настоящее время еще недостаточно хорошо продуманы методы классификации, каталогизации и извлечения информации о программных компонентах
Поиск и адаптация компонентов	Компоненты ПО нужно найти в библиотеке, изучить и адаптировать к работе в новых условиях, что "не укладывается" в обычный процесс разработки ПО

Покомпонентная разработка

1. Компонент — это независимо выполняемый программный объект. Исходный код компонента может быть недоступен, поэтому такой компонент не компилируется совместно с другими компонентами системы.

2. Компоненты объявляют свой интерфейс и все взаимодействия с ними осуществляются с его помощью. Интерфейс компонента описывается в терминах параметризованных операций, а внутреннее состояние компонента всегда скрыто.

Компоненты определяются через свои интерфейсы. В большинстве случаев компоненты можно описать в виде двух взаимосвязанных интерфейсов:

1. Интерфейс поставщика сервисов, который определяет сервисы, предоставляемые компонентом.
2. Интерфейс запросов, который определяет, какие сервисы доступны компоненту из системы, использующей этот компонент.

Рисунок 46 - Компонент службы печати

Компоненты могут существовать на разных уровнях абстракции:

– Функциональная абстракция. Компонент реализует отдельную функцию. В сущности, интерфейсом поставщика сервисов здесь является сама функция.

– Бессистемная группировка. В данном случае компонент — это набор слабо связанных между собой программных объектов и подпрограмм, например объявлений данных, функций и т.п. Интерфейс поставщика сервисов состоит из названий всех объектов в группировке.

– Абстракции данных. Компонент является абстракцией данных или классом, описанным на объектно-ориентированном языке. Интерфейс поставщика сервисов состоит из методов (операций), обеспечивающих создание, изменение и получение доступа к абстракции данных.

– Абстракции кластеров. Здесь компонент — это группа связанных классов, работающих совместно. Такие компоненты иногда называют структурой. Интерфейс поставщика сервисов является композицией всех интерфейсов объектов, составляющих структуру.

– Системные абстракции. Компонент является полностью автономной системой. Повторное использование абстракций системного уровня иногда называют повторным использованием коммерческих продуктов. Интерфейсом поставщика сервисов на этом уровне является так называемый программный интерфейс приложений (Application Programming Interface — API), который предоставляет доступ к системным командам и методам.

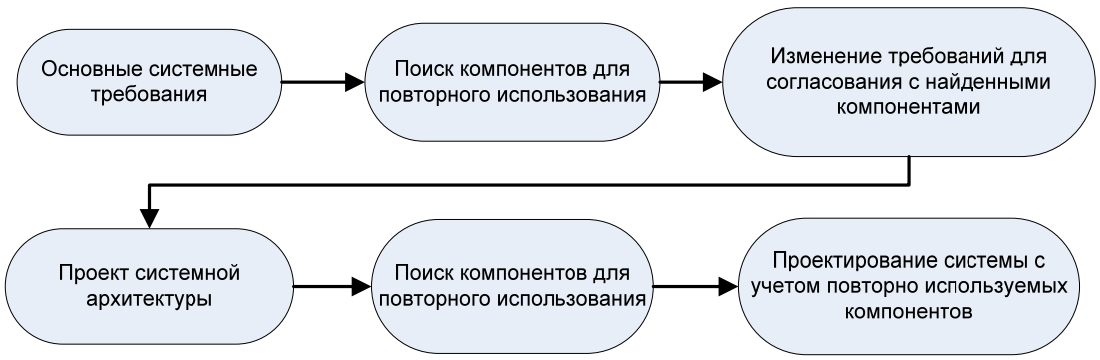


Рисунок 47 - Процесс проектирования с повторным использованием компонентов:

Объектные структуры приложений

Существует три основных класса объектных структур:

1. Инфраструктуры систем. Обеспечивают разработку инфраструктур для систем связи (коммуникационных систем), пользовательских интерфейсов и компиляторов.
2. Интеграционные структуры. Как правило, состоят из набора стандартов и связанных с ними классов объектов, обеспечивающих взаимодействие и обмен данными между компонентами.
3. Структуры инструментальных сред разработки приложений. Связаны с отдельными прикладными областями, такими как телекоммуникации или финансы. Они встраиваются в систему знаний области приложения и поддерживают разработку приложений конечного пользователя.

Одной из самых известных и распространенных объектных структур для графических интерфейсов пользователя (GUI) является объектная структура «модель-представление-контроллер»:

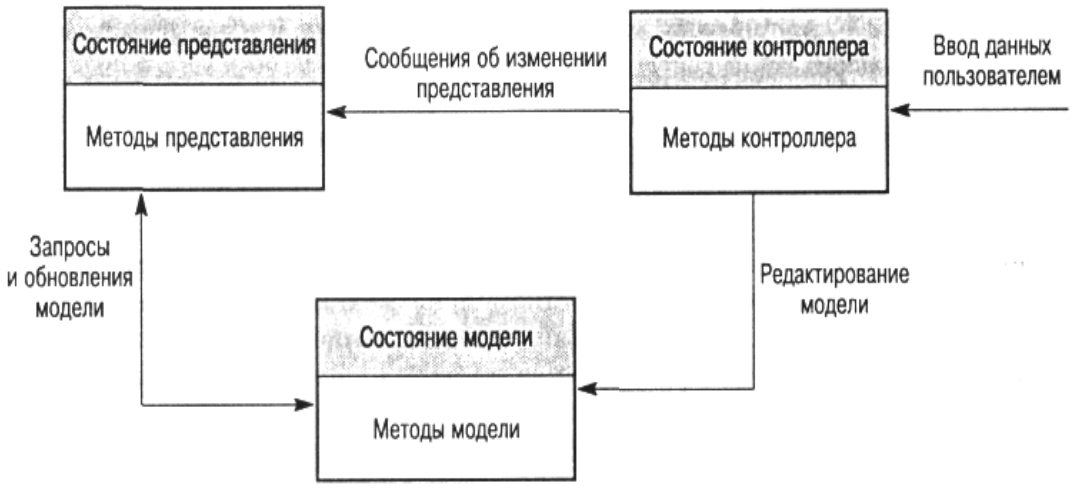


Рисунок 48 – Пример объектной структуры

Основные недостатки объектных структур — их сложность и время, необходимое для того, чтобы научиться работать с ними. Для полного изучения объектных структур может понадобиться несколько месяцев. Именно поэтому в больших организациях некоторые разработчики ПО специализируются по объектным структурам. Нет никаких сомнений в эффективности данного подхода к повторному использованию, однако высокие затраты на изучение объектных структур ограничивают его повсеместное распространение.

Повторное использование коммерческих программных продуктов

Термин "коммерческие программные продукты" можно применить к любому компоненту, созданному независимым производителем.

В принципе использование коммерческих систем не отличается от использования любого другого более специализированного компонента. Для этого необходимо изучить интерфейсы системы и использовать их для организации взаимодействия с другими системными компонентами, также необходимо разработать системную архитектуру, которая поддерживала бы коммерческие системы при совместной работе.

Разработка повторно используемых компонентов

Разработка идеального компонента для повторного использования должна быть процессом (основанным на опыте и знаниях о проблемах повторного использования) создания обобщенных компонентов, которые можно адаптировать для разных вариантов их использования.

Программный компонент, предназначенный для повторного использования, имеет **ряд особенностей**:

- Должен отражать стабильные абстракции предметной области, т.е. фундаментальные понятия области приложения, которые меняются медленно.
- Должен скрывать способ представления своего состояния и предоставлять операции, которые позволяют обновлять состояния и получать к нему доступ.
- Должен быть максимально независимым. В идеале компонент должен быть настолько автономным, чтобы не нуждаться в других компонентах.
- Все исключительные ситуации должны быть частью интерфейса компонента. Компоненты не должны сами обрабатывать исключения, так как в разных приложениях существуют разные требования для обработки исключительных ситуаций. Лучше определить те исключения, которые необходимо обрабатывать, и объявить их как часть интерфейса компонента.

Семейства приложений

Один из наиболее эффективных подходов к повторному использованию базируется на понятии семейства приложений. Семейство приложений, или серия программных продуктов, — это набор приложений, имеющих общую архитектуру, отражающую специфику конкретной предметной области. Вместе с тем все приложения одной серии различны. Каждый раз при создании нового приложения повторно используется общее ядро семейства приложений.

Существуют различные специализации семейств приложений:

1. Платформенная специализация, при которой для разных платформ разрабатываются свои версии приложения.

2. Конфигурационная специализация, при которой разные версии приложения создаются для управления различными периферийными устройствами.
3. Функциональная специализация, при которой создаются разные версии приложения для заказчиков с различными требованиями.

Процесс адаптации семейства приложений для создания нового приложения состоит из нескольких этапов:

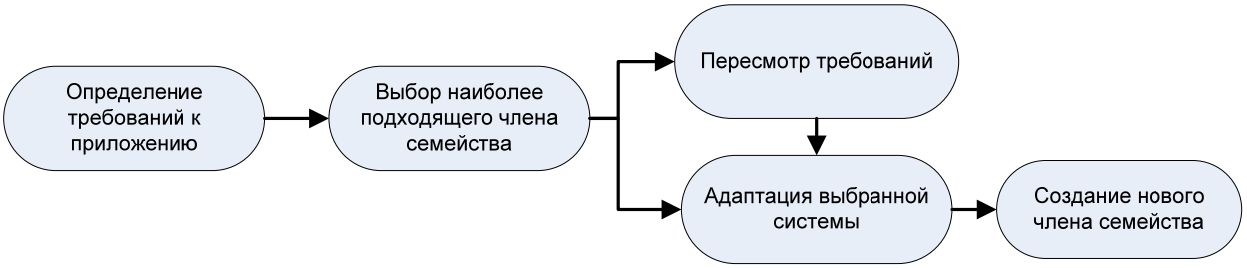


Рисунок 49 – Процесс адаптации семейства приложений

Проектные паттерны

Попытки повторно использовать действующие компоненты постоянно ограничиваются конкретными решениями, принятыми системными разработчиками. Один из способов решения данной проблемы — повторное использование более абстрактных структур, не содержащих деталей реализации. Такие структуры разрабатываются специально для того, чтобы соответствовать определенному приложению.

Паттерн — это описание проблемы и метода ее решения, позволяющее в дальнейшем использовать это решение в разных условиях. Паттерн не является детальной спецификацией. Скорее, он представляет собой описание, в котором аккумулированы знания и опыт. Паттерн — решение общей проблемы.

При создании ПО проектные паттерны всегда связаны с объектно-ориентированным проектированием. Чтобы обеспечить всеобщность,

паттерны часто используют такие объектно-ориентированные понятия, как наследование и полиморфизм. Однако общий принцип использования паттернов одинаково применим при любом подходе к проектированию ПО.

Основные элементы проектного паттерна:

1. Содержательное имя, которое является ссылкой на паттерн.
2. Описание проблемной области с перечислением всех ситуаций, в которых можно использовать паттерн.
3. Описание решений с отдельным описанием различных частей решения и их взаимоотношений. Это не описание конкретного проекта, а шаблон проектных решений, который можно использовать различными способами.
4. Описание "выходных" результатов — это описание результатов и компромиссов, необходимых для применения паттерна. Обычно используется для того, чтобы помочь разработчикам оценить конкретную ситуацию и выбрать для нее наиболее подходящий паттерн.

Часто в описание паттерна вводятся также разделы **мотивации** (обоснование полезности паттерна) и **применимости** (описание ситуаций, в которых можно использовать паттерн).

В качестве примера рассмотрим паттерн Обозреватель. Данный паттерн используется тогда, когда необходимы разные представления состояния объекта. Он выделяет нужный объект и представляет его в разных формах:

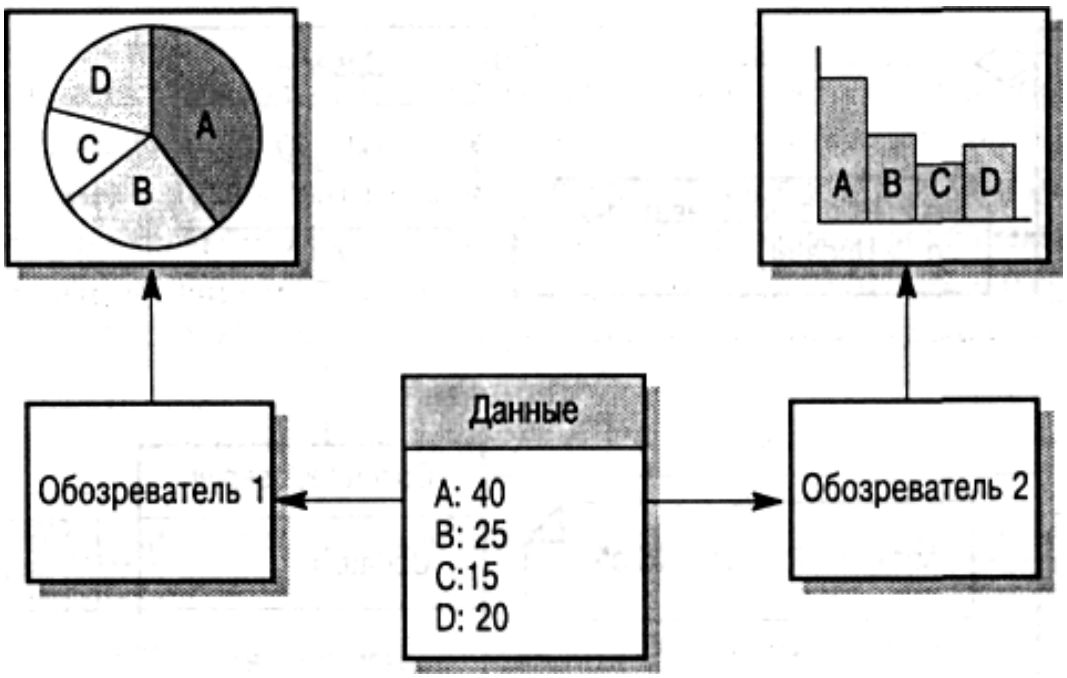


Рисунок 50 - Паттерн Обозреватель - используется тогда, когда необходимы разные представления состояния объекта. Он выделяет нужный объект и представляет его в разных формах.

Проектирование интерфейса пользователя

Проектирование вычислительных систем охватывает широкий спектр проектных действий — от проектирования аппаратных средств до проектирования интерфейса пользователя.

Организации-разработчики часто нанимают специалистов для проектирования аппаратных средств и очень редко для проектирования интерфейсов.

Виды интерфейсов: **текстовый** и **графический.**

Графические интерфейсы обладают рядом преимуществ:

- Их относительно просто изучить и использовать. Пользователи, не имеющие опыта работы с компьютером, могут легко и быстро научиться работать с графическим интерфейсом.

– Каждая программа выполняется в своем окне (экране). Можно переключаться из одной программы в другую, не теряя при этом данные, полученные в ходе выполнения программ.

– Режим полноэкранного отображения окон дает возможность прямого доступа к любому месту экрана.

На схеме изображен итерационный процесс проектирования пользовательского интерфейса. Наиболее эффективным подходом к проектированию интерфейса пользователя является разработка с применением моделирования пользовательских функций.

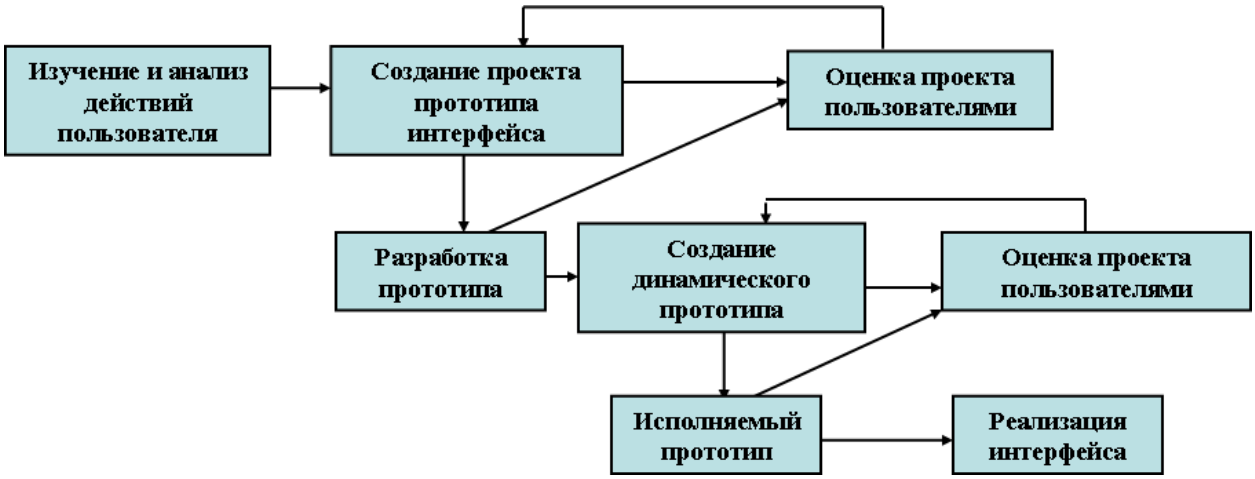


Рисунок 51 – Процесс проектирования интерфейса пользователя

Таблица 5 - Принципы проектирования интерфейсов

Принцип	Описание
Учёт знаний пользователя	Необходимо использовать термины и понятия, взятые из опыта будущих пользователей системы, а объекты, управляемые системой, должны быть напрямую связаны с рабочей средой пользователя.
Согласованность	Команды и меню системы должны быть одного формата, параметры должны

	передаваться во все команды одинаково и пунктуация команд должна быть схожей. Удобно, когда для всех типов объектов системы поддерживаются одинаковые методы.
Минимум неожиданностей	Одно и тоже действие в различных ситуациях должно приводить к аналогичной реакции.
Способность к восстановлению	В интерфейсах должны быть средства предотвращающие ошибки пользователя, а также позволяющие корректно восстановить информацию после ошибок. Это подтверждение деструктивных действий и возможность отмены действий.
Руководство пользователя	Интерфейс должен предоставлять необходимую информацию в случае ошибок пользователя и поддержать средства контекстно-зависимой справки.
Учёт разнородности пользователей	В интерфейсе должны быть средства для удобного взаимодействия с пользователями, имеющими разный уровень квалификации и различные возможности.

Разработчиками интерфейсов предусмотрены 5 основных стилей взаимодействия пользователя с системой.

Таблица 6 – Стили взаимодействия с пользователем

1.	Непосредственное манипулирование	

2.	Выбор из меню	
3.	Заполнение форм	
4.	Командный язык	
5.	Естественный язык	

Таблица 7 - Преимущества и недостатки стилей взаимодействия

Стиль взаимодействия	**Основные преимущества**	**Основные недостатки**	**Примеры приложений**
Прямое манипулирование	Быстрое и интуитивно понятное взаимодействиеЛегок в изучении	Сложная реализация. Подходит только там, где есть зрительный образ задач и объекта	Видеоигры; системы автоматического проектирования
Выбор из меню	Сокращение количества ошибок пользователя.	Медленный вариант для опытных пользователей. Может быть сложным, если меню состоит из большого количества вложенных пунктов	Главным образом системы общего назначения
Заполнение форм	Ввод с клавиатуры	Занимает пространство	Системы управления запасами; обработка

	минимальный	на экране	финансовой информации
Командный язык	Простой ввод данных.	Труден в изучении.	Операционные системы; библиотечные системы
Естественный язык	Легок в изучении, мощный и гибкий. Подходит неопытным пользователям. Легок в настройке	Сложно предотвратить ошибки ввода Требует большого ручного набора	Системы расписания; системы хранения данных WWW

Модель с разделенными интерфейсом командного языка и графическим интерфейсом лежит в основе некоторых операционных систем, в частности Linux.

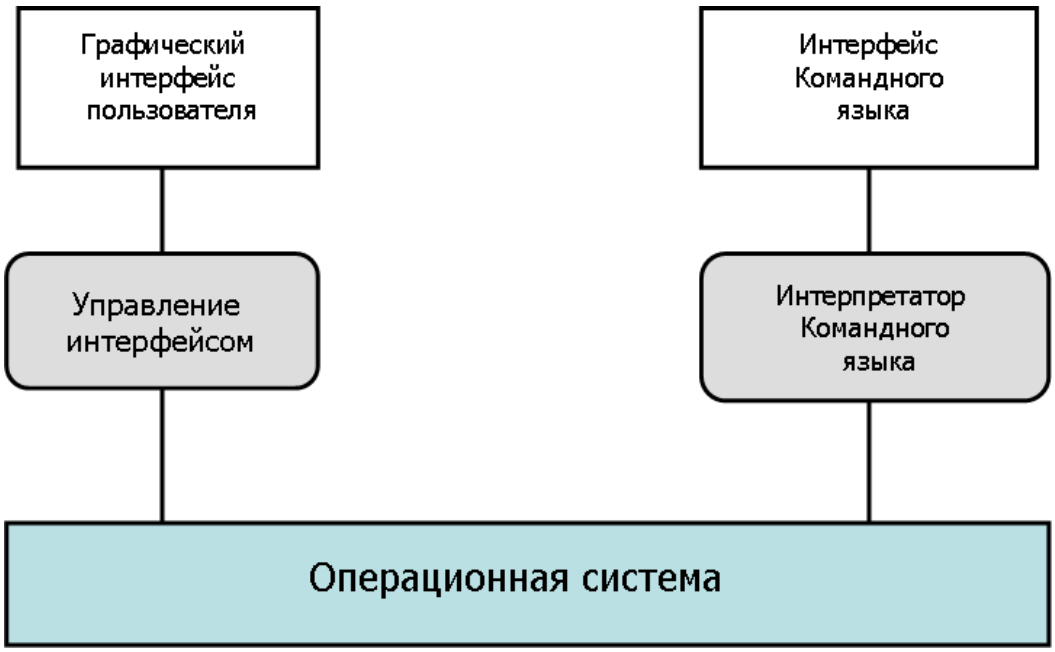


Рисунок 52 - Модель с разделенными интерфейсом

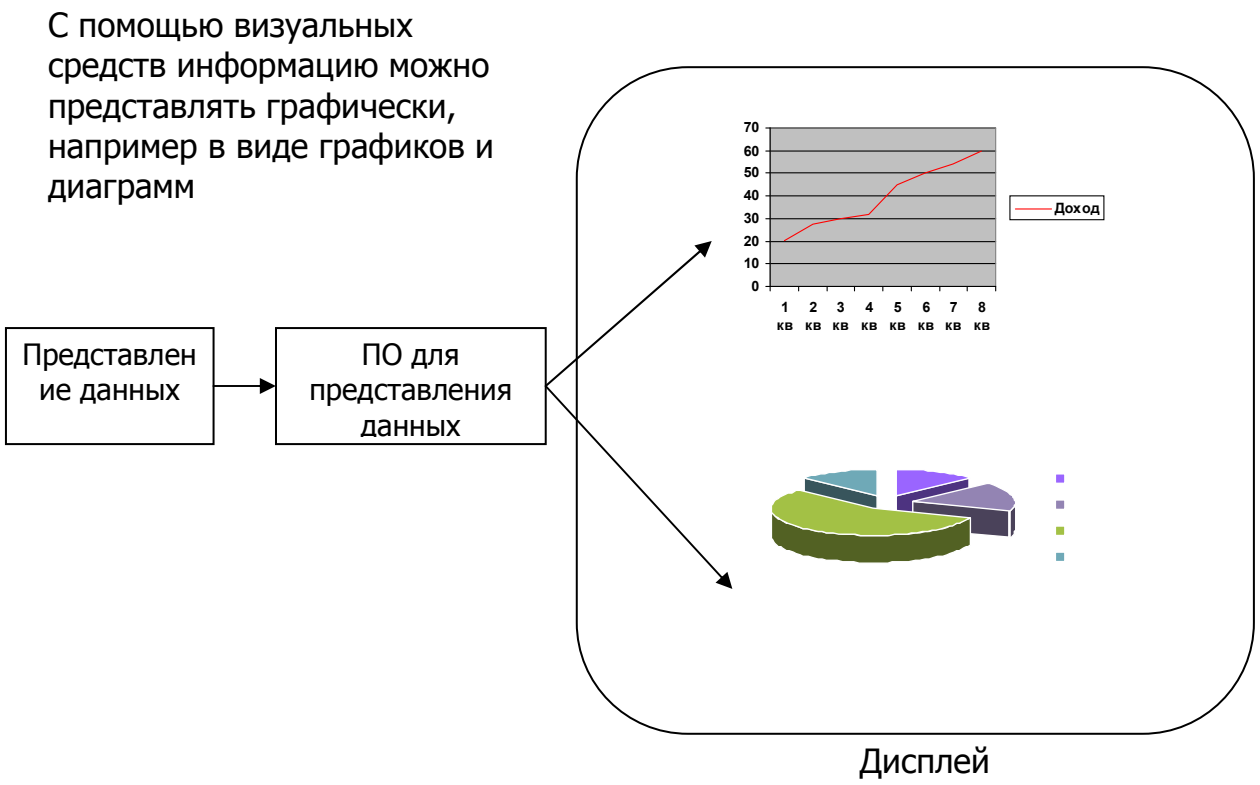


Рисунок 53 – Модель представления информации

Принимая решение по представлению данных, разработчик должен учитывать ряд факторов:

– Что нужно пользователю: точные значения данных или соотношения между значениями?

– Насколько быстро будут происходить изменения значений данных? Нужно ли немедленно показывать пользователю изменение значений?

– Должен ли пользователь предпринимать какие-либо действия в ответ на изменение данных?

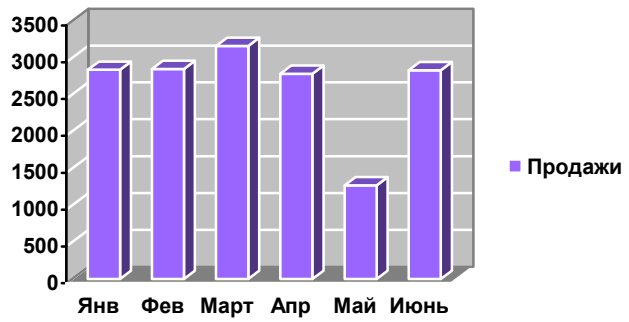
– Нужно ли пользователю взаимодействовать с отображаемой информацией посредством интерфейса с прямым манипулированием?

– Информация должна отображаться в текстовом (описательно) или числовом формате? Важны ли относительные значения элементов данных?

Альтернативы

Часто визуальное представление информации нагляднее, чем табличный аналог

Янв	Фев	Март	Апр	Май	Июль
2842	2851	3164	2789	1273	2835



Использование в интерфейсах цвета:

– Используйте ограниченное количество цветов

– Используйте разные цвета для показа изменений в состоянии системы

– Для помощи пользователю используйте цветовое кодирование

– Используйте цветовое кодирование продуманно и последовательно

– Осторожно используйте дополняющие цвета

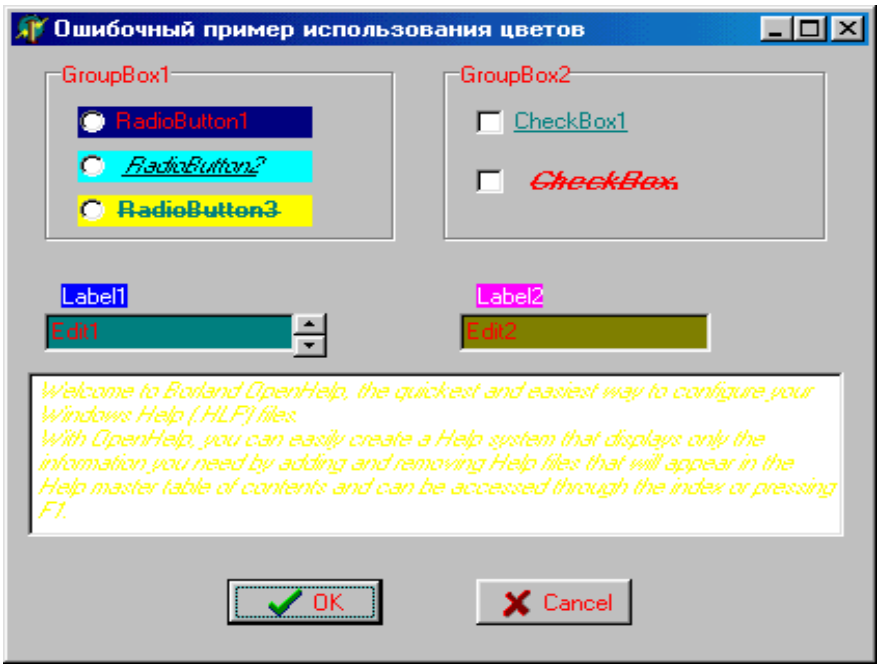


Рисунок 54 - Пример неправильного использования цветов

Средства поддержки пользователя

Одним из основных аспектов проектирования пользователя является **справочная система**. Справочную систему приложения составляют:

- сообщения, генерируемые системой в ответ на действия пользователя;
- диалоговая справочная система;
- документация, поставляемая с системой.

Таблица 8 - Факторы проектирования текстовых сообщений

Фактор	**Описание**
Содержание	Справочная система должна знать, что делает пользователь, и реагировать на его действия сообщениями соответствующего содержания

Профессиональный уровень пользователя	Сообщения должны содержать сведения, соответствующие профессиональному уровню пользователей.
Опыт пользователя	Если пользователи хорошо знакомы с системой, им не нужны длинные и подробные сообщения. В то же время начинающим пользователям такие сообщения покажутся сложными, мало понятными и слишком краткими.
Фактор	**Описание**
Стиль сообщений	Сообщения должны иметь положительный, а не отрицательный оттенок. Всегда следует использовать активный, а не пассивный тон обращения. Не должно быть оскорблений или попыток пошутить
Культура	Разработчик сообщений должен быть знаком с культурой той страны, где продается система. Сообщение, вполне уместное в культуре одной страны, может оказаться неприемлемым в другой

Таблица 9 - Факторы проектирования текстовых сообщений

Фактор	**Описание**
Содержание	Справочная система должна знать, что делает пользователь, и реагировать на его действия сообщениями соответствующего содержания
Профессиональный уровень пользователя	Сообщения должны содержать сведения, соответствующие профессиональному уровню пользователей.
Опыт пользователя	Если пользователи хорошо знакомы с системой, им не нужны длинные и подробные сообщения. В то

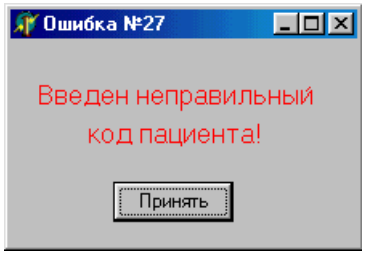
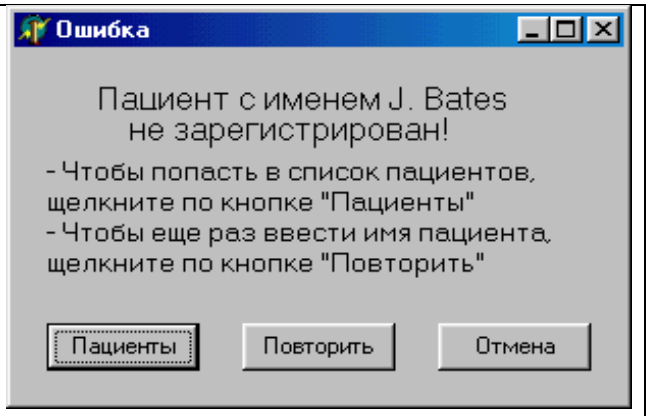
	же время начинающим пользователям такие сообщения покажутся сложными, мало понятными и слишком краткими.
Фактор	**Описание**
Стиль сообщений	Сообщения должны иметь положительный, а не отрицательный оттенок. Всегда следует использовать активный, а не пассивный тон обращения. Не должно быть оскорблений или попыток пошутить
Культура	Разработчик сообщений должен быть знаком с культурой той страны, где продается система. Сообщение, вполне уместное в культуре одной страны, может оказаться неприемлемым в другой

Первое впечатление пользователя о системе основано на **сообщениях об ошибках**, они должны:

- Быть последовательными и конструктивными
- Быть вежливыми, краткими, не содержать оскорблений.
- Не содержать звуковых сигналов, которые могут сбить с толку посетителей.

Желательно:

- Связать сообщение с контекстно-зависимой справкой.
- Включить в сообщение варианты исправления ошибки.

	
Это сообщение скорректировано плохо: – Оно обвиняет пользователя в совершении ошибки. – Не рассчитано на уровень знаний пользователя. – Не предполагаются способы исправления ошибки.	Это сообщение лучше: – Оно доброжелательно – В нем используются медицинские термины. – Предполагается простой способ исправления ошибки

В связи с тем, что **справочная система** имеет иерархическую структуру, где на верхних уровнях иерархии содержится более полная информация, а на нижних – более подробная, может возникнуть следующая ситуация: пользователь заходит в систему получив сообщение об ошибке и затем перемещается в системе по ссылкам. Через некоторое время он запутывается и ему необходимо начинать все сначала. Чтобы таких ситуаций не возникало информацию удобно отображать в нескольких окнах.

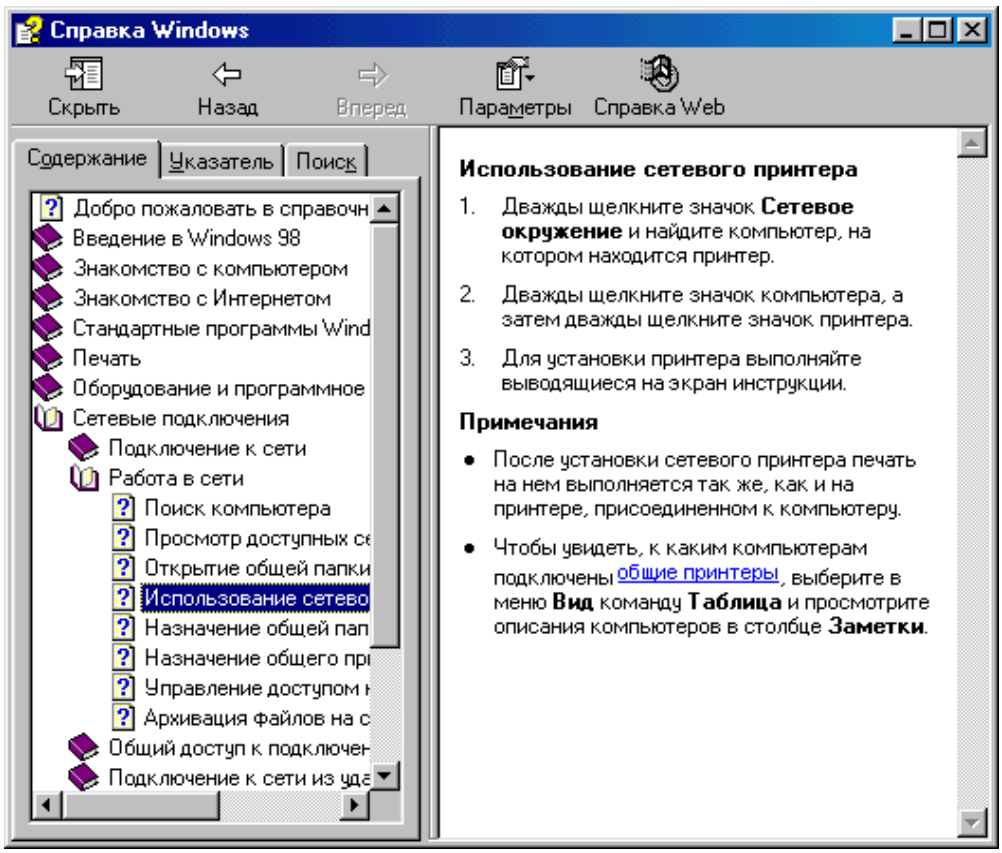


Рисунок 55 - Пример справочной системы

Тексты справочной системы должны быть:

- Написаны совместно с создателями приложения.
- Продуманы так, чтобы его можно было прочитать в окне малого размера(только необходимая информация).
- Адаптированы к неопытному пользователю.

Документация пользователя должна содержать 5 документов:

- **Функциональное описание**, в котором кратко представлены функциональные возможности системы. Прочитав функциональное описание, пользователь должен определить, та ли это система, которая ему нужна.
- **Документ по инсталляции системы**, в котором содержится информация по установке системы.

– **Вводное руководство**, представляющее неформальное введение в систему, описывающее ее "повседневное" использование.

– **Справочное руководство**, в котором описаны возможности системы и их использование, представлен список сообщений об ошибках и возможные причины их появления, рассмотрены способы восстановления системы после выявления ошибок.

– **Руководство администратора**, необходимое для некоторых типов программных систем. В нем дано описание сообщений, генерируемых системой при взаимодействии с другими системами, и описаны способы реагирования на эти сообщения.

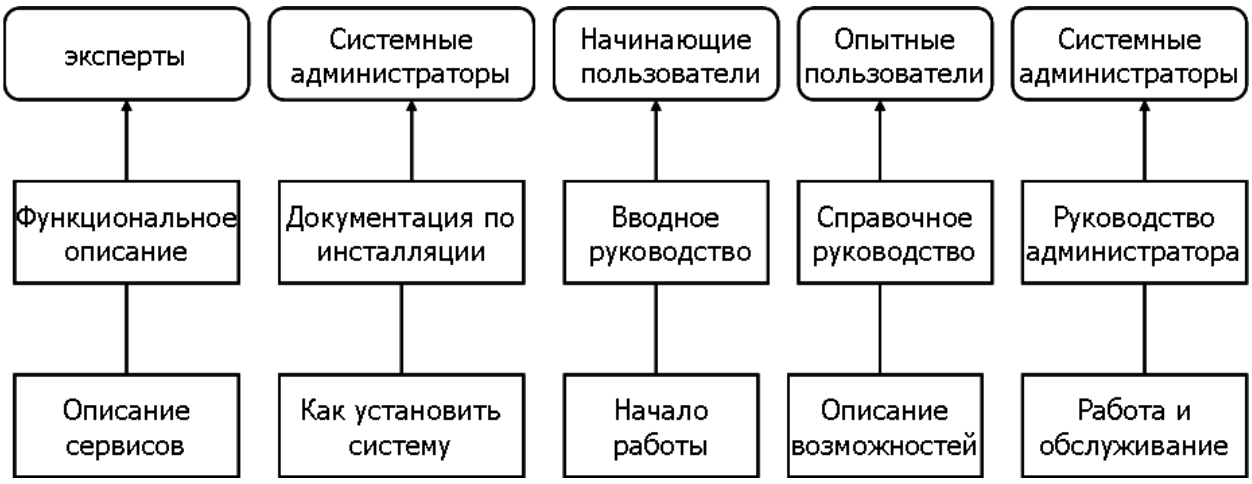


Рисунок 56 - Документация пользователя

Вместе с перечисленными руководствами необходимо предоставлять другую удобную в работе документацию. Для опытных пользователей системы удобны разного вида предметные указатели, которые помогают быстро просмотреть список возможностей системы и способы их использования.

Оценивание интерфейса

Это часть общего процесса тестирования и аттестации систем ПО, в котором оценивается удобство использования и степень соответствия интерфейса требованиям пользователя.

Таблица 10 - Показатели удобства использования.

ПОКАЗАТЕЛЬ	ОПИСАНИЕ
Изучаемость	Количество времени обучения, необходимое для начала продуктивной работы.
Скорость работы	Скорость реакции системы на действия пользователя.
Устойчивость	Устойчивость системы к ошибкам пользователя.
Восстанавливаемость	Способность системы восстанавливаться после ошибок пользователя.
Адаптируемость	Способность системы “подстраиваться” к разным стилям работы пользователя.

Существуют простые и не дорогостоящие методики оценивания, позволяющие выявить отдельные дефекты в интерфейсах.

- **Анкеты, в которых пользователи оценивают интерфейс.** Эти сведения дают возможность разработчикам зафиксировать, пользователи с каким уровнем знаний имеют проблемы с интерфейсом.
- **Наблюдения за работой пользователей.** Позволяют отслеживать, какие используются сервисы, совершаемые ошибки, как пользователи взаимодействуют с системой.
- **Видеонаблюдения типичного использования системы.** Может оказаться полезным для обнаружения проблем, но для уточнения используются другие методы оценивания.

– **Добавление в систему программного кода, который собирал бы информацию о наиболее часто используемых системных сервисах и наиболее распространенных ошибках.** Способствует изменению интерфейса так, чтобы доступ к наиболее часто использующимся операциям был минимален.

Выводы

1. Грамотно спроектированный интерфейс пользователя крайне важен для успешной работы системы. Сложный в применении интерфейс, как минимум, приводит к ошибкам пользователя. Основой принципов проектирования интерфейсов пользователя являются человеческие возможности.
2. Важным аспектом интерфейса является грамотное взаимодействие с пользователем: ввод данных и их представление.
3. Разработчики ПО должны уделять должное внимание средствам поддержки пользователя.
4. Оценивание интерфейса является частью общего процесса тестирования и аттестации систем ПО.

Вопросы для обсуждения

1. Каково место проектирования и оценивания интерфейса пользователя в жизненном цикле ПО?
2. Почему проектирование интерфейса является важным моментом при создании ПО?
3. Какими принципами должен руководствоваться разработчик ПО при разработке интерфейса пользователя.

4. Перечислите преимущества и недостатки основных стилей взаимодействия пользователя с системой.
5. В каких случаях следует представлять «голые» данные для пользователя, а в каких некоторое представление от данных?
6. Какие ошибки допускают разработчики интерфейсов при использовании цветов?
7. Существует мнение, что пользователю необязательно показывать сообщение с ошибкой, а лучше исправить её системными средствами, не напрягая лишний раз пользователя. Верно ли оно? Обосновать.
8. Что входит в документацию пользователя?
9. Обосновано ли привлечение специалистов (каких?) для оценивания интерфейса?

Глава 4. Управление проектами по созданию и внедрению ПО

«Необходимость управления программными проектами вытекает из того "прискорбного" факта, что процесс создания профессионального ПО всегда является субъектом бюджетной политики организации, где оно разрабатывается, и имеет временные ограничения. Работа руководителя программного проекта по большому счету заключается в том, чтобы гарантировать выполнение этих бюджетных и временных ограничений с учетом бизнес-целей организации относительно разрабатываемого ПО.»

«Хорошее управление не гарантирует успешного завершения проекта, но плохое управление обязательно приведет к его провалу.»

Отличия процесса разработки ПО от процессов реализации технических проектов:

- Программный продукт нематериален
- Не существует стандартных процессов разработки ПО
- Большие программные проекты - это часто "одноразовые" проекты

Процессы управления:

- Написание предложений по созданию ПО. Предложения должны содержать описание целей проектов и способов их достижения. Они также обычно включают в себя оценки финансовых и временных затрат на выполнение проекта. При необходимости здесь могут приводиться обоснования для передачи проекта на выполнение сторонней организации или команде разработчиков.
- Планирование и составление графика работ по созданию ПО. На этапе планирования проекта определяются процессы, этапы и

полученные на каждом из них результаты, которые должны привести к выполнению проекта. Реализация этого плана приведет к достижению целей проекта. Определение стоимости проекта напрямую связано с его планированием, поскольку здесь оцениваются ресурсы, требующиеся для выполнения плана.

- Оценивание стоимости проекта.
- Контроль за ходом выполнения работ. Менеджер должен постоянно отслеживать ход реализации проекта и сравнивать фактические и плановые показатели выполнения работ с их стоимостью.
- Подбор персонала. Руководители — менеджеры проектов обычно обязаны сами подбирать исполнителей для своих проектов. В идеальном случае профессиональный уровень исполнителей должен соответствовать той работе, которую они будут выполнять в ходе реализации проекта. Однако во многих случаях менеджеры должны полагаться на команду разработчиков, которая далека от идеальной.
- Написание отчетов и представлений. Менеджер проекта обычно обязан посылать отчеты о ходе его выполнения как заказчику, так и подрядным организациям. Это должны быть краткие документы, основанные на информации, извлекаемой из подробных' отчетов о проекте. В этих отчетах должна быть та информация, которая позволяет четко оценить степень готовности создаваемого программного продукта.

Планирование проекта

План, разработанный на начальном этапе проекта, рассматривается всеми его участниками как руководящий документ, выполнение которого должно привести к успешному завершению проекта. Этот первоначальный

план должен максимально подробно описывать все этапы реализации проекта.

Таблица 11 – Планирование проекта

План	**Описание**
План качества	Описывает стандарты и мероприятия по поддержке качества разрабатываемого ПО
План аттестации	Описывает способы, ресурсы и перечень работ, необходимых для аттестации программной системы
План управления конфигурацией	Описывает структуру и процессы управления конфигурацией
План сопровождения ПО	Предлагает план мероприятий, требующихся для сопровождения ПО в процессе его эксплуатации, а также расчет стоимости сопровождения и необходимые для этого ресурсы
План по управлению персоналом	Описывает мероприятия, направленные на повышение квалификации членов команды разработчиков

Алгоритм планирования проекта можно представить следующим образом:

Определение проектных ограничений

Первоначальная оценка параметров проекта

Определение этапов выполнения проекта и контрольных отметок

while *пока проект не завершится или не будет остановлен*

loop

Составление графика работ

Начало выполнения работ

Ожидание окончания очередного этапа работ

Отслеживание хода выполнения работ

Пересмотр оценок параметров проекта

Изменение графика работ

Пересмотр проектных ограничений

if** (возникла проблема) **then

Пересмотр технических или организационных параметров проекта

end if

end loop

План проекта:

1. **Введение.** Краткое описание целей проекта и проектных ограничений (бюджетных, временных и т.д.), которые важны для управления проектом.
2. **Организация выполнения проекта.** Описание способа подбора команды разработчиков и распределение обязанностей между членами команды.
3. **Анализ рисков.** Описание возможных проектных рисков, вероятности их проявления и стратегий, направленных на их уменьшение.
4. **Аппаратные и программные ресурсы,** необходимые для реализации проекта. Перечень аппаратных средств и программного обеспечения, необходимого для разработки программного продукта. Если аппаратные средства требуется закупать, приводится их стоимость совместно с графиком закупки и поставки.
5. **Разбиение работ на этапы.** Процесс реализации проекта разбивается на отдельные процессы, определяются этапы выполнения проекта, приводится описание результатов ("выходов") каждого этапа и контрольные отметки.
6. **График работ.** В этом графике отображаются зависимости между отдельными процессами (этапами) разработки ПО, оценки времени их выполнения и распределение членов команды разработчиков по отдельным этапам.

7. **Механизмы мониторинга и контроля за ходом выполнения проекта.** Описываются предоставляемые менеджером отчеты о ходе выполнения работ, сроки их предоставления, а также механизмы мониторинга всего проекта.

Контрольные отметки— вехи, отмечающие окончание определенного этапа работ.

По завершении основных больших этапов, таких как разработка спецификации, проектирование и т.п., заказчику ПО предоставляются результаты их выполнения, так называемые **контрольные проектные элементы**.

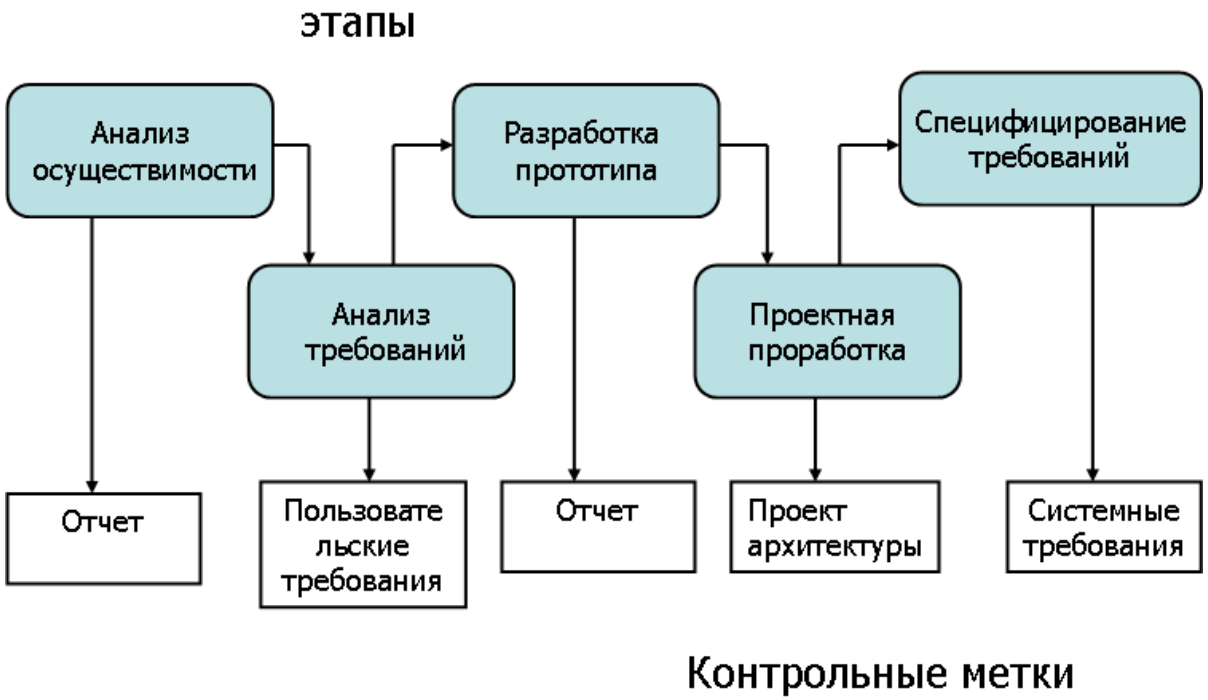


Рисунок 57 - Контрольные отметки

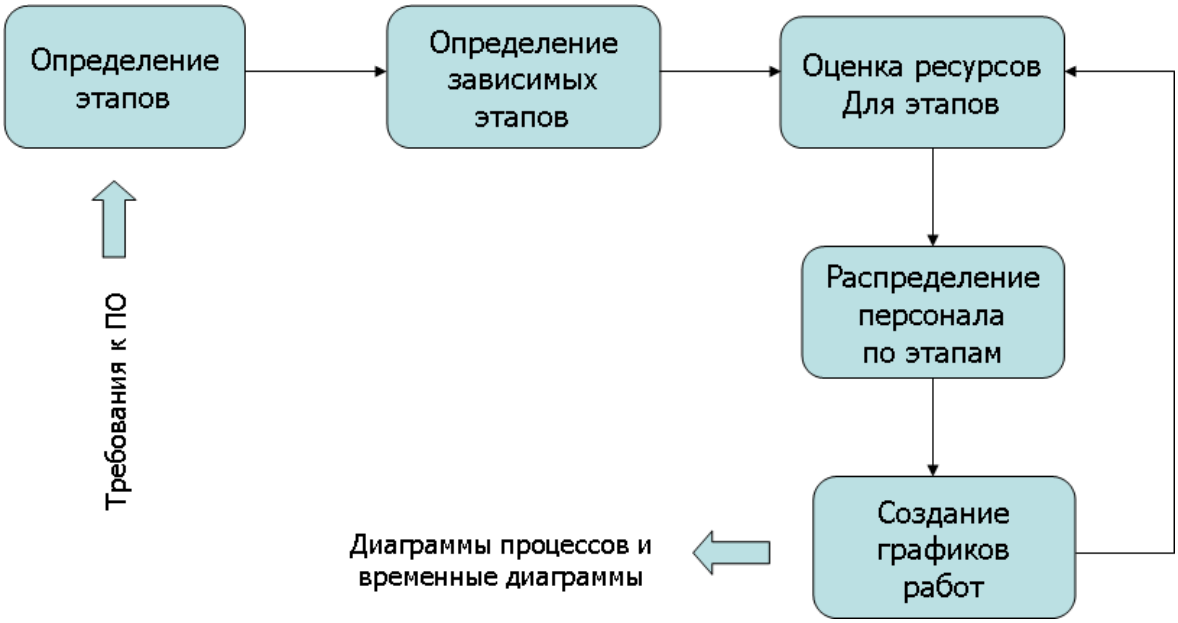


Рисунок 58 - График работ

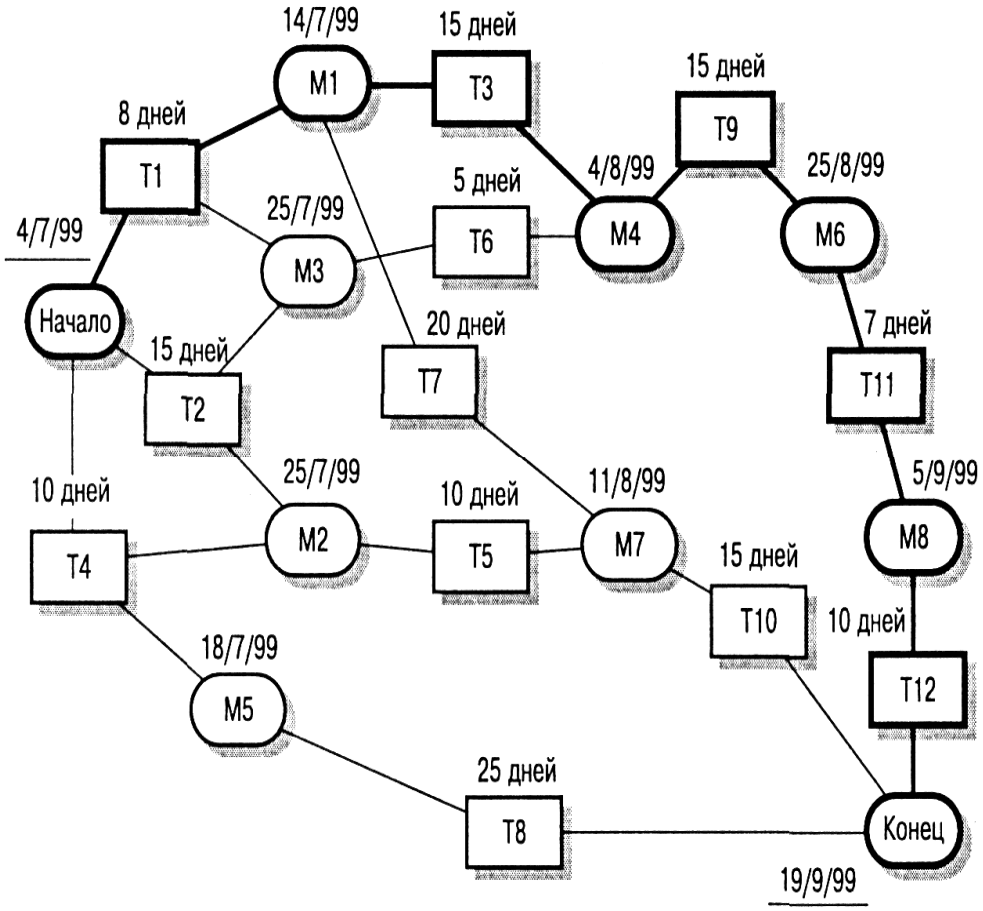


Рисунок 59 – Сетевая диаграмма

Таблица 12 – Описание сетевой диаграммы

Этап	Длительность (дни)	Зависимость
T1	**8**	
T2	**15**	
T3	**15**	**T1 (M1)**
T4	**10**	
T5	**10**	**T2, T4 (M2)**
T6	**5**	**T1, T2 (M3)**
T7	**20**	**T1 (M1)**
T8	**25**	**T4 (M5)**
T9	**15**	**T3, T6 (M4)**
T10	**15**	**T5, T7 (M7)**
T11	**7**	**T9 (M6)**
T12	**10**	**T11 (M8)**

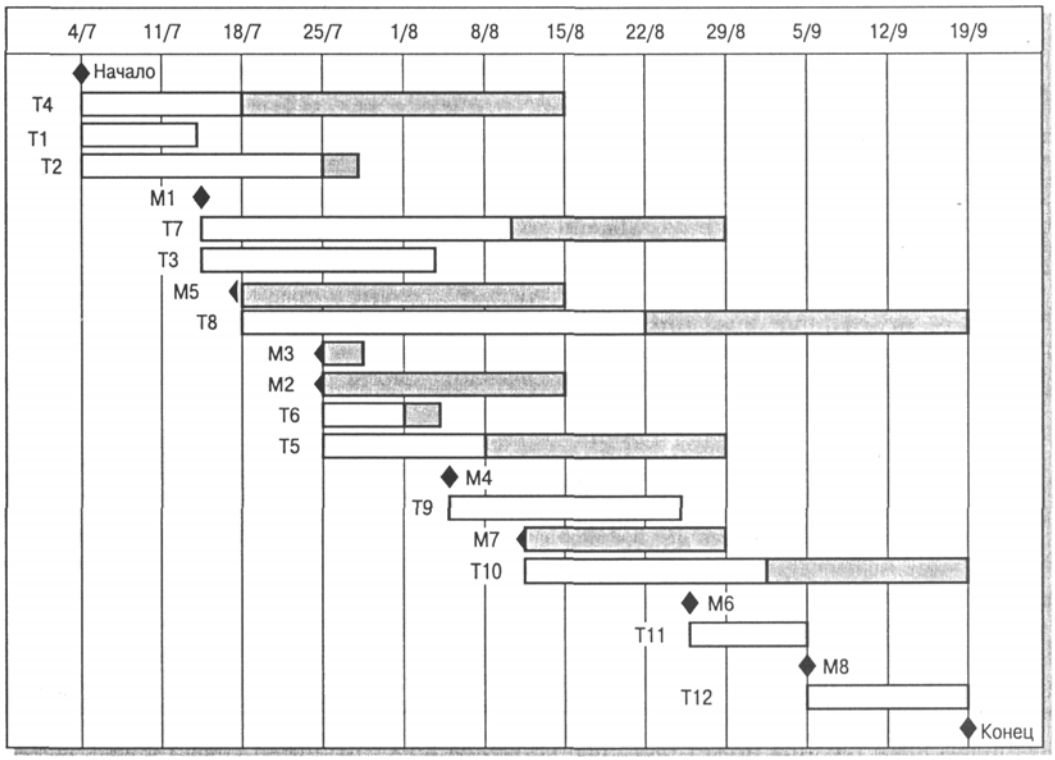


Рисунок 60 – Временная диаграмма

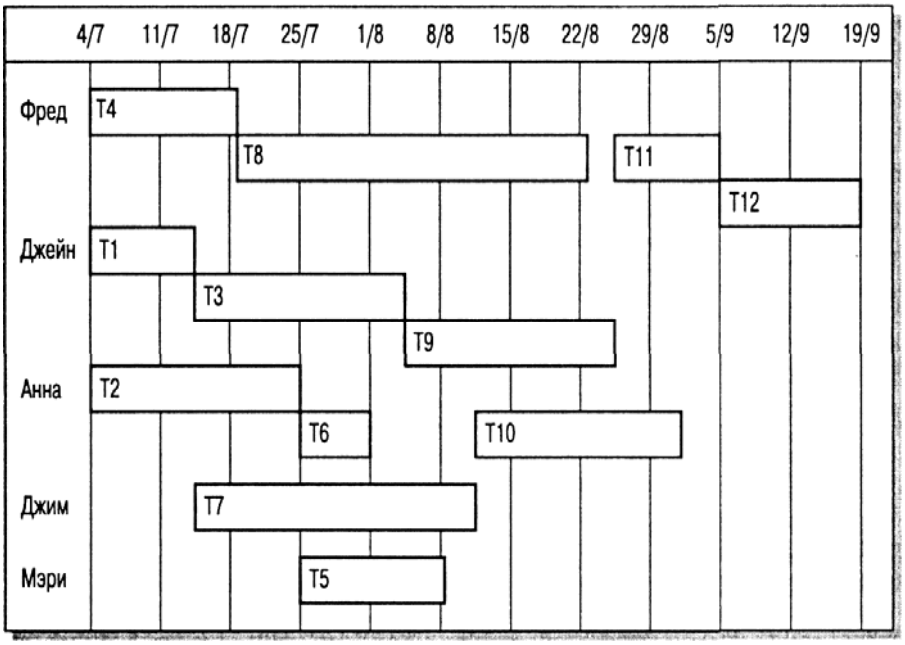


Рисунок 61 – Диаграмма распределения исполнителей по этапам

Таблица 13 – Распределение исполнителей

Этап	**Исполнитель**
T1	Джейн
T2	Анна
T3	Джейн
T4	Фред
T5	Мэри
T6	Анна
T7	Джим
T8	Фред
T9	Джейн
T10	Анна
T11	Фред
T12	Фред

Управление рисками

Риск можно понимать как вероятность проявления каких-либо неблагоприятных обстоятельств, негативно влияющих на реализацию проекта.

Возможные риски для программных проектов:

1. **Риски для проект**а, которые влияют на график работ или ресурсы, необходимые для выполнения проекта.
2. **Риски для разрабатываемого продукта**, влияющие на качество или производительность разрабатываемого программного продукта.
3. **Бизнес-риски,** относящиеся к организации-разработчику или поставщикам.

Таблица 14 - Возможные риски программных проектов

Риск	**Типы риска**	**Описание риска**
Текучесть разработчиков	Риск для проекта	Опытные разработчики покидают проект до его завершения
Изменение в управлении организацией	Риск для проекта	Организация меняет свои приоритеты в управлении проектом
Неготовность аппаратных средств	Риск для проекта	Аппаратные средства, которые необходимы для проекта, не поступили вовремя или не готовы к эксплуатации
Изменение требований	Риск для проекта и для разрабатываемого	Появление большого количества

	продукта	непредвиденных изменений в требованиях, предъявляемых к разрабатываемому ПО
Задержка в разработке спецификации	Риск для проекта и для разрабатываемого продукта	Спецификации основных интерфейсов подсистем не поступили к разработчикам в соответствии с графиком работ
Недооценка размера разрабатываемой системы	Риск для проекта и для разрабатываемого продукта	Размер системы значительно превысил первоначальную оценку
Недостаточная эффективность CASE-средств	Риск для разрабатываемого продукта	CASE-средства, предназначенные для поддержки проекта, оказались менее эффективными, чем ожидалось
Изменения в технологии разработки ПО	Бизнес-риск	Основные технологии построения программной системы заменяются новыми
Появление конкурирующего программного продукта	Бизнес-риск	На рынке программных продуктов до окончания проекта появилась конкурирующая программная система

1. **Определение рисков.** Определяются возможные риски для проекта, для разрабатываемого продукта и бизнес-риски.
2. **Анализ рисков.** Оценивается вероятность и последовательность появления рисковых ситуаций.
3. **Планирование рисков.** Планируются мероприятия по предотвращению рисков или минимизации их воздействия на проект.
4. **Мониторинг рисков.** Постоянное оценивание вероятностей рисков и выполнение мероприятий по смягчению последствий проявления рисковых ситуаций.

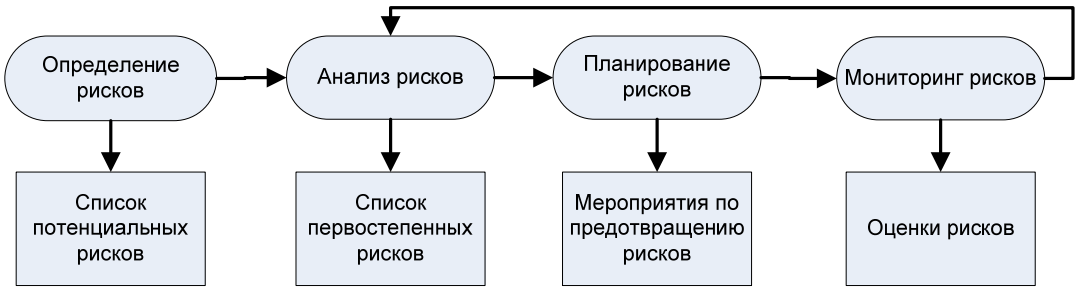


Рисунок 62 - Схема процесса управления рисками

Список возможных категорий рисков:

1. Технологические риски. Проистекают из программных и аппаратных технологий, на основе которых разрабатывается система.
2. Риски, связанные с персоналом. Связаны с членами команды разработчиков.
3. Организационные риски. Проистекают из организационного окружения, в котором выполняется проект.
4. Инструментальные риски. Связаны с используемыми CASE-средствами и другими средствами поддержки процесса создания ПО.

5. Риски, связанные с системными требованиями. Проявляются при изменении требований, предъявляемых к разрабатываемой системе.
6. Риски оценивания. Связаны с оцениванием характеристик программной системы и ресурсов, необходимых для реализации проекта.

Таблица 15 – Категория рисков

Категория рисков	**Примеры рисков**
Технологические риски	База данных, которая используется в программной системе, не обеспечивает обработку ожидаемого объема транзакций. Программные компоненты, которые используются повторно, имеют дефекты, ограничивающие их функциональные возможности
Риски, связанные с персоналом	Невозможно подобрать работников с требуемым профессиональным уровнем. Ведущий разработчик заболел в самое критическое время. Невозможно организовать необходимое обучение персонала
Организационные риски	В организации, выполняющей разработку ПО, произошла реорганизация, в результате чего изменились приоритеты в управлении проектом. Финансовые затруднения в организации привели к уменьшению бюджета проекта
Инструментальные риски	Программный код, генерируемый CASE-

	средствами, не эффективен. CASE-средства невозможно интегрировать с другими средствами поддержки проекта
Риски, связанные с системными требованиями	Изменения требований приводят к значительным повторным темными требованиями работам по проектированию системы. Первоначальная нечеткая формулировка пользовательских требований привела к значительным изменениям системных требований, проявившихся на поздних стадиях разработки проекта
Риски оценивания	Недооценки времени выполнения проекта. Скорость выявления дефектов в системе ниже ранее запланированной. Размер системы значительно превышает первоначально рассчитанный

Таблица 16 - Список рисков после проведения их анализа

Риск	Вероятность*	Степень ущерба
Финансовые затруднения в организации привели к уменьшению бюджета проекта	Низкая	Катастрофическая
Невозможно подобрать работников с требующимся профессиональным уровнем	Высокая	Катастрофическая
Ведущий разработчик заболел в самое критическое	Средняя	Серьезная

время		
Программные компоненты, используемые повторно, имеют дефекты, ограничивающие их функциональные возможности	Средняя	Серьезная
Изменения требований приводят к значительным повторным работам по проектированию системы	Средняя	Серьезная
В организации, выполняющей разработку ПО, произошла реорганизация, в результате чего изменились приоритеты в управлении проектом	Высокая	Серьезная
База данных, которая используется в программной системе, не обеспечивает обработку ожидаемого объема транзакций	Средняя	Серьезная
Недооценки времени выполнения проекта	Высокая	Серьезная
CASE-средства невозможно интегрировать с другими средствами поддержки проекта	Высокая	Терпимая
Первоначальная нечеткая	Средняя	Терпимая

формулировка пользовательских требований привела к значительным изменениям системных требований, проявившихся на поздних стадиях разработки проекта		
Невозможно организовать необходимое обучение персонала	Средняя	Терпимая
Скорость выявления дефектов в системе ниже ранее спланированной	Средняя	Терпимая
Размер системы значительно превышает первоначально рассчитанный	Высокая	Терпимая
Программный код, генерируемый CASE-средствами, неэффективен	Средняя	Незначительная

* Вероятность риска считается очень низкой, если она имеет значение менее 10%; низкой, если ее значение от 10 до 25 %; средней при значениях от 25 до 50%; высокой, если значение колеблется от 50 до 75%; очень высокой при значениях более 75%.

Планирование заключается в определении стратегии управления каждым значимым риском, отобранным для мониторинга после анализа рисков.

Таблица 17 – Стратегии управления рисками

Риск	Стратегия

Финансовые проблемы организации	Подготовить краткий документ для руководства организации, показывающий важность данного проекта для достижения финансовых целей организации
Проблемы неквалифицированного персонала	Предупредить заказчика о потенциальных трудностях и возможной задержке проекта, рассмотреть вопрос о покупке компонентов системы
Болезни персонала	Реорганизовать работу команды разработчиков таким образом, чтобы обязанности и работа членов команды перекрывали друг друга, вследствие этого разработчики будут знать и понимать задачи, выполняемые другими сотрудниками
Дефектные системные компоненты	Заменить потенциально дефектные системные компоненты покупными компонентами, гарантирующими качество работы
Изменения требований	Попытаться определить требования, наиболее вероятно подверженные изменениям; в структуре системы не отображать детальную информацию
Реорганизация компании-разработчика	Подготовить краткий документ для руководства компании, показывающий важность данного проекта для достижения финансовых целей компании
Недостаточная производительность базы данных	Рассмотреть возможность покупки более производительной базы данных
Недооценки времени	Рассмотреть вопрос о покупке системных

выполнения проекта	компонентов, исследовать возможность использования генератора программного кода

Существует три категории стратегий управления рисками:

1. Стратегии предотвращения рисков. Согласно этим стратегиям следует проводить мероприятия, снижающие вероятность проявления рисков. Примером может служить стратегия исключения потенциально дефектных компонентов.
2. Минимизационные стратегии. Направлены на уменьшение возможного ущерба от рисков. Примером служит стратегия уменьшения ущерба от болезни членов команды разработчиков.
3. Планирование "аварийных" ситуаций. Согласно этим стратегиям необходимо иметь план мероприятий, которые следует выполнить в случае проявления рисковой ситуации. В табл. 4.7 это стратегия поведения при возникновении финансовых проблем у организации разработчика.

Мониторинг рисков заключается в регулярном пересчете вероятностей рисков и ущерба, который они могут нанести.

Таблица 18 – Типы рисков

Тип риска	**Признаки**
Технологические риски	Задержки в поставке оборудования или программных средств поддержки процесса создания ПО, многочисленные документированные технологические проблемы
Риски, связанные с персоналом	Низкое моральное состояние персонала, натянутые отношения между членами команды разработчиков, низкое качество выполненной работы

Организационные риски	Разговоры среди персонала о пассивности и недостаточной компетентности высшего руководства организации
Инструментальные риски	Нежелание разработчиков использовать программные средства поддержки, неодобрительные отзывы о CASE-средствах, запросы на более мощные инструментальные средства
Риски, связанные с системными требованиями	Необходимость пересмотра многих системных требований, недовольство заказчика ПО
Риски оценивания	Изменения графика работ, многочисленные отчеты о нарушении графика работ

Вопросы для обсуждения

1. Объясните, почему нематериальность программных систем порождает особые проблемы в процессе управления программными проектами.
2. Объясните, почему хорошие программисты не всегда могут быть хорошими менеджера проектов.
3. Объясните, почему процесс планирования проекта является итерационным и почему план должен постоянно пересматриваться в течение всего срока выполнения проекта.
4. Менеджер проекта предупреждает о возможной задержке выполнения работ, которой можно избежать только за счет бесплатных сверхурочных работ команды разработчиков. Все члены команды имеют семьи, требующие определенной доли внимания. Обсудите возможность отклонения предложения менеджера о бесплатных сверхурочных работах либо согласия предпочесть

интересы организации семейным интересам. Какие аргументы наиболее весомы в этой дискуссии?

5. Как опытному программисту, вам предложили возглавить управление проектом, но вы чувствуете, что больше пользы можете принести в качестве технического специалиста, а не менеджера проекта. Обсудите возможности принятия или отклонения предложения возглавить программный проект.

Глава 5. Управление персоналом при реализации проектов

Люди, работающие в компаниях по разработке ПО, являются их самым ценным "активом". Именно они представляют интеллектуальный капитал, и от менеджеров по разработке ПО зависит, получит ли компания наилучшие из возможных дивиденды от инвестиций в человеческие ресурсы.

В успешно развивающихся компаниях и экономических структурах это достигается в том случае, если организация уважает своих сотрудников.

Круг выполняемых ими обязанностей и уровень вознаграждения должны соответствовать их умению, которое, в свою очередь, зависит от квалификации.

Организация человеческой памяти

1. Кратковременная память с быстрым доступом, но ограниченными возможностями. Доступна для обработки поступающей информации.
2. Промежуточная память с высокими возможностями. Хранение «коротко срочной» информации.
3. Долговременная память. Это память с самыми широкими возможностями, относительно трудным доступом и крайне ненадежными механизмами хранения.

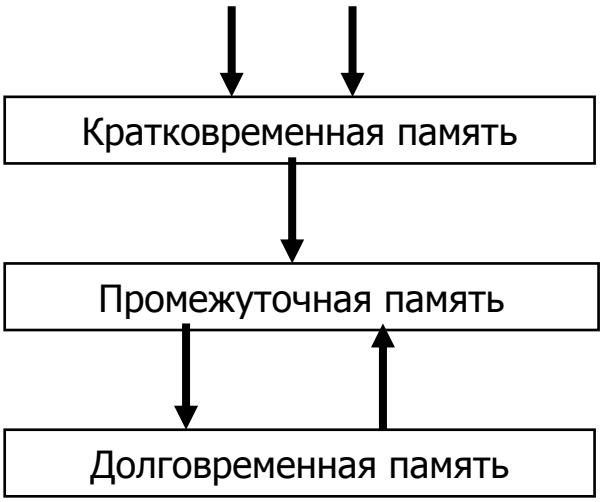


Рисунок 63 – Строение человеческой памяти

Семантические знания. Это знания об основных понятиях, таких, например, как функционирование оператора присвоения, представление о классе объектов, о технике хешированного поиска или о структуре организации программ. Эти знания приобретаются через опыт и обучение и сохраняются в форме автономных представлений.

Синтаксические знания. Это детализированные знания (подробности) об отдельных объектах и явлениях, например о том, как дать описание объекта в UML, какие стандартные функции доступны в языке программирования, создается ли оператор присваивания с помощью знака "=" или знака ":=" и т.д. Эти знания хранятся в неструктурированном виде.

Решение задач

Для того чтобы создать систему ПО, в первую очередь необходимо понять поставленную задачу (проблему), разработать стратегию поиска решения и преобразовать решение в программу.

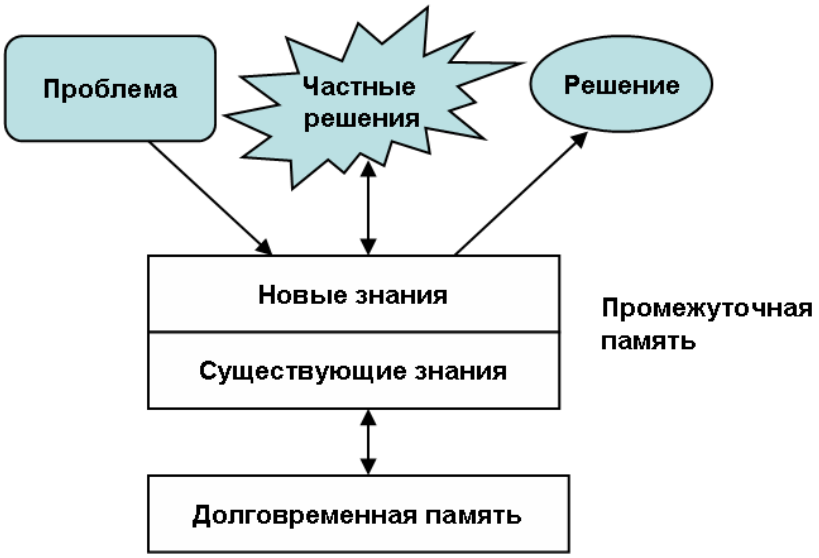


Рисунок 64 – Схема решения задачи

Первый этап включает переход постановки задачи из кратковременной памяти в промежуточную. Далее проблема сопоставляется и интегрируется с уже имеющимися знаниями в долговременной памяти, а затем обрабатывается в целях составления определенного решения. В заключение найденное решение переносится в исполняемую программу.

Если менеджерам необходимо определить, кого включить в долгосрочный проект, в первую очередь следует оценить способность специалиста решать всеобъемлющие проблемы и его опыт работы в данной области и лишь потом его мастерство программиста.

Как только приходит понимание поставленной задачи, у опытных программистов возникают приблизительно одинаковые трудности в разработке программы, независимо от того, какой при этом используется язык программирования.

Несомненно, навыки программирования необходимы, и для их развития потребуется достаточно много времени. Однако, гораздо легче освоить определенный язык программирования, чем развить в себе способности к решению задач.

Мотивация человека направлена на удовлетворение своих потребностей. Эти потребности имеют иерархическую структуру.

Люди, работающие в организациях, которые занимаются разработкой программного обеспечения, как правило, не испытывают сильного голода или жажды и чувствуют себя в относительной безопасности в своем окружении. Таким образом, в аспекте управления этими людьми главной задачей менеджмента является удовлетворение их потребностей, связанных с оценкой, самореализацией и необходимостью быть членом определенной социальной группы.

Рисунок 65 – Пирамида мотиваций

Тактика удовлетворения социальных потребностей основывается на **предоставлении людям возможности и времени для встреч с коллегами**, а также на том, чтобы обеспечить место для таких встреч. Неформальные и легкие в использовании средства общения (например, электронная почта) с этих позиций представляют исключительную ценность.

Для удовлетворения потребности в оценке крайне важно дать понять людям, насколько важна их роль в организации. **Открытое признание их достижений** — наиболее простой и эффективный способ удовлетворения

этой потребности. Кроме того, люди должны чувствовать, что их **работа оплачивается на должном уровне**, который определяется их знаниями и опытом.

Чтобы удовлетворить потребности персонала в самореализации важно **предоставить каждому сотруднику определенный уровень ответственности** за сделанную работу. Это достигается путем поручения им достаточно трудных задач (но ни в коем случае не невыполнимых), а также проведения обучения, в процессе которого могут развиваться их навыки.

Можно выделить три типа профессионалов:

1. Люди с **целевой ориентацией**, получающие достаточно мотивации от работы, которую выполняют. К этому типу относятся "технари", мотивация которых вызвана интеллектуальными задачами по разработке программного обеспечения.
2. Люди с **самоориентацией**, мотивация которых основана на личном успехе и признании. Они заинтересованы в разработке программного обеспечения, преследуя при этом личные интересы.
3. Люди с **внешней ориентацией**, мотивация которых требует присутствия и деятельности сотрудников. Так как в наше время создание программ становится все более ориентированным на пользователя, такие люди все чаще вовлекаются в разработку программного обеспечения.

Групповая работа

Организация команды, которая могла бы эффективно работать над программой, является достаточно сложной задачей для менеджера.

Необходимо, чтобы в команде было **равное соотношение технических навыков, опыта и выражения индивидуальности.**

Хорошо функционирующая команда — это нечто большее, чем простой набор людей с необходимым соотношением навыков. В хорошей команде присутствует **дух товарищества**, который мотивирует сотрудников через успехи всей команды, включая и достижение собственных целей.

Поэтому менеджеры должны стимулировать деятельность, направленную непосредственно на **"строительство команды"**, чтобы содействовать формированию чувства преданности ее интересам.

- **Состав команды.** Команда должна иметь правильное соотношение навыков, опыта и личностных качеств.
- **Сплоченность команды.** Члены рабочей группы должны воспринимать себя как единую команду, а не как простую совокупность индивидуумов, работающих над одной проблемой.
- **Общение в команде.** Между членами команды должны быть дружеские отношения.
- **Организация команды.** Необходимо организовать команду таким образом, чтобы каждый чувствовал свою ценность и был удовлетворен своей ролью.

Создание команды

Группа, в которой сотрудники дополняют друг друга, может работать намного эффективнее группы, отбор в которую проводился исключительно на основе навыков программирования.

Люди, которые любят свою работу (**целевая ориентация**), могут стать прекрасными профессионалами.

Люди с **самоориентацией** на наилучший результат смогут довести дело до конца.

Сотрудники с **внешней ориентацией** успешно налаживают общение внутри группы. Они настроены на общение и поэтому могут определить (и предотвратить) возникновение какого-либо напряжения или конфликтов на ранней стадии. Именно такие люди помогут разрешить личные проблемы членов команды и разногласия между ними, прежде чем те окажут влияние на всю команду.

Важное место в команде занимает **лидер**. Он (или она) отвечает за техническое руководство и административное управление. Лидеры группы должны быть в курсе повседневной деятельности группы, гарантируя эффективную работу команды и тесное сотрудничество с менеджерами проекта при планировании деятельности по его реализации.

Лидер — это, как правило, назначаемая должность, он подотчетен главному менеджеру проекта. Назначаемый лидер может и не быть лидером команды в прямом смысле этого слова, он ведет группу только в технических вопросах.

Члены группы могут выбрать другого лидера команды. Он может лучше назначенного лидера разбираться в технических вопросах или лучше мотивировать членов группы к выполнению работы.

Сплоченность команды

Члены сплоченной команды привержены ее интересам больше, чем своим собственным. Это укрепляет группу, она становится способной самостоятельно справляться с проблемами и непредвиденными ситуациями.

Хорошо сплоченная команда имеет ряд преимуществ:

- Возможность становления стандарта качества группы. Так как этот стандарт определяется всей группой единогласно, его легче

– Члены команды поддерживают тесные рабочие контакты. Работая в группе, люди учатся друг у друга. Скованность и затягивание работы, вызванные незнанием или неосведомленностью, уменьшаются по мере того, как происходит взаимное обучение.

– Члены команды ознакомлены с деятельностью друг друга. Этим достигается возможность продолжения работы даже после ухода одного из сотрудников.

– Возможно внедрение в практику группы безличного программирования. Созданная программа должна быть собственностью всей команды, а не отдельной личности.

– Менеджеры могут развивать сплоченность несколькими путями. Можно организовывать социальные мероприятия для работников и их семей. Можно привить группе чувство самобытности, для чего ее надо назвать, определить сущность команды и сферу ее деятельности. Менеджеры должны проводить мероприятия (например, игры и спорт), прямо направленные на создание команды.

– Однако наилучший способ воспитать дух команды - дать возможность каждому почувствовать, что он несет определенную долю ответственности и что ему доверяют, а также гарантировать доступ к проектной информации для всех членов группы.

– Иногда менеджерам кажется, что они не должны раскрывать определенную информацию. Однако такая линия поведения будет постоянно создавать в группе чувство недоверия. Простой

обмен информацией — самый дешевый и эффективный способ дать людям почувствовать себя частью команды.

Для группы по разработке программных продуктов просто необходим развитой коммуникационный фактор.

На эффективность общения могут оказать влияние следующие показатели.

1. Размер группы. Чем больше группа, тем труднее обеспечить постоянное общение между ее членами.
2. Различие в социальном положении членов группы приводит к появлению большего количества односторонних связей.
3. Структура группы. Работники, состоящие в группах с неформальной структурой, легче общаются между собой, чем в группах, которые имеют определенную официальную иерархию в отношениях.
4. Состав группы. Если в группе много людей с похожими личностными характеристиками, они могут конфликтовать друг с другом, вследствие чего может значительно снизиться уровень общения в группе. Лучше всего люди общаются в смешанных разнополых группах, чем в однородных по полу.
5. Рабочее окружение. Правильная организация рабочего места — основополагающий фактор в развитии или торможении коммуникационных связей в группе.

Организация группы

Чтобы использовать высококвалифицированный персонал с наибольшей отдачей, многие специалисты предлагают строить группу вокруг одного высококвалифицированного ведущего программиста. Основной принцип такой организации состоит в том, чтобы компетентный и опытный

сотрудник отвечал за разработку всего программного продукта. Ведущего программиста не следует загружать рутинной работой, ему наоборот нужна хорошая поддержка в решении вопросов административного и технического плана. Такого сотрудника также следует избавить от излишнего общения со специалистами вне группы.

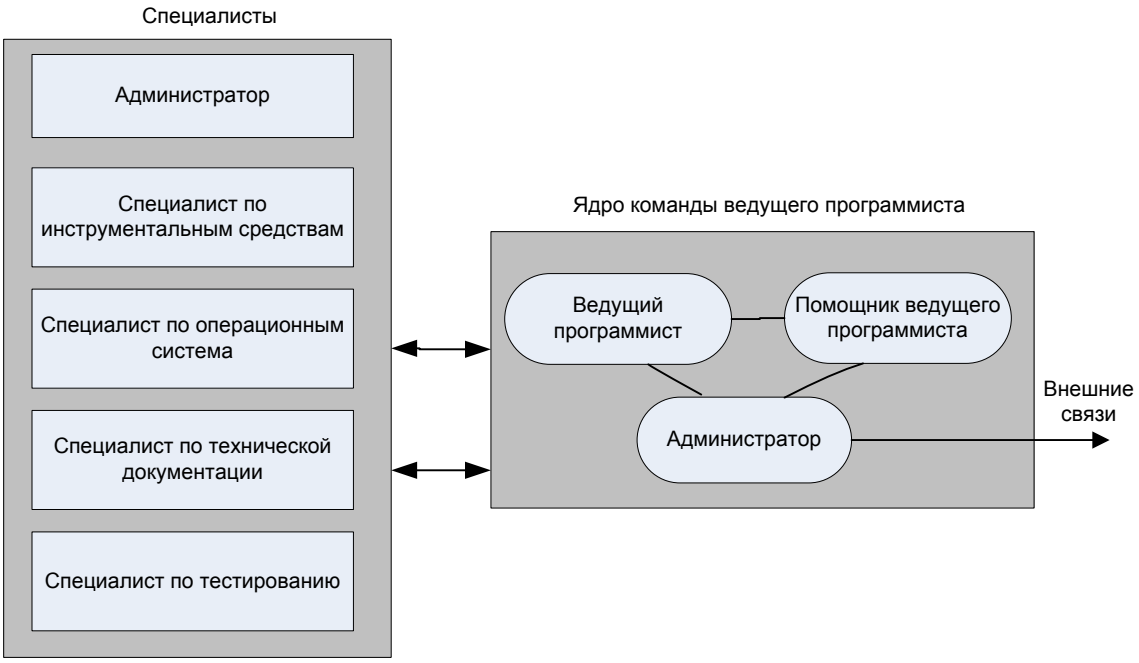


Рисунок 66 – Структура группы разработчиков

Таблица 19 – Факторы выбора сотрудников

Фактор	Пояснение
Знания об области применения ПО	Для того чтобы разработать хорошо функционирующую систему, программист должен иметь четкое представление о той прикладной области, где будет применять разрабатываемое ПО
Опыт работы на многих компьютерных платформах	Этот фактор может оказаться важным при низкоуровневом программировании, в общем случае он не является решающим.
Образование	Образование служит своеобразным показателем тех основных знаний и умений, которыми должен владеть кандидат, а также его способности к

	обучению. Этот показатель становится менее значимым пропорционально опыту, получаемому в работе над различными проектами.
Коммуникабельность Способность адаптироваться	Этот фактор достаточно важен, так как в процессе реализации проекта программистам нужно будет общаться в устной и в письменной форме с другими специалистами, менеджерами и потребителями. Этот фактор также может показать способность к обучению.
Жизненная позиция	Люди, работающие над проектом, должны любить свою работу и стремиться получать новые знания и навыки. Это очень показательный фактор, однако его трудно оценить

Решение о назначении нового сотрудника по проекту основывается на трех видах информации:

- Информация об образовании и практическом опыте, предоставляемая кандидатом на должность (резюме или автобиография).
- Информация, получаемая при интервьюировании кандидата.
- Рекомендации от других людей, имеющих опыт совместной работы с кандидатом.

Вопросы для обсуждения

1. Дайте краткое описание иерархической структуры человеческой памяти. Объясните, почему данной структурой обеспечивается

лучшее понимание объектно-ориентированных систем, чем систем, построенных на функциональной декомпозиции.

2. Какие факторы прежде всего принимаются во внимание при подборе сотрудников для работы над программным проектом?
3. Объясните, каким образом доступность информации о ходе разработки проекта и тех технических решениях, которые имеют отношение ко всем членам группы, могут усилить сплоченность группы.
4. Почему открытые и общие помещения менее пригодны для работы команды программистов, чем индивидуальные кабинеты? В каких случаях, по вашему мнению, открытые офисы оказываются более подходящими?
5. Как вы думаете, порядочно ли схитрить и дать те ответы на вопросы в психологическом тесте, которые работодатель хочет от вас услышать, а не говорить того, что вы на самом деле думаете?

Глава 6. Оценка стоимости программного продукта

Рассматривается проблема оценки затрат и времени, необходимых для выполнения определенных этапов проекта.

Менеджерам необходимо получить ответы на следующие вопросы.

- Какие затраты необходимы для выполнения этапа?
- Сколько это займет времени?
- Какова стоимость выполнения данного этапа?

Этапы расчета оценки стоимости:

- Предварительные расчеты должны быть выполнены на ранней стадии для утверждения бюджета.
- Во время выполнения проекта все расчеты должны регулярно обновляться. Это помогает планировать работу и содействует эффективному использованию средств.

Цена продукта включает:

- издержки производства;
- предлагаемую прибыль

Параметры, используемые для оценки проекта:

- Стоимость аппаратных средств и программного обеспечения, включая их обслуживание.
- Расходы на командировки и обучение.
- Расходы на персонал (в основном на привлечение со стороны специалистов по программному обеспечению), включающие:
  - расходы на содержание, отопление и освещение офисов;

- на содержание вспомогательного персонала- бухгалтеров, секретарей, уборщиц и технического персонала;
- на содержание компьютерной сети и средств связи;
- на централизованные услуги - библиотеки, места отдыха и развлечения и т.д.;
- на социальное обеспечение и выплаты служащим (например, пенсии и медицинская страховка).

Таблица 20 - Факторы, влияющие на стоимость программного продукта

Фактор	**Описание**
Возможности рынка ПО	Организация-разработчик может выставить низкие цены на программный продукт из-за намерения переместиться в другой сегмент рынка ПО, что в будущем может привести к более высоким доходам.
Непредвиденные факторы	Если организация примет фиксированную величину стоимости, издержки производства могут возрасти из-за непредвиденных расходов
Условия контракта	Если, например, право на владение программным кодом после завершения проекта передано заказчику, то проект стоит дороже.
Изменение требований	После заключения контракта за изменение требований можно назначить дополнительную цену
Финансовая стабильность	Во избежание банкротства фирмы,

	испытывающие финансовые затруднения, для получения заказа могут снизить цены на свои разработки.

Оценка производительности разработки ПО основана на измерении количественных показателей программных продуктов и последующем делении их на количество усилий, затраченных на разработку этих продуктов:

- Показатель размера. Зависит от размера выходного результата очередного этапа работ, например, количество строк программного кода.
- Функциональный показатель. Зависит от функциональных возможностей программного продукта в целом, например, количество функциональных и объектных точек.

Показатель размера

Количество строк программного кода за человеко-месяц наиболее популярный критерий оценки производительности, но он не всегда оптимален, потому что используются разные языки программирования. Он определяется путем деления общего количества строк кода на количество времени в человеко-месяцах, которое потребуется для завершения проекта.

Функциональный показатель

Для определения можно воспользоваться методом функциональных точек. Критерием оценки производительности выступает количество функциональных точек, созданных за человеко-месяц. Функциональная точка — это комбинация свойств ПО:

- Интенсивности использования ввода и вывода внешних данных.
- Взаимодействия системы с пользователем.

– Внешних интерфейсов.

– Файлов, используемых системой.

Нескорректированный подсчет функциональных точек (UFC) выполняется путем вычисления суммы произведений оценки каждого фактора (количество элементов, составляющих данный фактор) на выбранную весовую величину этого фактора:

UFC = 2 (количество элементов данного типа) x (весовая величина).

Для определения можно воспользоваться **методом объектных точек**.

Количество объектных точек в программе можно получить путем предварительного подсчета ряда элементов:

– Количество изображений на дисплее. Простые изображения принимаются за 1 объектную точку, изображения умеренной сложности принимаются за 2 точки, а очень сложные изображения принято считать за 3 точки.

– Количество представленных отчетов. Для простых отчетов назначаются 2 точки, умеренно сложным отчетам назначаются 5 точек. Написание сложных отчетов оценивается в 8 точек.

– Каждый модуль на языке третьего поколения считается за 10 объектных точек.

Производительность программиста

Самым важным фактором являются индивидуальные способности.

Таблица 21 - Факторы, влияющие на производительность программиста

Фактор	**Описание**
Опыт разработки ПО для	Для эффективной разработки программного

предметной области	продукта необходимо знание той предметной области, где будет эксплуатироваться области разрабатываемое ПО.
Процесс управления качеством	Применяемый метод программирования может оказать существенное влияние на производительность написания кода.
Размер проекта	Чем больше проект, тем больше времени уходит на согласование различных вопросов внутри группы разработчиков и ниже производительность.
Поддержка технологии разработки ПО	Хорошая поддержка технологии разработки ПО, например CASE-средства или системы управления конфигурацией, может значительно повысить производительность труда программиста
Рабочая обстановка	Спокойное рабочее окружение с индивидуальными рабочими местами способствует повышению производительности

Основная проблема в оценке себестоимости проектов заключается в низкой точности применяемых методов оценивания.

Таблица 22 – Методы оценки стоимости ПО

Метод	**Описание**
Алгоритмическое моделирование себестоимости	Метод основан на анализе статистических данных о ранее выполненных проектах, при этом определяется зависимость себестоимости проекта от какого-нибудь количественного показателя программного продукта (обычно это

	размер программного кода).
Оценка эксперта	Проводится опрос нескольких экспертов по технологии разработки ПО, знающих область применения создаваемого программного продукта.
Оценка по аналогии	Проект оценивается по уже реализованным аналогичным проектам.
Закон Паркинсона	Усилия, затраченные на работу, распределяются равномерно по выделенному на проект времени. Здесь критерием для оценки затрат по проекту являются человеческие ресурсы, а не целевая оценка самого программного продукта.
Назначение цены с целью выиграть контракт	Затраты на проект определяются наличием тех средств, которые имеются у заказчика. Поэтому себестоимость проекта зависит от бюджета заказчика, а не от функциональных характеристик создаваемого продукта.

Предварительная оценка может выполняться с применением нисходящего и восходящего подходов:

- При нисходящем подходе оценка себестоимости начинается на уровне системы: рассматриваются функциональные возможности программы в целом и то, как эти возможности реализуются посредством функций более низкого уровня.
- Восходящий подход начинается на уровне системных компонентов. Система разбивается на компоненты и определяются затраты на разработку каждого из них. Затем эти

затраты суммируются для определения полной стоимости проекта.

Недостатки восходящего подхода являются достоинствами нисходящего и наоборот. Для работы с большими проектами необходимо применить несколько методов оценивания себестоимости для их последующего сравнения.

Алгоритмическую модель стоимости можно построить с помощью анализа затрат и параметров уже разработанных проектов.

В общем случае формула для вычисления алгоритмической оценки стоимости записывается следующим образом: затраты = А х размер х М, где А — постоянный коэффициент, который зависит от организации выполнения проекта, показатель размер может соотноситься либо с размером кода программы, либо с функциональной оценкой, выраженной в количестве объектных или функциональных точек,

Модель СОСОМО

Эта модель основана на опыте реализации многих программных проектов. Она создана путем сбора данных о большом количестве проектов и анализа этой информации.

Достоинства модели:

- Эта модель имеет хорошую техническую документацию, общедоступна,существуют
  коммерческие программные средства ее поддержки.
- Модель популярна и ценится среди широкого круга пользователей.
- Она прошла достаточно долгий путь развития со времени первого появления в 1981 году, была усовершенствована для разработки ПО на языке Ada, последняя версия модели опубликована в 1995 году.

Модель COCOMO охватывает три уровня:

- Уровень предварительного прототипирования. Для определения необходимых затрат осуществляется оценка размера системы на основе объектных точек прототипа.
- Уровень предварительного проектирования. Этот уровень предусматривает окончание работы над системными требованиями и, возможно, над начальным проектом архитектуры программы. Оценка затрат на этом уровне основана на функциональных точках, которые затем пересчитываются в количество строк кода программ.
- Постархитектурный уровень. После разработки архитектуры системы существует реальная возможность достаточно точно оценить размер программы.

Алгоритмические модели стоимости применяются для сравнения различных инвестиций в целях снижения стоимости проекта.

Стоимость проекта складывается из трех компонентов:

- Стоимость целевых аппаратных средств, на которых будет функционировать разрабатываемая система.
- Стоимость платформы (вычислительная техника плюс программное обеспечение), используемой для разработки системы.
- Стоимость затрат на разработку системы.

Стоимость программного продукта (SC) вычисляется следующим образом:

SC = оценка затрат x RELY x TIME x STOR x TOOL x EXP x $15000.

Менеджеры проектов должны определить длительность выполнения проекта (т.е. составить временной график работ) и время начала найма персонала для непосредственной работы.

Распределяя прогнозируемые затраты на реализацию проекта, не можем точно знать, сколько человек необходимо включить в команду разработчиков. Часто набор программистов происходит по принципу от меньшего к большему с последующим постепенным уменьшением их численности.

Вопросы для обсуждения

1. Какое место занимает оценка стоимости ПО в жизненном цикле и какого её значение в нём? К каким последствиям могут привести ошибки на этапе оценки стоимости ПО?
2. Какой фактор, по Вашему мнению, наиболее НЕпредсказуем при оценке стоимости ПО?
3. Как Вы думаете, какие ещё действия может предпринять менеджер при превышении планируемых затрат на ПО?
4. Какие способы оценки производительности труда программиста Вы считаете наиболее адекватными?
5. Какие плюсы и минусы у метода оценки стоимости ПО - «выиграть контракт»?
6. Какие последствия возможны при сжатии графика работ до минимума? Как их избежать?

Глава 7. Управление качеством созданных программных систем

Качество программного продукта должно соответствовать некоторым техническим требованиям. Здесь возникает ряд проблем:

1. **Технические требования.** Должны одновременно удовлетворять интересам и заказчика и разработчика (н-р, удобство сопровождения).
2. **Сложность в определении и измерении показателей качества.** (н-р, переносимость,удобство сопровождения и эффективность)
3. **Сложность в создании спецификации программного продукта.** Полнота спецификации не гарантирует получение высококачественного программного продукта.

Управление качеством предполагает возможность независимого контроля за процессом разработки ПО. Сам же процесс управления качеством состоит из трех основных видов деятельности:

1. **Обеспечение качества.** Определение множества организационных процедур и стандартов.
2. **Планирование качества.** Выделение подмножества стандартов и процедур и их адаптация к данному проекту.
3. **Контроль качества.** Проведение мероприятий по выполнению нормативных процедур и стандартов качества всеми членами группы разработчиков.

Особенности процесса управления качеством:

– Контрольные проектные элементы в процессе разработки ПО являются основой контроля качества. Это дает возможность

своевременного получения информации о проблемах и трудностях.

– Команда контроля за качеством не должна быть связана с группой разработчиков.

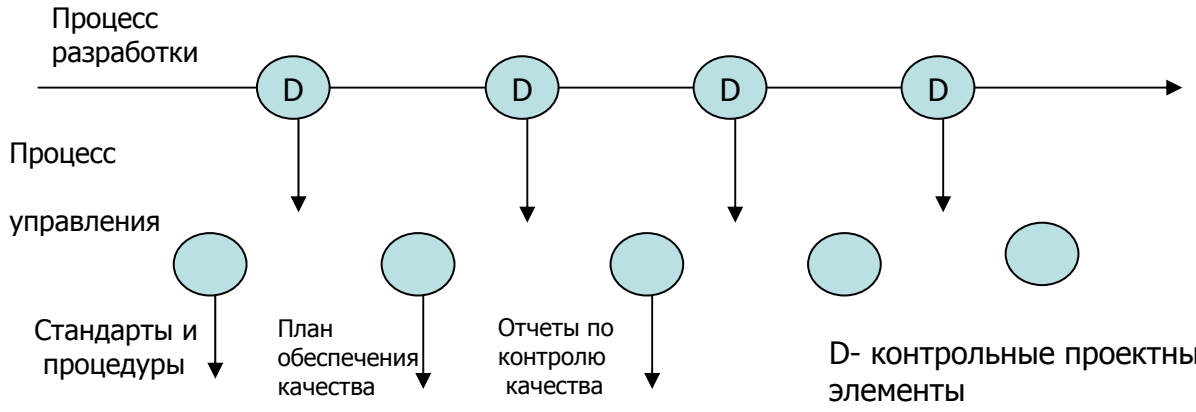


Рисунок 67 – Процесс управления качеством

ISO 9000- это целый ряд всевозможных стандартов, принимаемых за основу развития систем управления качеством.

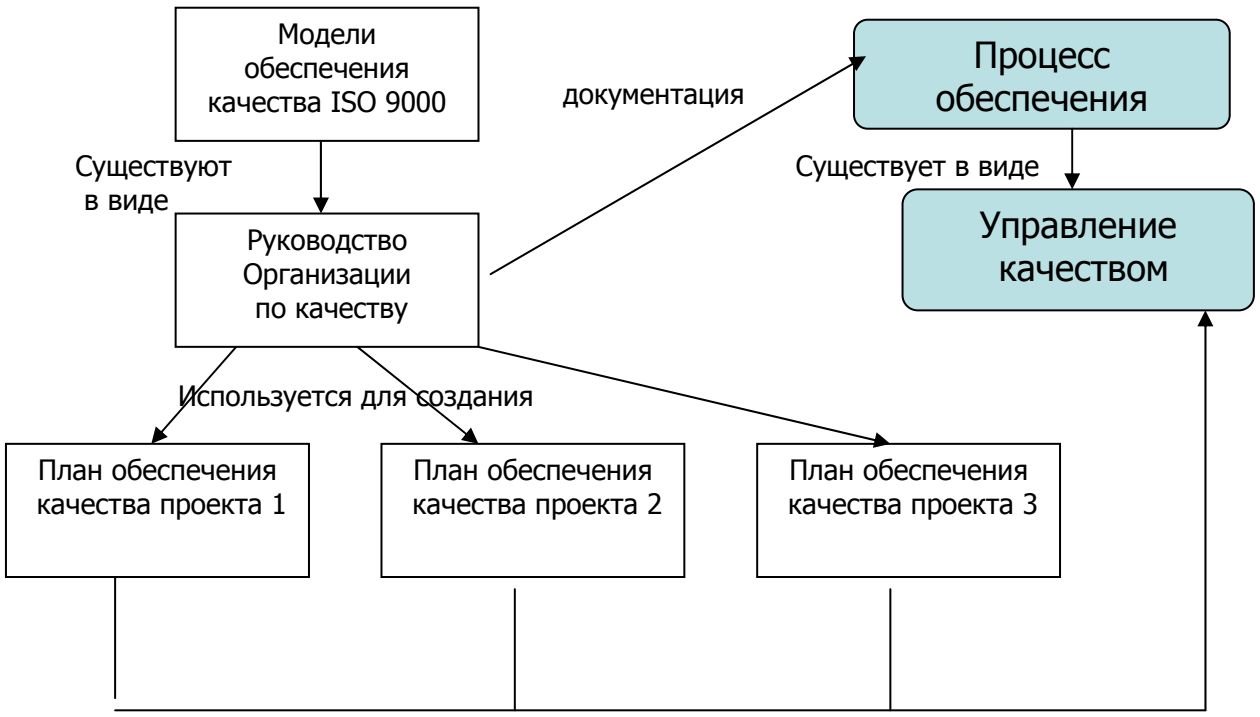


Рисунок 68 – Взаимодействие элементов качества

Таблица 23 - Стандарты на продукцию и процесс разработки ПО

Стандарты на продукцию	**Стандарты на процесс разработки ПО**
Форма пересмотра архитектуры ПО	Руководство по проведению пересмотра архитектуры ПО
Структура системных требований	Представление документации по нормативам ЕЭС
Формат заголовков программ и процедур	Процесс выпуска версии ПО
Стиль программирования языка JAVA	Процесс утверждения плана реализации проекта
Формат плана реализации проекта	Процесс контроля изменений
Форма запроса на изменение	Процесс регистрации выполнения тестов

Советы менеджеру по качеству

1. Необходимо вовлечь программистов в разработку стандартов. Описание стандартов должно содержать не только изложение норматива качества, но и объяснение необходимости выбора именно его.
2. Регулярно просматривать и обновлять стандарты, которые затем помещаются в справочник организации.
3. Подумать, как обеспечить поддержку стандартов программными средствами везде, где только можно.

Стандартные документы имеют четкую последовательную структуру, их легко читать и воспринимать.

Выделяют три основные типа стандартов на документацию:

1. **Стандарты на процесс создания документации.** Определяют способ создания технической документации
2. **Стандарты на документ.** Определяют структуру и внешний вид.
3. **Стандарты на обмен документами.** Гарантируют совместимость всех электронных версий документов.

В плане обеспечения качества отображаются стандарты наиболее подходящие к создаваемому ПО. Предлагается следующая структура плана:

1. **Представление продукта.** Описание продукта, намечаемый рынок его сбыта, а также ожидаемые свойства.
2. **Планы выпуска продукта.** Назначение крайних сроков выпуска версий программного продукта, распределение ответственности за его разработку и обслуживание.
4. **Описания процессов.** Представление процессов разработки и обслуживания программного продукта в ходе выполнения проекта и управления им.
5. **Цели качества.** Планы и цели обеспечения качества продукта, включая описание наиболее важных его характеристик.
6. **Риски и управление рисками.** Описание основных видов риска, которые могут оказать влияние на уровень качества продукта, и мероприятия, направленные на снижение рисков.

Процесс контроля качества имеет собственный набор процедур и отчетов, которые могут быть использованы в процессе разработки ПО. Они должны иметь четкую структуру.

Выделяют два взаимодополняющих подхода к процессу контроля качества:

1. Группа разработчиков анализирует документацию, сопровождающую программный продукт, проверяет соответствие документа стандартам.
2. Программный продукт и его документация проверяется специальной компьютерной программой на его соответствие стандарту.

В проверку качества включена группа специалистов, которые изучают отдельный этап или процесс разработки в целом. В таблице представлены некоторые типы проверок.

Таблица 24 – Типы проверок

Тип проверки	**Основная цель проверки**
Инспекция структуры и программного кода системы	Выявить ошибки в требованиях, в структуре и программном коде. Проверка проводится в соответствии с технологической картой возможных ошибок
Промежуточные проверки	Предоставить отчет о ходе выполнения проекта.
Проверки качества	Анализ компонентов продукта и документации для выявления несоответствия между спецификацией и структурой системы.

Измерение показателей ПО – получение числовых значений определенных показателей программного продукта или процесса его разработки.

Показатели программного обеспечения — это количественные показатели, которые можно измерить и которые характеризуют программную систему, процесс разработки ПО или сопровождающую документацию.

Показатели делятся на два вида: контрольные и прогнозируемые. Контрольные показатели обычно соотносятся с процессом разработки ПО, а прогнозируемые — с готовым программным продуктом.

Процесс измерения показателей ПО, который может быть частью контроля качества, показан на рисунке.

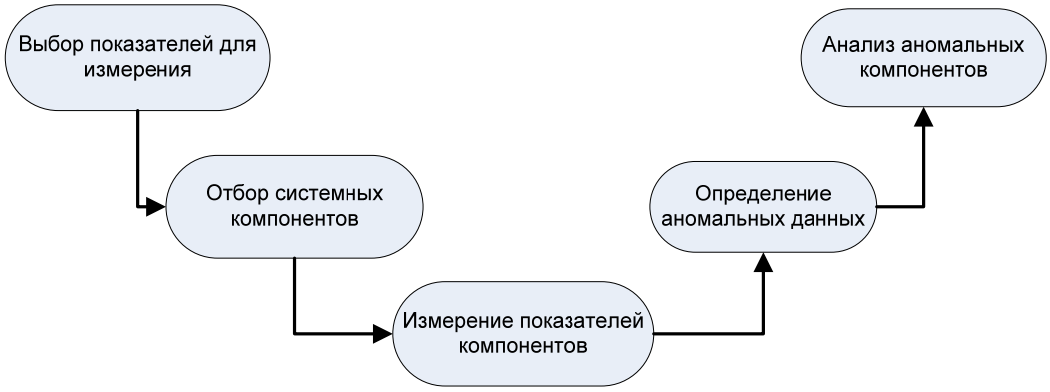


Рисунок 69 – Процесс измерений показателей качества

Процесс измерений состоит из пяти основных этапов:

1. **Выбор показателей для измерения.** Определяются измеряемые показатели.
2. **Отбор системных компонентов.** Часто совсем необязательно оценивать показатели всех компонентов программной системы.
3. **Измерение показателей компонентов.** Это процесс измерения значений выбранных показателей для отобранных компонентов.
4. **Определение аномальных данных.** Значения измеренных показателей нужно сравнить между собой и с предыдущими измерениями, занесенными в базу данных.
5. **Анализ аномальных компонентов.** Определив компоненты с аномальными показателями, их следует изучить для выявления возможного отрицательного влияния на качество программного продукта в целом

Показатели программного продукта можно разделить на два класса:

1. **Динамические показатели**, которые измеряются в процессе выполнения программы. Относительно легко измерить время выполнения определенных функций и оценить время, необходимое для запуска системы.
2. **Статические показатели**, которые отражают статические представления системы, например структуру, программный код или документацию. Статические показатели, как правило, имеют отдаленное отношение к качественным характеристикам ПО.

Вопросы для обсуждения

1. Какими качествами должна обладать система для удобства её сопровождения?
2. Как избежать ситуации, когда пользователь недоволен качеством ПО, но оно полностью соответствует спецификации?
3. Можно ли не следовать всем пунктам справочника стандартов ПО, и кто должен заниматься этим вопросом?
4. Для каких систем целесообразно применять проверку качества группой специалистов, а когда автоматизированной оценкой качества ПО?
5. Нужно ли оценивать качество прототипа системы? Если да, то какими принципами нужно руководствоваться проверяющему?
6. Как связаны статические и динамические показатели системы с оценкой качества ПО?

Литература

1. Соммервиль Иан. Инженерия программного обеспечения, 6-е издание. : Пер. с англ. – М.: Издательский дом “Вильямс”, 2002.
2. Свиридов С., Курьян А.. IDEF0: функциональное моделирование деловых процессов. // Центр ОТСМ-ТРИЗ технологий, Минск, Беларусь 1997. http://www.trizminsk.org
3. Announcing the Standard for Integration Definition For Function Modeling. // Draft Federal Information Processing Standards Publication 183, 1993.
4. Чувахин В. А. Описание отдельных концепций IDEF0. // Сайт “Корпоративный менеджмент”. http://www.cfin.ru/chuvakhin/idef0-r.shtml
5. Курьян А. Г., Серенков П.С., к.т.н., Белорусская Государственная Политехническая Академия. Использование IDEF0 для описания и классификации процессов в рамках системы качества МС ИСО семейства 9000 версии 2000. // http://www.interface.ru
6. Рубцов С.. IDEF0 и опыт разработки. Секреты моделирования и проектирования бизнес-процессов. // Открытые системы, 2002. http://big.spb.ru/
7. Верников Г.. Основные методологии обследования организаций. Стандарт IDEF0. // Управленческое консультирование. www.consulting.ru
8. Буч Г., Рамбо Дж., Джекобсон А. Язык UML. Руководство пользователя. – С-П.: Издательство «Питер», 2003.
9. Материалы сайта http://www.uml.org
10. Материалы сайта http://www.omg.org/technology/documents/formal/uml.htm
11. Материалы сайта http://www.uml.ru
12. Материалы сайта http://www.citforum.ru