

Содержание

ТЕМА 1. ОРГАНИЗАЦИЯ ПРОЦЕССА РАЗРАБОТКИ	3
1.1 Понятие способа, метода, методики и технологии	3
1.2 Программа и программная система. Программный продукт, проект. Сложность ПО. Участники разработки ПО.	3
1.3 Бизнес и ИТ-проекты. Условия успеха. Анализ успешности проектов. Факторы успеха проектов. Причины провалов. Сложности управления. Как бороться со сложностью задач и управления.	4
1.4 Классификация ПО по назначению и способу распространения. Типы программных систем.	5
1.5 Программная инженерия (ПИ) как инженерная дисциплина. Дисциплины ПИ. Цели программных инженеров.	6
1.6 Понятие жизненного цикла (ЖЦ). Основные, вспомогательные и организационные процессы.....	7
1.7 Свод знаний по программной инженерии (SWEBOK). Основные области знаний SWEBOK	8
1.8 Общие принципы разработки программного обеспечения.	8
1.9 Жизненный цикл ПО	9
1.10 Стандартные модели ЖЦ.....	14
1.11 Иерархическая структура работ проекта.....	19
1.12 Методологии и технологии проектирования ПО. Требования к технологии.	20
1.13 Технологии программирования.....	22
1.14 Обзор современных методологий разработки ПО	24
1.15 Стратегии разработки программной системы.....	27
1.16 Подходы к разработке программного обеспечения.	28
ТЕМА 2. МОДЕЛИРОВАНИЕ БИЗНЕС-ПРОЦЕССОВ.....	28
2.1 Бизнес-моделирование.....	28
2.2 Стадии бизнес-моделирования (AS-IS, TO-BE).....	30
2.3 Виды моделирования.....	30
2.4 Виды диаграмм	31
2.5 Бизнес-процесс.....	32
2.6 Управление бизнес-процессами	36
2.7 Бизнес-стратегии.....	36
2.8 Методология описания бизнес-процессов	38
2.9 Функциональное моделирование	40
2.10 Процессное моделирование.....	46
ТЕМА 3. ОСНОВЫ АНАЛИЗА ТРЕБОВАНИЙ.....	49
3.1 Роль этапа анализа требований в общем процессе разработки ПО.....	49
3.2 Классификация требований по предмету, назначению и уровню описания.....	50
3.3 Классификация требований по Вигерсу и Леффингвеллу.....	50
3.4 Функциональные требования (бизнес-требования, пользовательские и функциональные).	52
3.5 Нефункциональные требования (бизнес-правила, атрибуты качества, внешние интерфейсы, ограничения)....	53
3.6 Системные требования.	54
3.7 Свойства требований.	55
3.8 Проектная документация и стандарты, регламентирующие работу с требованиями.....	55
3.9 Работа с требованиями. Основные этапы.	56
3.10 Управление требованиями - идентификация, изменение, прослеживаемость.....	66

3.11 Средства автоматизации управления требованиями	68
3.12 Системное моделирование для разработки требований. Модели структурного и объектно-ориентированного анализа.	69
ТЕМА 4. ОСНОВЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНЫХ СИСТЕМ	69
4.1 Стратегии и методы проектирования программного обеспечения	69
4.2 Ключевые понятия архитектуры ПО.....	70
4.3 Ключевые вопросы проектирования.....	72
4.4 Горячие точки архитектуры	74
4.5 Этапы архитектурного проектирования.....	76
4.6 Архитектурные шаблоны	78
ТЕМА 5. СТРУКТУРНЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ	80
5.1 Понятие структурного анализа.....	80
5.2 Анализ требований. Модели анализа: информационная, функциональная, поведенческая. Общая модель требований к системе. Структура модели требований.	84
5.3 Проектирование	99
ТЕМА 6. ОСНОВЫ ОБЪЕКТНОЙ ТЕХНОЛОГИИ	100
6.1 Принципы объектно-ориентированного представления программных систем	100
6.2 Объекты 100	
6.3 Классы 102	
6.4 Базис языка визуального моделирования	105
ТЕМА 7. ДИАГРАММЫ UML	107
7.1 Унифицированный язык моделирования (UML). Диаграммы UML.....	107
7.2 Диаграммы прецедентов (сценариев) Use Cases. Определение требований к системе.	108
7.3 Диаграммы классов (ДК). Общие сведения. Уровни ДК. Работа с атрибутами: стереотипы, видимость, метод локализации. Операции.....	109
7.4 Диаграммы классов (ДК). Типы классов.....	110
7.5 Диаграммы классов (ДК). Связи классов. Ассоциации. Кратность ассоциации. Зависимости. Агрегации. Обобщение. Определение супер- и подтипов.....	111
7.6 Диаграммы взаимодействия: диаграммы последовательности. Работа с сообщениями. Параметры синхронизации и частоты. Работа с примечаниями.	112
7.7 Диаграммы взаимодействия: кооперативные диаграммы. Основные обозначения для диаграмм кооперации.	
112	
7.8 Диаграммы состояний. Описание динамики поведения системы. Понятие действий и деятельности.	113
7.9 Диаграммы деятельности. Рекомендации по использованию.	113
7.10 Диаграммы компонентов.	114
7.11 Диаграммы размещения.....	114
7.12 Пакеты в языке UML. Отображение архитектуры с использованием пакетов UML. Механизм пакетов. Уровни и разделы многоуровневой архитектуры.....	115
ТЕМА 8. ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ	115
8.1 Объектно-ориентированный анализ.....	115
8.2 Объектно-ориентированной проектирование. Основные принципы проектирования.....	118
8.3 Микросервисная архитектура.....	127
8.4 Принципы объектно-ориентированного проектирования	130

ТЕМА 1. ОРГАНИЗАЦИЯ ПРОЦЕССА РАЗРАБОТКИ

1.1 Понятие способа, метода, методики и технологии

Метод – логическая последовательность действий, приводящая к достижению поставленной цели.

Способ – средства + методы + формы деятельности.

Методика – совокупность способов.

Технология – методика, дающая гарантированный результат.

Технология – совокупность производственных процессов в определенной отрасли производства, а также научное описание способов производства.

1. Методология – совокупность методов, применяемых в отдельных науках;
2. Методология – учение о структуре, логической организации, методах и средствах деятельности.
Т.е. методология - наука, обобщающая методики в рамках единой и общей цели.

Технологии

- технологии программирования
- технологии проектирования
- технологии управления проектом
- технологии анализа
- технологии разработки
- технологии сопровождения

1.2 Программа и программная система. Программный продукт, проект. Сложность ПО. Участники разработки ПО.

Программа – данные, предназначенные для управления конкретными компонентами системы обработки информации в целях реализации определенного алгоритма.

Программное обеспечение (Software) – совокупность программ системы обработки информации и программных документов, необходимых для эксплуатации этих программ (ГОСТ 19781-90).

Программное обеспечение ≠ Программа

Программные системы (ПО) состоят из:

1. совокупности программ,
2. файлов конфигурации, необходимых для установки этих программ,
3. документации, которая описывает структуру системы, а также содержит инструкции для пользователей, объясняющие работу с системой,
4. адреса Web-узла, где пользователь может найти самую последнюю информацию о данном программном продукте.

Создается ПО промышленным способом коллективом профессионалов-инженеров.

Продается пользователям в виде **программных продуктов**.

Разрабатываются программные продукты в виде **программных проектов**.

Над созданием программного продукта работают представители разных специальностей:

- Аналитики;
- Архитекторы, Проектировщики;
- Управленцы (менеджеры);
- Тестеры;
- Кодировщики (программисты);
- Технические писатели;
- Системные администраторы;
- Специалисты по повторному использованию;
- Дизайнеры;
- Специалисты по эргономике;

Разработка программного обеспечения (software engineering, software development) — это род деятельности (профессия) и процесс, направленный на создание и поддержание работоспособности, качества и надежности программного обеспечения, используя технологии, методологию и практики из информатики, управления проектами, математики, инженерии и других областей знания.

Сложность разработки ПО. Как и другие традиционные инженерные дисциплины, разработка программного обеспечения имеет дело с проблемами качества, стоимости и надёжности. Некоторые программы содержат миллионы строк исходного кода, которые должны правильно исполняться в изменяющихся условиях.

Сложность современного ПО заключается в связях и коммуникационных путях между компонентами. Связи между компонентами создают зависимости между распределенными компонентами, которые могут быть сложны для понимания и управления. Трудность усугубляется тем, что компоненты часто разрабатываются и управляются людьми и коллективами, даже не известными друг другу.

1.3 Бизнес и ИТ-проекты. Условия успеха. Анализ успешности проектов. Факторы успеха проектов.

Причины провалов. Сложности управления. Как бороться со сложностью задач и управления.

Сейчас программирование – это не только наука и искусство, но и бизнес. Для того, чтобы этот бизнес был успешным, необходимо выполнение многих условий.

Необходимые условия успеха ИТ-компании:

- выпуск продукта на рынок
 - надлежащего качества;
 - вовремя;
 - интересного потенциальным пользователям.
- расходы должны соответствовать изначальному бюджету.

Проваленные ИТ-проекты: закончились неудачей – цель вообще не была достигнута.

Испытавшие большие проблемы ИТ-проекты: закончились созданием продукта, но превысили бюджет или (и) не уложились вовремя или (и) имеют лишь частичную функциональность.

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

Факторы успеха проектов.

- Вовлечение пользователей – 15,9%
Поддержка руководства – 13,9%
- Четкая и ясная постановка требований – 13,0%
Хорошее планирование – 9,6%
- Реалистичные ожидания – 8,2%
Частые контрольные точки – 7,7%
Компетентная команда – 7,2%
Владение (требованиями) – 5,3%

Причины провалов:

1. Нереалистичные временные рамки.
2. Недостаток количества исполнителей.
3. Размытые границы проекта.
4. Недостаток средств.
5. Нехватка квалифицированных кадров.

Проектами в области разработки ПО очень сложно эффективно управлять. **Сложности управления:**

- Человеческий фактор. Традиционные трудности управления коллективом.
- Новая молодая область. Не хватает квалифицированных кадров, методов, технологий, опыта.
- Сложность задач. Рост возможностей приводит к росту сложности решаемых задач и наоборот.
- Трудности в работе с заказчиками и будущими пользователями. Заказчики не знают, что они хотят.

Пользователи плохо владеют ПК и не понимают, как он работает и что от него ждать. Все говорят на разных языках.

Как бороться со сложностью задач и управления:

Разрабатывать и внедрять специальные технологии и методологии:

1. Технологии программирования (изучали в курсах Основы программирования, ООП).
2. Методологии MSF, RUP, XP, ...

Только треть программных проектов завершается успешно. Главной причиной большинства неудач в разработке ПО является не столько сложность и масштабность разрабатываемых продуктов, сколько применение неадекватных методов управления программными проектами.

Чтобы программный проект стал успешным, необходимо:

1. Четко ставить цели.
2. Определять способ достижения целей.
3. Контролировать и управлять реализацией.

4. Анализировать угрозы и противодействовать им.
5. Создавать команду.

1.4 Классификация ПО по назначению и способу распространения. Типы программных систем.

Классификация ПО. Программное обеспечение принято подразделять:

- по назначению на:
 - системное
 - прикладное
 - инструментальное
 - служебное.
- по способу распространения и использования на:
 - несвободное/закрытое
 - открытое
 - свободное.

Свободное программное обеспечение может распространяться, устанавливаться и использоваться на любых компьютерах дома, в офисах, школах, вузах, а также коммерческих и государственных учреждениях без ограничений.

Системное ПО – это часть системы, которая помогает следить за аппаратной стороной ПК и управлять ею. Сюда входят программы, контролирующие работу оперативной памяти, центрального процессора, видеокарты, устройств ввода и вывода информации, сетевые подпрограммы. Таким ПО считается:

- Драйверы – утилиты небольшого размера, функционирование которых заключается в обеспечении корректной работы остальных элементов оборудования;
- Операционные системы;
- Дополнения – языковые пакеты или настройки расширения экрана.

Основное отличие системной разновидности считается то, что она не рассчитана на выполнение конкретной поставленной задачи. Она необходима, чтобы обеспечивать бесперебойную работу остальных частей компьютера.

Прикладное ПО (прикладная программа / программное приложение / приложение) представляет собой компьютерную программу, предназначенную для выполнения конкретной задачи, отличной от той, которая относится к работе самого компьютера, обычно используется конечными пользователями. Собирательное существительное «прикладное программное обеспечение» относится ко всем приложениям в совокупности. К прикладному программному обеспечению относят:

- Архиваторы, антивирусы, браузеры, САПР, плееры
- Проигрыватели, веб-обозреватели, видеоигры, редакторы графики
- Средства для резервирования данных, работы с реестром, текстовые процессоры
- Менеджеры автозапуска, проигрыватели, менеджеры загрузок, текстовые редакторы

Инструментальное ПО – специфическое обеспечение любой компьютерной техники. Основная функция – отладка, настройка, переписывание программного кода. Сюда входят компиляторы, отладчики, переводчики высокого уровня, редакторы, интерпретаторы и другие средства. Для того, чтобы объяснить технике, что надо сделать, требуется специальный «машинный язык». Постоянно пользоваться этим кодом базовым пользователям довольно сложно, поэтому были разработаны системы, которые позволяют переводить обычную речь в двоичную, привычную для ПК.

Служебное ПО (утилиты) – это программное обеспечение, предназначенное для анализа, настройки, оптимизации или обслуживания компьютера. Используется для поддержки компьютерной инфраструктуры. Однако утилиты часто являются частью прикладных систем. Например, пакетное задание может выполнять написанный пользователем код для обновления базы данных, а затем может включать шаг, который запускает служебную программу для резервного копирования базы данных, или задание может запускать служебную программу для сжатия диска перед копированием файлов.

Типы программных систем.

1. Системы, работающие только с людьми – например: информационные системы организаций – используются людьми для ввода, хранения поиска и отображения информации;
2. Системы, работающие с людьми и внешними техническими устройствами: – например: информационные системы соревнований.
3. Системы, работающие с внешними техническими устройствами – встроенные системы реального времени – встраиваются в технические устройства для выполнения управления ими.

1.5 Программная инженерия (ПИ) как инженерная дисциплина. Дисциплины ПИ. Цели программных инженеров.

Область действия программной инженерии:

- Поиск финансирования.
- Работа с заказчиком.
- Подбор персонала.
- Этические вопросы. Микроклимат в коллективе. Команда.
- Обеспечение качества программного продукта.

Программная инженерия как инженерная дисциплина.

Программная инженерия (Software engineering) определяется как системный подход к анализу, проектированию, оценке, реализации, тестированию, обслуживанию и модернизации программного обеспечения, то есть применение инженерного подхода к разработке программного обеспечения.

Термин "разработка программного обеспечения" является более общим и часто используемым, по сравнению с термином "программная инженерия" и не обязательно включает в себя парадигмы инженерии.

Дисциплины ПИ.

Для решения важных проблем индустрии, отображения состояния научно-технических достижений в ПИ и обучение студентов в состав ПИ предложены новые дисциплины программной инженерии (ПИ).

Данные дисциплины отражают процесс управления проектом с привлечением современных научных основ (управление, инженерия, экономика и др.) сопутствующих фундаментальных наук, которые используются при разработке линий (Product Lines) для конструирования качественных программных продуктов.

Классификация дисциплин ПИ

- Наука
 - Теория программирования
 - Теория сборки
 - CASE-средства
 - Инструменты автоматизации
- Экономика
 - Методы экспертиз
 - Методы измерения и оценки процессов и продуктов на линии
 - Методы оценки качества, затрат, стоимости
- Инженерия
 - Стандарты
 - SWEBOK
 - PMBOK
 - Процессы ЖЦ
 - Линии
- Производство
 - Линии
 - Запасы программ
 - Интерфейсные детали
 - Инструменты
 - Оборудование
 - Сборочный конвейер
- Управление
 - Методы организации коллективного производства ПП
 - Методы Pert, CRM, Gantt...

Научная дисциплина (теория программирования) включает классические науки (теория алгоритмов, множеств, доказательства, математическая логика и т.п.) и соответствующие общие языковые средства проектирования с привлечением абстрактных моделей и архитектур целевых программных объектов, моделей проектирования, а также теории сборки и взаимодействия компонентов между собой и средами.

Инженерная дисциплина — совокупность технологических средств и методов проектирования ПП на основе фундаментальных и стандартных моделей ЖЦ, техники анализа домена, инженерии требований, разработки, сопровождения, изменения ПС и адаптации ПП к другим платформам и средам.

Управленческая дисциплина — общая теория управления, адаптированная к коллективной разработке ПП, включающая в себя графики работ, наблюдения за их выполнением, управления рисками, версиями (конфигурационный файл) ПП и сопровождением.

Экономическая дисциплина — совокупность методов экспертного, количественного и качественного оценивания промежуточных артефактов, конечного результата выполнения процессов ЖЦ и методов расчета времени, объема, трудоемкости и стоимости изготовления ПП.

Производственная дисциплина — сборочный конвейер с подготовленными линиями производства определенных видов ПС и СПС, по которым изготавливаются члены семейства с помощью готовых программных ресурсов (КПИ, сервисы, аспекты, агенты и т.п.), расположенных в библиотеках или репозиториях путем конфигурирования их в выходную структуру, а также тестирования и оценивания качества полученного продукта.

Цели программных инженеров:

- Создать качественный продукт.

- Удобство в сопровождении. ПО должно допускать развитие в связи с изменением потребностей пользователей.
- Надежность. Возможные неполадки в работе не должны нанести существенный, тем более невосполнимый ущерб.
- Эффективность. ПО должно эффективно использовать имеющиеся ресурсы.
- Удобство в использовании. ПО должно приниматься пользователями «на ура», работа должна быть удобной и естественной.

- Уложиться в бюджет.

- Типовое распределение средств:
60% – разработка. 40% – тестирование. Развитие – зачастую больше, чем создание.
- Детали зависят от специфики предметной области, требований к ПО, используемых подходов к организации разработки.

- Уложиться в сроки.

- Необходимо грамотное планирование.
- Анализ возможных рисков и способы реагирования.
- Борьба за четкие границы проекта.
- Мотивирование сотрудников.

1.6 Понятие жизненного цикла (ЖЦ). Основные, вспомогательные и организационные процессы.

В основе деятельности по созданию и использованию программного обеспечения лежит понятие жизненного цикла (ЖЦ). **Жизненный цикл** – период времени, который начинается с момента принятия решения о необходимости создания программного обеспечения и заканчивается в момент его полного изъятия из эксплуатации. В соответствии со стандартом все процессы ЖЦ разделены на три группы:

- Основные процессы
- Вспомогательные процессы
- Организационные процессы

Основные процессы

- приобретение (заказ)
- поставка
- разработка
- эксплуатация
- сопровождение

Вспомогательные процессы

- документирование
- управление конфигурацией
- обеспечение качества
- верификация
- аттестация
- совместная оценка
- аудит
- решение проблем

Организационные процессы

- управление проектом
- создание инфраструктуры проекта
- усовершенствование: определение, оценка и улучшение самого ЖЦ
- обучение

1.7 Свод знаний по программной инженерии (SWEBOK). Основные области знаний SWEBOK

SWEBOK (Software Engineering Body Knowledge) – свод знаний по программной инженерии.

Знания разработчиков ПО отличаются большим разнообразием, являются не согласованными и разнородными, ориентированными на разные предметные области, поэтому мировая компьютерная общественность пришла к необходимости систематизировать знания в области программной инженерии, создав ядро знаний SWEBOK (Software Engineering Body Knowledge).

Согласно SWEBOK 2004, программная инженерия включает в себя 10 основных и 7 дополнительных областей знаний, на которых базируются процессы разработки ПО.

Основные области знаний:

1. Software requirements – программные требования.
2. Software design – дизайн (архитектура).
3. Software construction – конструирование программного обеспечения.
4. Software testing – тестирование.
5. Software maintenance – эксплуатация (поддержка) программного обеспечения.
6. Software configuration management – конфигурационное управление.
7. Software engineering management – управление в программной инженерии.
8. Software engineering process – процессы программной инженерии.
9. Software engineering tools and methods – инструменты и методы.
10. Software quality – качество программного обеспечения

Основные области знаний SWEBOK:

• Требования

- Основы требований
- Процесс работы с требованиями
- Извлечение требований
- Анализ требований
- Спецификация требований
- Утверждение требований
- Практические соображения

• Проектирование

- Основы проектирования
- Ключевые вопросы проектирования
- Структура и архитектура
- Анализ качества и оценка дизайна
- Нотации дизайна
- Стратегии и методы проектирования

• Конструирование

- Основы конструирования
- Управление конструированием
- Практические соображения

• Тестирование

- Основы тестирования
- Уровни тестирования
- Техники тестирования
- Метрики, связанные с тестированием
- Процесс тестирования

• Поддержка и эксплуатация

- Основы поддержки и эксплуатации
- Ключевые вопросы поддержки и эксплуатации
- Процесс
- Техники

1.8 Общие принципы разработки программного обеспечения.

Программное обеспечение различается по назначению, выполняемым функциям, формам реализации. Однако существуют некоторые **общие принципы**, которые следует использовать при разработке ПО.

• **Частотный принцип.** Основан на выделении в алгоритмах и в обрабатываемых структурах групп действий и данных по частоте использования. Для действий, которые чаще встречаются при работе ПО, обеспечиваются условия их наиболее быстрого выполнения. К которым, к которым происходит частое обращение, обеспечивается наиболее быстрый доступ. «Частые» операции стараются делать более короткими.

• **Принцип модульности.** Под модулем в общем случае понимают функциональный элемент рассматриваемой системы, имеющий оформление, законченное и выполненное в пределах требований системы, и

средства сопряжения с подобными элементами или элементами более высокого уровня данной или другой системы. Способы обособления составных частей ПО в отдельные модули могут быть существенно различными. Чаще всего разделение происходит по функциональному признаку. В значительной степени разделение системы на модули определяется используемым методом проектирования ПО.

• Принцип функциональной избирательности. Этот принцип является логическим продолжением частотного и модульного принципов и используется при проектировании ПО, объем которого существенно превосходит имеющийся объем оперативной памяти. В ПО выделяется некоторая часть важных модулей, которые постоянно должны быть в состоянии готовности для эффективной организации вычислительного процесса. Эту часть в ПО называют ядром или монитором. В состав монитора, помимо чисто управляющих модулей, должны войти наиболее часто используемые модули. Программы, входящие в состав монитора, постоянно хранятся в оперативной памяти. Остальные части ПО размещаются на внешних запоминающих устройствах и загружаются в оперативную память только по вызову, перекрывая друг друга при необходимости.

• Принцип генерируемости. Основное положение этого принципа определяет такой способ исходного представления ПО, который бы позволял осуществлять настройку на конкретную конфигурацию технических средств, круг решаемых проблем, условия работы пользователя.

• Принцип функциональной избыточности. Этот принцип учитывает возможность выполнения одной и той же работы (функции) различными средствами. Особенno важен учет этого принципа при разработке пользовательского интерфейса для выдачи данных из-за психологических различий в восприятии информации.

• Принцип умолчания. Применяется для облегчения организации связей с системой как на стадии генерации, так и при работе с уже готовым ПО. Принцип основан на хранении в системе некоторых базовых описаний структур, модулей, конфигураций оборудования и данных, заранее определяющих условия работы с ПО. Эту информацию ПО использует в качестве заданной, если пользователь забудет или сознательно не конкретизирует ее.

При создании и развитии ПО рекомендуется применять следующие общесистемные принципы:

• принцип включения, который предусматривает, что требования к созданию, функционированию и развитию ПО определяются со стороны более сложной, включающей его в свой состав системы;

• принцип системного единства, который состоит в том, что на всех стадиях создания, функционирования и развития ПО его целостность будет обеспечиваться связями между подсистемами, а также функционированием подсистемы управления;

• принцип развития, который предусматривает в ПО возможность его наращивания и совершенствования компонентов и связей между ними;

• принцип комплексности, который заключается в том, что ПО обеспечивает связность обработки информации как отдельных элементов, так и для всего объема данных в целом на всех стадиях обработки;

• принцип информационного единства, т. е. во всех подсистемах, средствах обеспечения и компонентах ПО используются единые термины, символы, условные обозначения и способы представления;

• принцип совместимости, который состоит в том, что язык, символы, коды и средства обеспечения ПО согласованы, обеспечивают совместное функционирование всех его подсистем и сохраняют открытой структуру системы в целом;

• принцип инвариантности, который предопределяет, что подсистемы и компоненты ПО инвариантны к обрабатываемой информации, т.е. являются универсальными или типовыми.

1.9 Жизненный цикл ПО

- Фазы жизненного цикла.
- Процессы жизненного цикла: основные, вспомогательные, организационные. Процессы, действия и задачи.
- Стадии и этапы жизненного цикла. Технологические процессы и фазы в методологии RUP
- Оценка трудозатрат по этапам ЖЦ при каноническом и автоматизированном проектировании.
- Модели этапов разработки: бизнес-процессов, требований, анализа, проектирования, реализации.
- Управление жизненным циклом приложений.
- Стадии разработки. Стадии разработки по ГОСТ 19.102-77 и ГОСТ 34.601-90.

В основе деятельности по созданию и использованию программного обеспечения лежит понятие жизненного цикла (ЖЦ). **Жизненный цикл** – период времени, который начинается с момента принятия решения о необходимости создания программного обеспечения и заканчивается в момент его полного изъятия из эксплуатации.

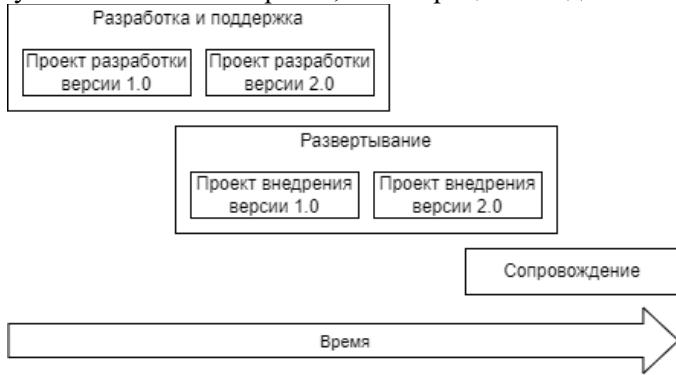
Согласно стандарту жизненный цикл программы, программной системы, программного продукта включает в себя **фазы жизненного цикла**:

- Разработка

• Поддержка. Поддержка заключается в разработке новой функциональности, переработке уже существующей функциональности, в связи с изменением требований, и улучшением продукта, а также устранение некритических замечаний к ПО, выявленных при его эксплуатации.

• **Развертывание.** Если программный продукт не коробочный, а достаточно сложный, то его развертывание у клиентов, как правило, реализуется отдельными самостоятельными проектами внедрения.

• **Сопровождение.** Сопровождение включает в себя устранение критических неисправностей в системе и реализуется часто не как проект, а как процессная деятельность.



Процессы жизненного цикла

Основным нормативным документом, регламентирующим состав процессов ЖЦ, является международный стандарт ISO/IBC 12207. В соответствии со стандартом все процессы ЖЦ разделены на три группы:

- Основные процессы
- Вспомогательные процессы
- Организационные процессы

Стандарт описывает структуру процессов ЖЦ, но не конкретизирует в деталях как реализовать или выполнить действия и задачи, включенные в эти процессы.

Стандарт определяет структуру жизненного цикла, содержащую **процессы, действия и задачи**, которые должны быть выполнены во время создания ПО.

Каждый процесс разделен на набор действий, каждое действие — на набор задач. Каждый процесс, действие или задача инициируется и выполняется другим процессом по мере необходимости, причем не существует заранее определенных последовательностей выполнения. Связи по входным данным при этом сохраняются.

Пример.

Процесс. Приобретение (действия и задачи заказчика, приобретающего ПО)

Действия

1. Инициирование приобретения
2. Подготовка заявочных предложений
3. Подготовка и корректировка договора
4. Надзор за деятельностью поставщика
5. Приемка и завершение работ

Задачи

1. Формирование требований к системе
2. Формирование списка программных продуктов
3. Установление условий и соглашений
4. Описание технических ограничений (среда функционирования системы и т. д.)

Основные процессы

- приобретение (заказ)
- поставка
- разработка
- эксплуатация
- сопровождение

Вспомогательные процессы

- документирование
- управление конфигурацией
- обеспечение качества
- верификация
- аттестация
- совместная оценка
- аудит
- решение проблем

Организационные процессы

- управление проектом
- создание инфраструктуры проекта
- усовершенствование: определение, оценка и улучшение самого ЖЦ
- обучение

Стадии и этапы жизненного цикла

Модель жизненного цикла ПО — структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении жизненного цикла. Модель жизненного цикла зависит от специфики, масштаба и сложности проекта и специфики условий, в которых система создается и функционирует.

Модель ЖЦ ПО включает в себя:

- Стадии;
- Результаты выполнения работ на каждой стадии;
- Ключевые события — точки завершения работ и принятия решений.

Стадия — часть процесса создания ПО, ограниченная определенными временными рамками и заканчивающаяся выпуском конкретного продукта (моделей, программных компонентов, документации), определяемого заданными для данной стадии требованиями.

Во всех стандартных моделях выделяют следующие основные **этапы жизненного цикла**:

1. планирование, бизнес-моделирование;
2. анализ (определение требований);
3. проектирование (предварительное и детальное);
4. реализация (или кодирование, программирование);
5. тестирование и отладка;
6. развертывание, внедрение;
7. эксплуатация и сопровождение.

Каждому этапу соответствуют определенный результат и набор документации, являющейся исходными данными для следующего этапа.

1. Requirement (определение требований);
2. Analysis&Design (анализ и проектирование);
3. Implementation (реализация, кодирование, программирование);
4. Testing (тестирование и отладка);
5. Deployment (развертывание);

Технологические процессы и фазы в методологии RUP

Один из самых известных процессов, использующих итеративную модель разработки — Rational Unified Process (RUP). Термин RUP означает как методологию разработки, так и продукт компании IBM для управления процессами разработки. Методология RUP описывает абстрактный общий процесс, на основе которого организация или проектная команда должна создать специализированный процесс, ориентированный на ее потребности.

Учитывая итерационный характер разработки в методологии Rational Unified Process (RUP) взамен этапам вводят понятия технологических процессов и стадий.

В RUP определено 9 технологических процессов, для каждого из которых предложена методика выполнения. Технологические процессы делятся на две категории – основные процессы и процессы поддержки.

К основным относятся:

1. Бизнес-анализ
2. Управление требованиями
3. Анализ и проектирование
4. Реализация
5. Тестирование
6. Развёртывание

Вспомогательные процессы включают:

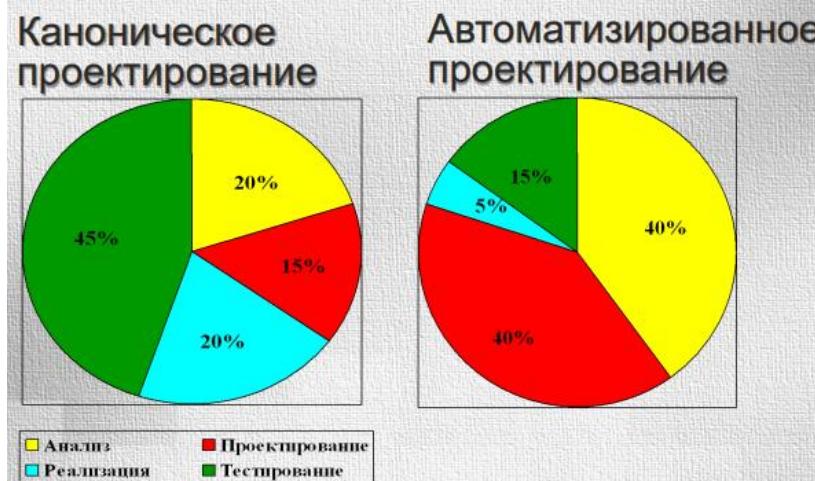
7. Управление проектом,
8. Управление конфигурацией,
9. Управление средой.

Для каждого технологического процесса предусмотрены:

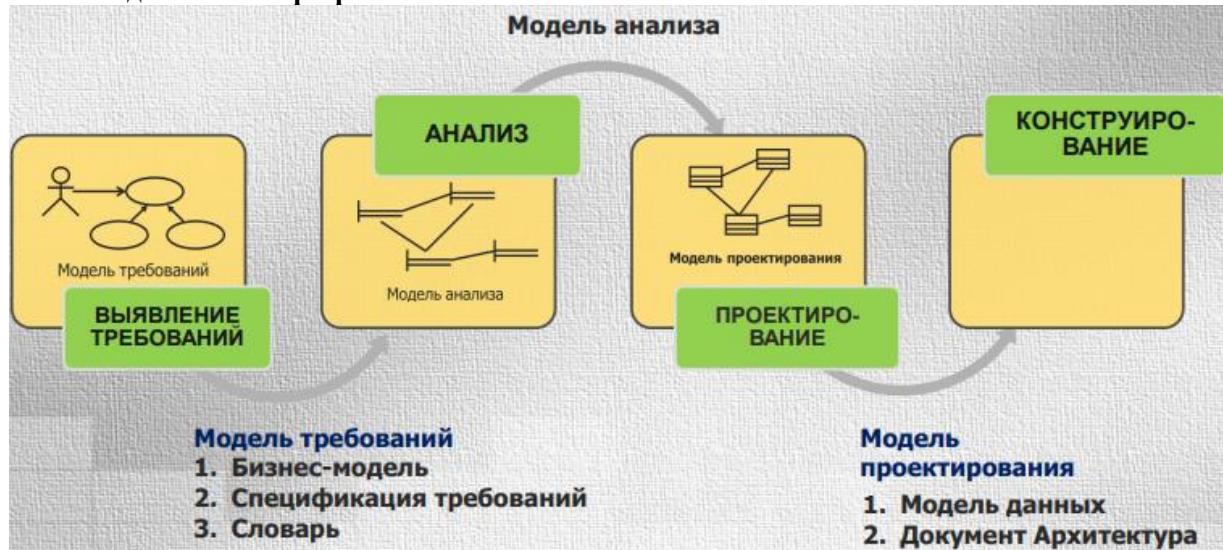
- роли, определяющие поведение и обязанности отдельных лиц и групп, работающих в одной команде (например, системный аналитик, тестировщик)

- виды деятельности, определяющие работы, выполняемые исполнителями (например, проектирование класса, проектирование ВИ)
- артефакты – документы, используемые, порождаемые или модифицируемые процессом. Основные артефакты в RUP – модель, элемент модели, документ, исходный код, исполняемая программа

Оценка трудозатрат по этапам ЖЦ



Модели этапов разработки



Начальная подготовка

- Область исследования
- Предмет исследования
- Проблема
- Потребности
- Цель
- Основные задачи

Бизнес-моделирование. Формирование компактного упрощённого представления о бизнесе, предназначенного для целостного представления и анализа деятельности всей системы взаимосвязанных бизнес-процессов.

Анализ. Требования к системе. Разработка требований к системе подразумевает определение ее функциональных возможностей, пользовательских требований, требований к надежности и безопасности, требований к внешним интерфейсам и т.д.

Проектирование (архитектурное и детальное). На этапе проектирования разрабатывается архитектура будущей ПС, а также программные компоненты и связи между ними.

Реализация. Реализация подразумевает выбор языка программирования и составление текста программы (кодирование), а также, возможно, выполнение тестирования и отладки отдельных фрагментов.

Тестирование и отладка. Этап тестирования и отладки включает выполнение комплексного тестирования всей программной системы специальной группой и исправление ошибок.

Интеграция и внедрение

- Интеграция отдельных компонентов системы в единый программный комплекс.
- Общая отладка всей программной системы (системное тестирование)
- Внедрение. Создание инсталляционного пакета, технической документации

Сопровождение и эксплуатация. На этапе сопровождения и эксплуатации программная система сдается в эксплуатацию, происходит обслуживание пользователей, возможно устранение незначительных ошибок.

Управление жизненным циклом приложений

Улучшению процессов создания программного обеспечения служит методология управления жизненным циклом приложений (**ALM** - application life cycle management), которая представляет собой концепцию управления программным проектом на всех этапах его жизни. ALM определяет непрерывный процесс управления жизненным циклом приложения по его управлению, развитию и обслуживанию.

Принципы ALM реализуются ИТ-решениями различных вендоров.

В разработке программного обеспечения, **стадии разработки** программного обеспечения используются для:

- описания степени готовности программного продукта.
- отражения количества реализованных функций, запланированных для определённой версии программы.

Стадии либо могут быть официально объявлены и регламентируются разработчиками, либо иногда этот термин используется неофициально для описания состояния продукта.

Каждой стадии разработки ПО присваивается определенный порядковый номер. Также каждый этап имеет свое собственное название, которое характеризует готовность продукта на этой стадии.

Жизненный цикл разработки ПО:

- Пре-альфа
- Альфа
- Бета
- Релиз-кандидат
- Релиз или RTM
- Пост-релиз
- Общая доступность (general availability, GA)

Стадии разработки ГОСТ 19.102-77

Настоящий стандарт устанавливает стадии разработки программ и программной документации для вычислительных машин, комплексов и систем независимо от их назначения и области применения.

1. Техническое задание. Обоснование необходимости разработки программы, научно-исследовательские работы, разработка и утверждение технического задания.
2. Эскизный проект. Разработка и утверждение эскизного проекта.
3. Технический проект. Разработка и утверждение технического проекта.
4. Рабочий проект. Разработка программы, разработка программной документации, испытания программы.

Стандарт ГОСТ 34.601-90

Стандарт ГОСТ 34.601-90 предусматривает следующие стадии и этапы создания автоматизированной системы (АС):

1. Формирование требований к АС
 - Обследование объекта и обоснование необходимости создания АС
 - Формирование требований пользователя к АС
 - Оформление отчета о выполнении работ и заявки на разработку АС
2. Разработка концепции АС
 - Изучение объекта
 - Проведение необходимых научно-исследовательских работ
 - Разработка вариантов концепции АС и выбор варианта концепции АС, удовлетворяющего требованиям пользователей
 - Оформление отчета о проделанной работе
3. Техническое задание
 - Разработка и утверждение технического задания на создание АС
4. Эскизный проект
 - Разработка предварительных проектных решений по системе и её частям
 - Разработка документации на АС и её части
5. Технический проект
 - Разработка проектных решений по системе и её частям
 - Разработка документации на АС и её части
 - Разработка и оформление документации на поставку комплектующих изделий

- Разработка заданий на проектирование в смежных частях проекта

6. Рабочая документация

- Разработка рабочей документации на АС и её части
- Разработка и адаптация программ

7. Ввод в действие

- Подготовка объекта автоматизации
- Подготовка персонала
- Комплектация АС поставляемыми изделиями (программными и техническими средствами, программно-техническими комплексами, информационными изделиями)
- Строительно-монтажные работы
- Пусконаладочные работы
- Проведение предварительных испытаний
- Проведение опытной эксплуатации
- Проведение приёмочных испытаний

8. Сопровождение АС

- Выполнение работ в соответствии с гарантийными обязательствами
- Послегарантийное обслуживание

Данный стандарт не вполне подходит для проведения разработок в настоящее время: многие процессы отражены недостаточно, а некоторые положения устарели.

1.10 Стандартные модели ЖЦ.

- Краткий исторический обзор.
- Стратегии разработки ПО.
- Пять видов основной деятельности. Макетирование. Защитная деятельность.
- Каскадная схема разработки ПО. V-метод.
- Итерационная модель.
- Спиральная модель ЖЦ ПО.
- Компонентно-ориентированная модель.

Краткий исторический обзор

Три способа

- Нисходящий способ. Это когда сначала глобально думают «что делать», потом «как делать», а потом уже делают, если времени осталось.
- Восходящий способ. Это когда сначала делают, понемножку, мелкими частями. Думают «что, да как» только над этой частью, не заморачиваясь на глобальное и непреходящее.
- Спиральный способ. Начинаем думать глобально, но по звонку будильника просыпаемся и выкатываем прототип. И так несколько раз, накапливая опыт, пока не появится решение согласно требованиям.

Нисходящая разработка. Создание способа приходится на 1960-е годы, органично дополняя требования структурного программирования. Глобальная задача делилась на более мелкие подзадачи, те, в свою очередь, на еще более детальные и так далее до уровня структурного блока, который можно непосредственно реализовать на языке программирования. В просторечии все нисходящие методы называют «водопадом».

Восходящий способ. На самом деле, восходящий способ еще древнее нисходящего. В 1950-х уже существовали и Фортран, и Кобол, но не было ясных представлений, как строить софт. Поэтому поступали самым естественным образом: сегодня сделаем одно, завтра другое, потом когда-нибудь склеим в третье. Если нужно реализовать несколько задач, то выбираем наиболее важные. Иногда способ также называют инкрементальным.

В 1990-х годах пошел мощный тренд замены "домашних" программистов на внешних консультантов. Одна из причин - усложнение технологий и специализация. Что улучшилось:

- планирование по-прежнему короткое, но охватывает не одну функцию, а их группу при этом срок этапа жестко ограничен;
- ведется протокол сделанного;
- теоретически, должно выделяться время на упрощение и чистку кода (рефакторинг);
- теоретически, риски регрессии должны парироваться всеобъемлющими тестами.

Плюсы восходящей разработки. Резко снижается порог входа. Чтобы запуститься "с нуля" не нужны дорогущие спецы по анализу и проектированию с опытом. Управлять процессом проще, контрольные точки и локальные цели прозрачны.

Проблемы. Проще управлять процессом, но практически невозможно - конечным результатом, так как он нигде четко не фиксируется. Риски перекладываются на заказчика. Если с технической архитектурой при наличии опыта, тестов и рефакторинга команда справляется (до определенного порога сложности), то с архитектурой функциональной дело плохо.

Сpirальная разработка. Упомянутые «водопадные» ограничения стали ясны еще в 1970-х, после массовых внедрений соответствующих методологий типа SADT/IDEF. Поэтому озадаченные проблемой исследователи выдали к середине 1980-х годов обновленное видение процесса софтостроения в виде спирали.

Принцип спиральной разработки. Следует ограничить сроки проектирования и жестко проверять принятые решения, выкатывая полный прототип. Прототип должен включать в себя большинство функций, пусть не до конца реализованных или даже с "заглушками". С полного прототипа, соответственно, получаем и полный фидбэк: здесь ошиблись, не поняли сути требований, тут не учли среду эксплуатации, там выходной формат не стыкуется с входным и т.д. С этим фидбэком начинаем второй виток спирали, обоснованно полагая высоким шанс довести следующий прототип до уровня готового продукта.

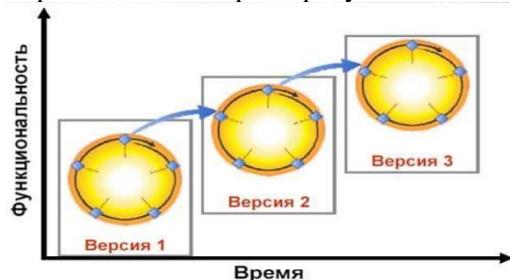


В программной инженерии, процесс разработки программного обеспечения представляет собой процесс разделения разработки программного обеспечения работы на отдельные фазы, чтобы улучшить дизайн, управление продуктом и управление проектами. Он также известен как жизненный цикл разработки программного обеспечения (SDLC). Методология может включать предварительное определение конкретных результатов и артефактов, которые создаются и завершаются проектной группой для разработки или поддержки приложения.

Стратегии разработки ПО

1. Однократный проход (водопадная стратегия)
2. Инкрементная стратегия (запланированное улучшение продукта). В начале процесса определяются все пользовательские и системные требования. Первая версия реализует часть запланированных возможностей, следующая версия реализует дополнительные возможности и т.д., пока не будет получена полная система.
3. Эволюционная стратегия. Система также строится в виде последовательности версий, но вначале процесса определены не все требования. Требования уточняются в результате разработки версий.

Стратегии 2 и 3 характеризуются так называемым итеративным подходом (рис.):



Пять видов основной деятельности

- Подготовка (сбор требований)
- Планирование (оценка, расписание)
- Моделирование (анализ и проектирование)
- Конструирование (кодирование и тестирование)
- Развёртывание (поставка и сопровождение)

Для многих программных проектов эти виды деятельности выполняются итеративно.

Обобщённый базис процессов для программной инженерии включает пять видов основной деятельности:

1. Подготовка. Подготовка заключается в тесном сотрудничестве с заказчиком и другими заинтересованными лицами. Главное – понять цели заинтересованных лиц в отношении продукта и проекта, собрать их требования к характеристикам и функциям ПО.

2. Планирование. Деятельность планирования создает план программного проекта. План определяет порядок инженерной работы. Он описывает технические задачи, которые надо выполнять, наиболее вероятные факторы риска, подстерегающие команду, требуемые ресурсы, рабочие продукты (модели, документы, данные, отчеты, формы и т. д.), которые будут созданы, и расписание работы.

3. Моделирование. Модель – упрощённое представление реальности. Модели наглядно демонстрируют желаемую структуру и поведение системы (обычно визуально). Модели позволяют лучше понять общую картину эскиз будущего решения. При необходимости эскиз дополняется деталями. Так формируется окончательный способ решения проблемы. Обычно моделирование включает в себя два действия: анализ и проектирование. Модель анализа улучшает понимание требований к ПО, а модель проектирования показывает эскиз структуры и поведения ПО, выполняющего эти требования.

4. Конструирование. Эта деятельность объединяет два действия: генерацию программного кода ПО (ручную или автоматическую) и тестирование, которое требуется для обнаружения ошибок в коде.

5. Развертывание. ПО (окончательный вариант или частично завершенная версия) поставляется заказчику, который оценивает полученный продукт и обеспечивает обратную связь, дающую возможность улучшения продукта. Последний этап нацелен на два действия: поставку разработанного продукта заказчику и сопровождение процесса эксплуатации этого продукта.

Макетирование

Достаточно часто заказчик не может сформулировать подробные требования по вводу, обработке или выводу данных для будущего программного продукта. С другой стороны, разработчик может сомневаться в приспособленности продукта под операционную систему, форме диалога с пользователем или эффективности реализуемого алгоритма. В этих случаях целесообразно использовать макетирование. Основная цель макетирования - снять неопределенности в требованиях заказчика. Макетирование или прототипирование - это процесс создания модели требуемого программного продукта.

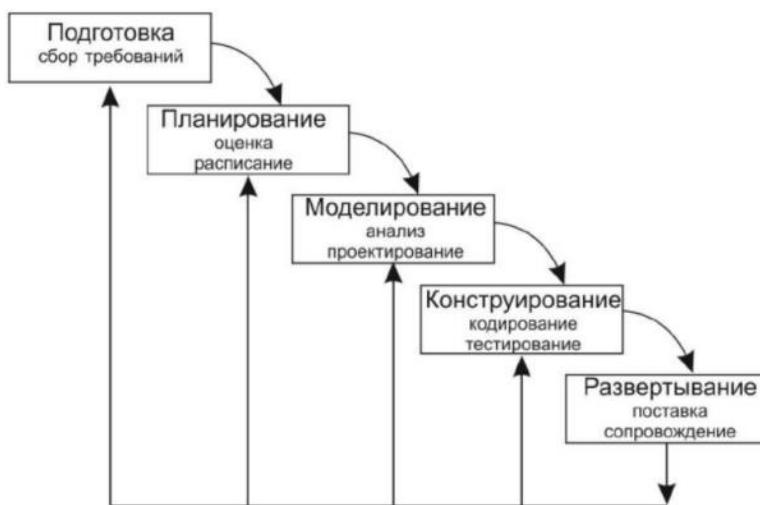
Защитная деятельность

Виды основной деятельности по разработке ПО дополняются набором видов защитной деятельности. Виды защитной деятельности «пронизывают» весь программный проект и помогают команде управлять процессом, контролировать развитие, качество, изменения, риск. Виды защитной деятельности

- Отслеживание (трассировка) и контроль программного проекта
- Управление риском
- Обеспечение качества ПО
- Технические проверки
- Измерение
- Управление конфигурацией ПО
- Управление повторной используемостью (reuse)
- Подготовка и производство программного продукта

Каскадная модель (модель водопада).

Первой по времени появления, и самой распространенной являлась каскадная модель или модель водопада.



Каскадная модель (модель водопада) предполагает следующие свойства взаимодействия этапов:

- модель состоит из последовательно расположенных этапов;
- каждый этап полностью заканчивается до того, как начнется следующий;
- этапы не перекрываются во времени, т.е. следующий этап не начинается, пока не завершится предыдущий;
- возврат к предыдущим этапам не предусматривается или крайне ограничен;
- результат появляется только в конце разработки.

Преимущества. Чем больше продумано в начале, тем меньше лишней работы придется делать в середине и конце. Исключается дублирование функций и ошибок. Нужны хорошие аналитики, проектировщики с опытом работы в данных предметных областях. И даже в этом случае есть риск затянуть планирование реализации на неприличный срок.

Главный недостаток заключается в том, что любое внесение изменений в требования к ПС обходится достаточно дорого. Это происходит потому, что необходимость внесения изменений обнаруживается при таком подходе обычно к концу разработки, поэтому требуется править модель всей системы и вносить изменения в

программный код большого объема. Такой подход хорош для маленьких проектов и в тех случаях, когда требования к ПС строго определены и гарантированно не будут изменяться.

V-метод.

В дальнейшем способ неоднократно улучшался, например, V-метод, где стадиям разработки по левому спуску буквы «V» соответствуют стадии тестов и приемок подъема справа. По горизонтальному уровню сразу видно, какой проектный документ подрядчика служит основанием для приемочного документа заказчика.



Итерационная модель (iterative and incremental development, IID).

Модель IID предполагает разбиение жизненного цикла проекта на последовательность итераций, каждая из которых напоминает «мини-проект», включая все процессы разработки в применении к созданию меньших фрагментов функциональности, по сравнению с проектом в целом. Цель каждой итерации — получение работающей версии программной системы, включающей функциональность, определённую интегрированным содержанием всех предыдущих и текущей итерации. Результат финальной итерации содержит всю требуемую функциональность продукта. Таким образом, с завершением каждой итерации продукт получает приращение — инкремент — к его возможностям, которые, следовательно, развиваются эволюционно. Итеративность, инкрементальность и эволюционность в данном случае есть выражение одного и то же смысла разными словами со слегка разных точек зрения.

Инкрементная модель (рис. 1) подразумевает пошаговое добавление функционала к уже существующему продукту, что позволяет быстро получить рабочий прототип. В то же время итеративная модель (рис. 2) предполагает разработку продукта в итерациях, с более четким планированием и контролем процесса.



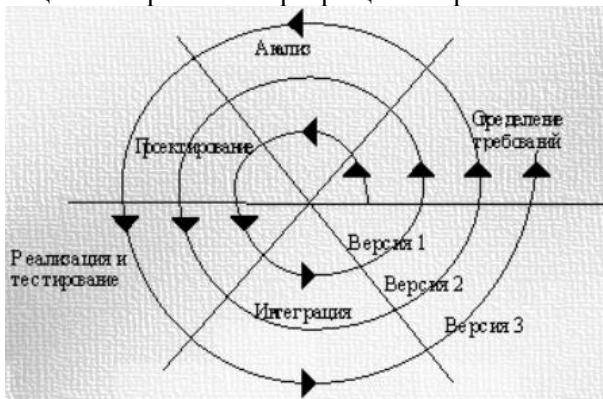
Итеративная разработка обладает рядом преимуществ по сравнению с последовательной моделью.

- Реализация наиболее важных функций может быть завершена в ходе нескольких первых итераций. После их завершения (то есть намного раньше окончания всего проекта) заказчик сможет начать использование системы.
- Уже в начале проекта пользователи получают возможность оценить функциональность системы и ее соответствие своим потребностям. Необходимые изменения и дополнения могут быть сделаны в течение следующих итераций.
- Основные проектные риски могут (и должны) быть разрешены на первых итерациях. Например, архитектурное решение, приводящее к неприемлемой производительности может быть обнаружено и исправлено уже в первой итерации.

Сpirальная модель - классический пример применения эволюционной стратегии разработки. Спиральная модель, предложенная Барри Боэмом в 1986 году, стала существенным прорывом в понимании природы разработки ПО. При использовании этой модели ПО создается в несколько итераций (витков спирали) методом прототипирования. Делается упор на начальные этапы жизненного цикла: анализ и проектирование. Каждый виток спирали соответствует созданию фрагмента или версии программного обеспечения, на нем уточняются цели и характеристики проекта, определяется его качество и планируются работы следующего витка спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который доводится до реализации.

На каждой итерации оцениваются:

1. риск превышения сроков и стоимости проекта;
2. необходимость выполнения ещё одной итерации;
3. степень полноты и точности понимания требований к системе;
4. целесообразность прекращения проекта.



Отличительной особенностью этой модели является специальное внимание рискам, влияющим на организацию жизненного цикла. Бозем формулирует десять наиболее распространённых (по приоритетам) рисков:

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

В сегодняшней спиральной модели определён следующий общий набор контрольных точек:

- Concept of Operations (COO) — концепция (использования) системы;
- Life Cycle Objectives (LCO) — цели и содержание жизненного цикла;
- Life Cycle Architecture (LCA) — архитектура жизненного цикла; здесь же возможно говорить о готовности концептуальной архитектуры целевой программной системы;
- Initial Operational Capability (IOC) — первая версия создаваемого продукта, пригодная для опытной эксплуатации;
- Final Operational Capability (FOC) — готовый продукт, развернутый (установленный и настроенный) для реальной эксплуатации.

На каждом витке спирали могут применяться разные модели процесса разработки ПО. В конечном итоге на выходе получается готовый продукт. Неполное завершение работ на каждом этапе позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем. При итеративном способе разработки недостающую работу можно будет выполнить на следующей итерации. Главная задача — как можно быстрее показать пользователям системы работоспособный продукт. Основная проблема спирального цикла — определение момента перехода на следующий этап. Для ее решения необходимо ввести временные ограничения на каждый из этапов жизненного цикла. Переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена.

Компонентно-ориентированная модель

Компонентно-ориентированная модель является развитием спиральной модели и тоже основывается на эволюционной стратегии разработки. В этой модели модифицируется содержание квадранта моделирования конструирования - оно отражает тот факт, что в современных условиях новая разработка должна основываться на повторном использовании существующих программных компонентов.

Программные компоненты, созданные в реализованных программных проектах, хранятся в библиотеке. В новом программном проекте, исходя из требований заказчика, появляются кандидаты в компоненты, далее проверяется наличие этих кандидатов в библиотеке. Если они найдены, то компоненты извлекаются из библиотеки и используется повторно. В противном случае создаются новые компоненты, они применяются в проекте и включаются в библиотеку.

1.11 Иерархическая структура работ проекта

- Два основных принципа построения ИСР.
- Продуктовая и функциональная иерархические структуры.
- Принципы декомпозиции:
 - по фазам ЖЦ
 - по ключевым результатам проекта
 - по организационной структуре проекта
 - по подпроектам
- Этапы разработки иерархической структуры работ

Иерархическая структура работ (ИСР) — это иерархическое разбиение всей работы, которую необходимо выполнить для достижения целей проекта, на более мелкие операции и действия до такого уровня, на котором способы выполнения этих действий вполне ясны и соответствующие работы могут быть оценены и спланированы. Она включает также определение промежуточных результатов всех составляющих эту структуру работ.

ИСР обеспечивает выявление всех работ, необходимых для достижения целей проекта. Если ИСР составлена корректно, то любая работа, которая в нее не вошла, не может считаться работой по проекту. ИСР делит проект на субпроекты, пакеты работ, субпакеты. Каждый следующий уровень декомпозиции обеспечивает последовательную детализацию содержания проекта, что позволяет производить оценку сроков и объемов работ. ИСР должна включать все промежуточные и конечные продукты.

Два основных принципа построения ИСР:

1. Правило 100%. Оно говорит о том, что в ИСР должны быть включены все результаты, продукты и работы, которые должны быть выполнены по проекту, вне зависимости от того это внешние, внутренние результаты или переданные для выполнения З-м лицам. Данное правило также применяется ко всем нижестоящим компонентам.

2. Взаимоисключаемость элементов. Данный принцип говорит о том, что в ИСР не может быть 2 элементов с одинаковым содержанием. Т.е. не может быть 2-х одинаковых элементов в разных ветках ИСР. Если это допустить, вас ожидает дублирование работы и непонимание командой проекта своих работ.

Декомпозиция

Существует несколько подходов к разбиению результатов проекта на элементы. Основными являются два из них: **продуктовый и функциональный**. Данные критерии совмещать на одном уровне нельзя, а на разных уровнях иерархии допустимо и даже целесообразно.

Продуктовый подход необходим РМ для общения с заказчиком. Заказчику важно видеть, что и когда менеджер будет ему сдавать, все ли продукты проекта будут ему представлены и т.п.

Функциональный подход удобен менеджеру для взаимодействия с исполнителями. По нему проще назначить ответственных за отдельные блоки операций.

Иерархическая структура работ, формируемая под руководством РМ, на верхних уровнях иерархии может использовать любой критерий, исходя из логики декомпозиции. Часто, если проект выполняется в несколько этапов, может использоваться подход, исходя из кубической модели структур проекта. На верхнем уровне мы разбиваем результаты по этапам жизненного цикла проекта, затем структурируем по производимым проектным продуктам, и, наконец, пакеты работ комплектуем по функциональному критерию или по видам деятельности.

Принципы декомпозиции

1. По фазам жизненного цикла. Например, проект выполняется по таким фазам: Продажа, Аналитика, Проектирование, Дизайн, Продакшн, Верстка и т.д. Можно представить все эти объекты на первом уровне, чтобы дальше разбить их на более измеримые кусочки. Это удобно, когда у руководителя проекта и заказчика нет разногласий по набору фаз проекта, а также каждая фаза имеет понятный всем участникам набор результатов.

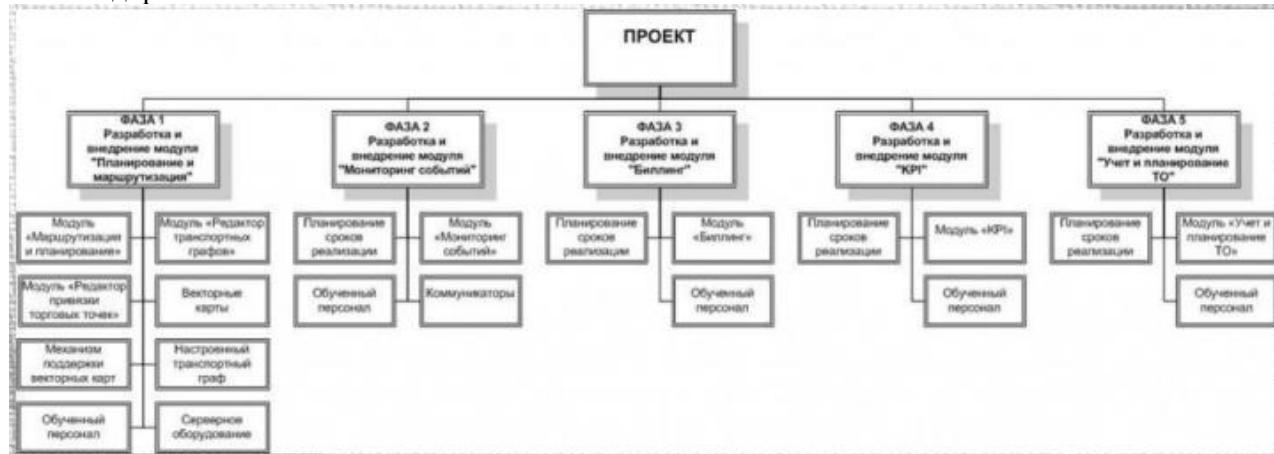
2. По ключевым результатам проекта (deliverables). Например, проект по внедрению системы управления транспортом может иметь такой набор результатов: Информационная система, Карты для маршрутизации транспорта, Векторные графы дорог и маршрутов, Механизм поддержки, Обученный персонал, Оборудование. Эти результаты удобно отразить на первом уровне, чтобы заказчик ясно и точно видел, что будет сдано ему по завершению проекта. Здесь важно помнить, что если результат не представлен на ИСР, то в проекте он не будет получен.

3. По организационной структуре проекта. Например, в вашем проекте есть 4 структуры, которые вовлечены в реализацию проекта:

- бизнес-заказчик. Он может отвечать за результаты:
 - бизнес требования к системе
 - подготовленные к загрузке в систему данные
 - обученные пользователи
 - бизнес-тестирование и т.д.
- ИТ со стороны заказчика. Они могут отвечать за:
 - серверы
 - закупленные лицензии

- интерфейс обмена с другими учетными системами и т.д.
- вендор (поставщик программного продукта) отвечает за
 - аудит технической архитектуры
 - компания — внедренец программного обеспечения (ваша структура):
 - техническое задание
 - настройка системы
 - доп. программирование системы и т.п.
- компания — внедренец программного обеспечения (ваша структура):
 - техническое задание
 - настройка системы
 - доп. программирование системы и т.п.

4. По подпроектам



Этапы разработки иерархической структуры работ

Этап 1. Определение конечных результатов проекта. Рекомендуется рассмотреть и проанализировать документы, описывающие общий объем работ по проекту: концепцию, устав, бизнес-план (если он есть).

Этап 2. Выбор критериев разбиения на элементы. Среди продуктового, функционального и организационного критериев разбиения нужно выбрать последовательность декомпозиции результатов на элементы.

Этап 3. Определение основных пакетов работ. Основой для этого могут служить результаты, необходимые для создания продукта проекта, но не являющиеся его целями.

Этап 4. Определение степени детализации работ. Степень детализации должна соответствовать внутренней системе управления и единой системе контроля. Контроль не может стать самоцелью.

Этап 5. Анализ и усовершенствование. Продолжаются до тех пор, пока все участники проекта не будут удовлетворены результатами предварительного планирования.

Современный процесс разработки коммерческого ПО должен быть инкрементальным. Это означает, что на верхнем уровне декомпозиции проекта должны находиться продукты проекта, а на следующем уровне — компоненты, из которых эти продукты состоят. Компоненты далее могут быть декомпозированы на «фиши» — функции, которые они должны реализовывать.

Выделение компонентов, составляющих программный продукт, — это элемент высокого уровня проектирования, которое должно выполняться на фазе планирования проекта, не дожидаясь проработки всех функциональных требований к разрабатываемому ПО.

1.12 Методологии и технологии проектирования ПО. Требования к технологиям.

- Основа проекта ПС: методологии, технологии, инструментальные средства.
- Определения: метод, способ, технология, методология
- Технологии программирования, проектирования. Технологическая операция.
- Стандарты, стандарты проектирования.
- CASE-технологии.

Основа проекта ПС:

Методологии, технологии и инструментальные средства проектирования (CASE-средства) составляют основу проекта любой ПС. Проект ПС = Методология + Технологии + CASE

Методология — совокупность методов, применяемых в какой-либо науке. Методология реализуется через конкретные технологии и поддерживающие их стандарты, практики, методики и инструментальные средства, которые обеспечивают выполнение процессов ЖЦ. Другими словами, методология соединяет вместе технологии, стандарты и инструментальные средства в единое пространство проектирования. Каждая методология делает это по-разному.

Метод – логическая последовательность действий, приводящая к достижению поставленной цели.

Способ – средства + методы + формы деятельности.

Методика – совокупность способов.

Технология – методика, дающая гарантированный результат.

Создание программных продуктов – технологический процесс. При этом применяются различные технологии. Эти технологии относятся не только к кодированию, но и к анализу, проектированию, управлению проектом, разработке документации.

Технологии

- технологии программирования
- технологии проектирования
- технологии управления проектом
- технологии анализа
- технологии разработки
- технологии сопровождения

Технология – совокупность производственных процессов в определенной отрасли производства, а также научное описание способов производства.

Технология программирования – совокупность методов, приемов и средств для сокращения стоимости и повышения качества разработки программных систем.

Технологии проектирования определяются как совокупность трех составляющих:

1. пошаговой процедуры, определяющей последовательность технологических операций проектирования;
2. критериев и правил, используемых для оценки результатов выполнения технологических операций;
3. нотаций (графических и текстовых средств), используемых для описания проектируемой системы.

Технологическая операция



Реальное применение любой технологии проектирования, разработки и сопровождения ИС в конкретной организации и конкретном проекте невозможно без выработки ряда стандартов (правил, соглашений), которые должны соблюдаться всеми участниками проекта. К таким **стандартам** относятся следующие:

- стандарт проектирования;
- стандарт оформления проектной документации;
- стандарт пользовательского интерфейса.

Стандарт проектирования должен устанавливать:

- набор необходимых моделей (диаграмм) на каждой стадии проектирования и степень их детализации;
- правила фиксации проектных решений на диаграммах, в том числе: правила именования объектов (включая соглашения по терминологии), набор атрибутов для всех объектов и правила их заполнения на каждой стадии, правила оформления диаграмм, включая требования к форме и размерам объектов, и т. д.;
- требования к конфигурации рабочих мест разработчиков, включая настройки операционной системы, настройки CASE-средств, общие настройки проекта и т. д.;
- механизм обеспечения совместной работы над проектом, в том числе: правила интеграции подсистем проекта, правила поддержания проекта в одинаковом для всех разработчиков состоянии (регламент обмена проектной информацией, механизм фиксации общих объектов и т.д.), правила проверки проектных решений на непротиворечивость и т. д.

Назначение CASE-технологий:

- Применяются прежде всего на этапах анализа и проектирования
- Обеспечивают качество принимаемых технических решений

- Поддерживают подготовку проектной документации
- Применение средств визуального (наглядного) представления информации
 - построение структурных и иных диаграмм (моделей) в реальном масштабе времени
 - использование многообразной цветовой палитры
 - сквозная проверка синтаксических правил

1.13 Технологии программирования.

- Структурное программирование
- Модульное программирование
- Объектно-ориентированное программирование
- Компонентное программирование
- Разработка, управляемая моделями MDA
- Автоматное программирование

Основные этапы развития технологии разработки

- *Первый этап – «стихийное» программирование.*
- *Второй этап – структурный подход к программированию (60 – 70-е годы XX в.).*
- *Третий этап – объектный подход к программированию (с середины 80-х годов до нашего времени).*
- *Четвертый этап – компонентный подход и CASE-технологии (с середины 90-х годов до нашего времени).*
- *Пятый этап – разработка, ориентированная на архитектуру и CASE-технологии (с начала XXI в. До нашего времени).*

1 этап. Стихийное программирование

- Использовалась до середины 60-х годов.
- Отсутствуют технологии, программирование – это искусство.
- Типичная программа состояла из основной программы, области глобальных данных, и набора подпрограмм, выполнявших обработку всех данных или их части.
- Самое слабое место – при увеличении количества подпрограмм возрастала вероятность искажения части глобальных данных какой-либо подпрограммой.
- Чтобы сократить количество таких ошибок в подпрограммах, было предложено применять локальные данные.

Переход к следующей технологии

- В начале 60-х годов разразился кризис программирования.
- Это было вызвано несовершенством технологии программирования. Стихийно использовалась разработка «снизу-вверх».

2 этап. Структурное программирование (60-70 годы)

В основе СП лежит процедурная декомпозиция сложных систем с целью последующей реализации в виде отдельных небольших подпрограмм. Такой подход требовал представления задачи в виде иерархии подзадач простейшей структуры. Таким образом проектирование осуществлялось «сверху вниз» и подразумевало реализацию общей идеи с проработкой интерфейсов подпрограмм.

Центральный технологический принцип структурного программирования состоит в том, что формулировку алгоритма и его запись в виде программы рекомендуется выполнять на основе базиса из трех алгоритмических конструкций, применяя при необходимости их суперпозицию. Результатом последовательного применения этого принципа будет более ясная структура программы.

Переход к следующей технологии.

Дальнейший рост размеров и сложности разрабатываемого ПО потребовал развития структурирования данных. Появляется возможность определения пользовательских типов данных. Одновременно усиливается стремление разграничить доступ к глобальным данным программы. В результате появилась и начала развиваться технология модульного программирования.

Модульное программирование

МП предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные, в отдельно компилируемые модули (библиотеки подпрограмм). Например, модуль графических ресурсов, модуль подпрограмм вывода на принтер. Связи между модулями осуществляются через специальный интерфейс, в то время как доступ к реализации модуля (телам подпрограмм и внутренним переменным) запрещен. Этую технологию поддерживают версии современных языков.

Модульное программирование состоит в разработке под конкретную задачу или круг задач (предметную область) собственного базиса в виде набора модулей, позволяющего наиболее эффективно по целому ряду критериев построить программный комплекс. Модули, входящие в базис, это целые программы, решающие некоторые подзадачи основных задач.

С применением модульного программирования появляются возможности коллективной разработки программ как набора «независимых» частей, последовательного уменьшения сложности методом разбиения сложной задачи на более простые подзадачи, наконец, возможности повторного использования созданного ранее кода.

Переход к следующей технологии

• Практика показала, что структурный подход в сочетании с модульным программированием позволяет получать надежные программы, размер которых не превышает 100 000 операторов.

• Узким местом модульного программирования является то, что ошибка в интерфейсе при вызове подпрограммы выявляется только при выполнении программы. Из-за раздельной компиляции модулей обнаружить эти ошибки раньше невозможно.

• При увеличении размера программы обычно возрастает сложность межмодульных интерфейсов и с некоторого момента предусмотреть взаимовлияние отдельных частей программы становится невозможным.

3 этап. Объектно-ориентированное программирование (с середины 80 до конца 90)

В основе технологии лежат объектная модель и объектная декомпозиция. К основным принципам объектной модели часто относят следующие:

- абстракция;
- инкапсуляция;
- иерархия (наследование, агрегация);
- полиморфизм;
- модульность.

Суть объектной декомпозиции состоит в выделении в предметной области классов и объектов, а также связей между ними, и лишь потом данных и алгоритмов, которыми характеризуется каждый класс. Таким образом, именно классы становятся основным «строительным блоком» в ООП, тогда как ранее таковыми блоками являлись алгоритмы. Основные методы и концепции ООП:

• Метод объектно-ориентированной декомпозиции – заключается в выделении объектов и связей между ними. Метод поддерживается концепциями инкапсуляции, наследования и полиморфизма.

• Метод абстрактных типов данных – метод, лежащий в основе инкапсуляции. Поддерживается концепцией абстрактных типов данных.

• Метод пересылки сообщений – заключается в описании поведения системы в терминах обмена сообщениями между объектами. Поддерживается концепцией сообщения.

4 этап. Компонентное программирование (с середины 90-х и до нашего времени)

Компонентное программирование представляет собой развитие объектно-ориентированной технологии. В отличие от ООП введен следующий уровень абстракции – классы объединяются в компоненты. Компонент:

- программный код в виде самостоятельного модуля;
- может быть использован в неизменном виде;
- может допускать настройку;
- обладает поведением (функциональностью).

Основной принцип компонентного программирования: сборка приложения из готовых компонент, в общем случае написанных на разных языках.

Компонент изолирован от внешнего мира своим интерфейсом – набором методов (их сигнатурами).

Компонентная программа – набор независимых компонент, связанных друг с другом посредством интерфейсов.

Технологии компонентного подхода

- Технологии на базе COM (DCOM) (компонентная модель объекта)
 - OLE-automation
 - ActiveX
 - MTS (сервер управления транзакциями)
 - MIDAS (сервер многозадачных распределенных приложений)
- Технология создания распределенных приложений CORBA
 - Компонентное сборочное программирование — объектно-ориентированное сборочное программирование, основанное на распространении классов в бинарном виде и предоставление доступа к методам класса через строго определенные интерфейсы.
 - Аспектно-ориентированное сборочное программирование — разновидность сборочного программирования, основанная на сборке полнофункциональных приложений из многоаспектных компонентов, инкапсулирующих различные варианты реализации.
 - Объектно-ориентированное сборочное программирование — разновидность сборочного программирования, основанная на методологии объектно-ориентированного программирования и

предполагающая распространение библиотек классов в виде исходного кода или упаковку классов в динамически компонуемую библиотеку.

- *Модульное сборочное программирование — разновидность сборочного программирования, основанная на процедурах и функциях методологии структурного императивного программирования.*

5 этап. Разработка, управляемая моделями и CASE-технологии (с начала XXI в. до нашего времени)

Разработка, управляемая моделями, (model-driven development) – это стиль разработки программного обеспечения, когда модели становятся основными артефактами разработки, из которых генерируется код и другие артефакты.

Модели, описанные на предметно-ориентированном языке программирования, могут быть использованы, как точки расширения каркасов. Наиболее известными MDE-инициативами являются:

- концепция модельно-ориентированного подхода к разработке программного обеспечения (Model Driven Architecture), разрабатываемая консорциумом OMG.
- Eclipse Modeling Framework — свободный фреймворк, основанный на Eclipse, для генерации кода, инструментов и прочих приложений на основе структурированной модели данных (метамодели).

Model Driven Architecture – модельно-ориентированный подход к разработке программного обеспечения.

Для конструирования программного приложения должна быть построена подробная, формально точная модель, из которой потом может быть автоматически генерирован исполняемый программный код приложения.

Автоматное программирование (90-е годы)

Автоматное программирование — парадигма программирования, при использовании которой программа или её фрагмент осмысливается как модель какого-либо формального автомата. Известна также и другая «парадигма автоматного программирования, состоящая в представлении сущностей со сложным поведением в виде автоматизированных объектов управления, каждый из которых представляет собой объект управления и автомат».

Определяющими для автоматного программирования являются следующие особенности:

- временной период выполнения программы разбивается на шаги автомата, каждый из которых представляет собой выполнение определённой (одной и той же для каждого шага) секции кода с единственной точкой входа; такая секция может быть оформлена, например, в виде отдельной функции и может быть разделена на подсекции, соответствующие отдельным состояниям или категориям состояний

- передача информации между шагами автомата осуществляется только через явно обозначенное множество переменных, называемых состоянием автомата; между шагами автомата программа (или её часть, оформленная в автоматном стиле) не может содержать неявных элементов состояния, таких как значения локальных переменных в стеке, адреса возврата из функций, значение текущего счётчика команд и т. п.

Полностью выполнение кода в автоматном стиле представляет собой цикл шагов автомата.

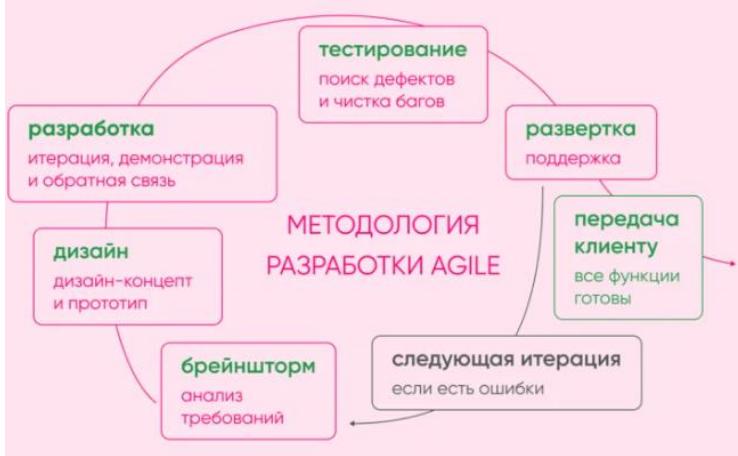
1.14 Обзор современных методологий разработки ПО

Методология Waterfall (каскадная или водопадная модель). Водопадная модель разработки — это линейный подход к разработке программного обеспечения. Она характеризуется строгим, структурированным процессом. Основной принцип — каждый этап проекта должен быть завершен до начала следующего. Как правило, модель состоит из следующих этапов разработки: системного анализа, анализа требований, проектирования, реализации, тестирования, внедрения и сопровождения. Эта методология подходит для хорошо продуманных проектов, в которых требования строго зафиксированы и вряд ли сильно изменяются в процессе разработки. Она предлагает четкие рамки проекта, что облегчает управление ресурсами и распределение времени.



Методология Agile (гибкая методология разработки). Методология Agile — это популярный подход, в котором основное внимание уделяется гибкости, сотрудничеству и оптимизации процессов для реализации качественного проекта. Это итеративный подход, и приоритет в нем отдается обратной связи от владельца продукта

и адаптации к изменяющимся требованиям. Цикл разработки ПО по Agile-методологии можно разбить на шесть этапов: планирование, проектирование, разработка, тестирование, развертывание и обслуживание.



Методология Scrum. Scrum — это система управления проектами, основанная на принципах Agile. Она предназначена для команд из десяти и менее человек. Они разделяют работу на небольшие задачи, которые нужно сделать в течение определенного времени. Как и в Agile, эти временные интервалы называются «спринты». Scrum способствует самоорганизации команды: они решают сложные задачи, анализируют победы и поражения и улучшают работу. Модель разработки ПО Scrum построена таким образом, чтобы помочь командам естественным образом адаптироваться к меняющимся условиям рынка и потребностям пользователей. В то же время короткие циклы позволяют разработчикам быть эффективнее. Scrum обеспечивает структуру, оптимизирует разработку и при этом остается гибким и учитывает желания владельца продукта.



«V-Model» («шаг за шагом» или validation and verification). Модель разработки ПО ориентирована на то, чтобы детально проверять и тестировать продукт на первых стадиях разработки. Уже в момент написания кода разработчиками тестировщики пишут модульные тесты, то есть начинают тестирование параллельно с разработкой. Рекомендуется придерживаться данного подхода, если вам крайне важно бесперебойное функционирование продукта, а также известны четкие требования. V-Model относят к практикам экстремального программирования.



Быстрая разработка приложений (RAD). Быстрая разработка приложений — это итеративная методология, при использовании которой важно разработать продукт быстро и, если необходимо, создать несколько прототипов. Метод Rapid Application Development (RAD) основан на обратной связи от пользователей и совместной работе всех членов команды, что позволяет ускорить выполнение проекта и избежать проблем после запуска. Работая по модели RAD, команда использует инструменты и фреймворки быстрой разработки и обычно опирается на визуальные среды разработки — они помогают создавать ПО в кратчайшие сроки. В рамках этой модели разработки программного обеспечения, продукт регулярно тестируют. И взаимодействие с пользователями помогает сделать так, чтобы ожидание и реальность совпали.



Spiral (спиральная модель). Данная модель направлена на анализ оценки рисков. И отлично подойдет там, где нет права на ошибку. Также ее удобно использовать для введения новых линеек продукта и проведения исследований. Выглядит эта модель, как спираль, так как начинает оценивать риски сначала на локальных программах, пытаясь предотвратить риски, далее переходит на новый виток спирали, перенаправляясь к работе с более комплексными задачами. Чаще всего мы проходим такие этапы, как: планирование, анализ рисков, конструирование, оценку результата — так, если он удовлетворяет, мы переходим на новый виток.



Экстремальное программирование (XP). Экстремальное программирование — это тоже методология на базе Agile. Для нее характерны командная работа, общение и быстрый фидбек. XP считается одной из самых радикальных форм Agile и сильно отличается от других подходов. XP делает акцент на клиентоориентированности и побуждает разработчиков ПО творчески подходить к работе. Важную роль в экстремальном программировании играют тестирование и проверка качества кода. Это нужно для того, чтобы избежать ошибок и в кратчайшие сроки запустить качественное ПО.



Методология Lean (бережливая разработка). Бережливая разработка — еще одна методология разработки на базе Agile. Она направлена на повышение эффективности за счет того, что все лишнее убирается из процесса. Разработчики делают акцент только на том, что необходимо, избегают ненужной работы и тем самым повышают качество продукта. Если убрать задачи и действия, не приносящие реальной пользы, члены команды достигают оптимальной эффективности. В данном случае к «ненужному» можно отнести дополнительные функции,

избыточный код, неэффективные процессы и излишнюю бюрократию. Проект делится на небольшие задачи, которые можно закончить быстро. Это сокращает время работы и упрощает обратную связь.



1.15 Стратегии разработки программной системы

- нисходящая («сверху–вниз»)
- восходящая («снизу–вверх»)
- «расширения ядра» («изнутри к границам»)
- «формирования ядра» («от границ внутрь»)

В процессе проектирования и реализации программной системы как правило используется одна из следующих стратегий:

1. нисходящая («сверху–вниз»),
2. восходящая («снизу–вверх»),
3. «расширения ядра» («изнутри к границам»),
4. «формирования ядра» («от границ внутрь»).

Нисходящая стратегия разработки программной системы

Нисходящее (сверху вниз) проектирование начинается с постановки задачи, которую требуется решить, и развивается далее путем разбиения ее на подзадачи. В конечном счете, это приводит к таким подзадачам, решение которых легко выразить в терминах базовых конструкций. Процесс проектирования включает в себя следующие действия:

- производится декомпозиция общей задачи на точно определенные подзадачи;
- доказывается, что если каждая подзадача решена корректно и полученные решения взаимосвязаны, то первоначальная задача будет решена корректно.

Процесс декомпозиции повторяется для подзадач, и доказывается корректность для выполненного разбиения. Этот процесс повторяется до получения подзадач настолько простых, что решение выражается базовыми конструкциями.

Такой способ проектирования алгоритмов и программ в виде иерархической структуры позволяет решать громоздкие и сложные задачи. При нисходящей разработке появляется возможность проверки алгоритма или проверки и отладки программы с самого начала разработки и тем самым создается ситуация выявления ошибок на ранних стадиях проектирования программ.

Восходящая стратегия разработки программной системы

Восходящее (снизу вверх) проектирование использует идею выделения достаточно крупных подалгоритмов и модулей, каждый из которых реализует определенные функции.

При восходящем методе проектирование системы начинается с определения отдельных задач внутри системы. Затем изучается, как решение этих задач может использоваться в качестве абстрактных инструментов для решения более общих задач.

Выбор модулей определяется различными соображениями: функциональной законченностью, размерами, однородностью структур данных и др. Для выделенных модулей составляются спецификации, определяется связь по входу и выходу. Далее каждый модуль алгоритма автономно разрабатывается и проверяется. После разработки модулей с учетом связей по входу и выходу осуществляется объединение в подсистемы, которые подвергаются контролю. Испытанные подсистемы объединяются в программную систему, которая подвергается комплексной проверке.

Следовательно, восходящее проектирование начинается с уровня базовых понятий и развивается путем формирования на их основе абстрактных понятий и, наконец, приводит к таким понятиям, в терминах которых выражается решение всей задачи.

Основными недостатками восходящего способа проектирования являются:

- сложность процесса объединения отдельных модулей в алгоритм;
- трудность исправления ошибок, допущенных на ранних стадиях разработки;
- сложность контроля разработки и прогноза завершения.

Стратегии «расширения ядра» («изнутри к границам»)

Подход к разработке программной системы на основе принципа «расширения ядра» или «изнутри к границам» представляет собой комбинацию двух предыдущих. Он предполагает первоначальное проектирование и разработку некоторой основы системы — её программного ядра, включающего компоненты, реализующие центральные, наиболее значимые функции. В дальнейшем ядро «наращивается» с использованием восходящего и нисходящего подходов. В соответствии с первым из них выполняется синтез новых, более крупных компонентов по принципу «изнутри–вверх» с целью построения целостной системы, а нисходящий подход используется для декомпозиции существующих компонентов ядра по принципу «изнутри–вниз» с целью их детальной реализации.

Стратегия «формирования ядра» («от границ внутрь»)

Стратегия проектирования и разработки на базе принципа «формирования ядра» или «от границ внутрь» также является комбинацией нисходящего и восходящего подходов. В этом случае в начале определяются и разрабатываются компоненты на самом верхнем и нижнем уровнях детализации системы. Затем выполняется декомпозиция компонентов на верхнем уровне и синтез компонентов на нижнем уровне, в результате чего формируется промежуточный уровень, состоящий из компонентов, образующих функциональное ядро программной системы.

1.16 Подходы к разработке программного обеспечения.

- функционально-модульный, или структурный.
- объектно-ориентированный

Подходы к разработке ПО

На сегодняшний день в инженерии программного обеспечения существует два основных подхода к разработке программных систем, различие между которыми обусловлено критериями декомпозиции:

1. Первый подход называют **функционально-модульным, или структурным**. В его основу положен принцип алгоритмической декомпозиции, когда выделяются функциональные элементы системы и устанавливается строгий порядок выполняемых действий.

2. Второй, **объектно-ориентированный подход** (Object-Oriented Analysis and Design, OOAD) использует объектную декомпозицию. При этом поведение системы описывается в терминах взаимодействия объектов.

Поддержка различных уровней принятия решений со стороны ИС и ИТ

Уровень принятия решений	Направленность принимаемых решений	Типичные приложения ИС	Типичные ИТ-решения
Стратегический	Стратегии, поддерживающие долгосрочные цели организации	Анализ рынка и сбыта, планирование разработки товаров, оценка эффективности	Добыча данных, управление знаниями
Тактический	Стратегии, поддерживающие краткосрочные задачи и распределение ресурсов	Анализ бюджета, прогнозирование фонда зарплаты, планирование запасов, обслуживание потребителей	Хранилища данных, анализ данных, электронные таблицы
Оперативный	Поддержка повседневной деятельности персонала и производство	Выплата жалованья, выписка счетов, бухгалтерский учет	Базы данных, обработка транзакций, генераторы приложений

ТЕМА 2. МОДЕЛИРОВАНИЕ БИЗНЕС-ПРОЦЕССОВ

2.1 Бизнес-моделирование

- Что такое бизнес-модель
- Цели моделирования. Участники бизнес-моделирования.
- Артефакты моделирования
- Что мы должны увидеть в модели
- Исполняемые и неисполнимые бизнес-процессы

Бизнес-модель определяется как формализованное (графическое, табличное, текстовое, символьное) описание бизнес-процессов, отражающее реально существующую или предполагаемую деятельность предприятия.

Цели моделирования. Участники бизнес-моделирования

Моделирование бизнес-процессов включает следующие цели:

- обеспечение понимания структуры организации и динамики происходящих в ней процессов;
- создание основы для формирования требований к программному обеспечению, автоматизирующему бизнес-процессы организации;
- обеспечение понимания текущих проблем организации и возможностей их решения;

- обеспечение единого восприятия заказчиками, пользователями и разработчиками целей и задач организации.

В моделировании бизнеса участвуют:

- бизнес-аналитик – специалист организации-разработчика, который возглавляет и координирует работы по моделированию бизнеса;
- бизнес-разработчик – специалист организации-разработчика, который детализирует и уточняет бизнес модели, определяет бизнес-исполнителей их обязанности и действия;
- заинтересованные лица – люди, предоставляющие информацию. Это все люди, заинтересованные как в собственно результатах моделирования, так и в будущей ПС.
- эксперт – представитель обследуемой организации, участвующий в разработке модели (консультации, организация встреч с заинтересованными лицами, оценка результатов).

Артефакты моделирования

При моделировании создаются следующие артефакты:

- Модели, описанных на IDEF, DFD, BPMN, UML ...
- Глоссарий – текстовый документ, содержащий определения основных понятий, используемых в данном бизнесе.
- Документ «Видение бизнеса» – определяет цели проведения бизнес-анализа.
- Структура организации – статическое описание подразделений организации и отношений подчиненности в виде диаграмм пакетов и/или классов.
- Оценка деятельности организации – текстовый документ, описывающий текущее состояние организации, в которой будет использоваться ПС.
- Бизнес-правила – текстовый документ, определяющий условия и ограничения, которым должен удовлетворять бизнес.
- Дополнительные спецификации – текстовый документ, содержащий описание свойств бизнеса, не включенных в бизнес-модель.

Что мы должны увидеть в модели

Модель бизнес-процесса должна определять:

- процедуры (функции, работы), которые необходимо выполнить для получения заданного конечного результата;

- последовательность выполнения процедур;
- входящие документы / информацию, используемые каждой процедурой процесса;
- исходящие документы / информацию, генерируемые процедурами процесса;
- механизмы контроля и управления в рамках рассматриваемого бизнес-процесса;
- субъекты выполнения процедур процесса;
- ресурсы, требующиеся для выполнения каждой процедуры процесса;
- документацию / условия, регламентирующие выполнение процедуры;
- параметры, характеризующие выполнение процедур и процесса в целом.



Исполняемые и неисполняемые бизнес-процессы

Исполняемые – те, которые действительно будут работать при помощи специального обеспечения, например, Bizagi.

Неисполняемые - бизнес-модели, необходимые только для изучения и демонстрации вариантов работы предприятия. В принципе, между их построением нет особой разницы, здесь важен исключительно желаемый результат.

Исполняемые бизнес-процессы обязательно должны быть выстроены в строгом соответствие всем правилам нотации BPMN, так как в противном случае программное обеспечение не сможет работать корректно с составленной бизнес-моделью. Исполняемые процессы нужны, например, на предприятиях, где принят процессный подход к деятельности

Неисполняемые бизнес-процессы нужны исключительно для демонстрации какой-либо бизнес-модели. (можно использовать любые удобные инструменты, в том числе, IDEF0)

2.2 Стадии бизнес-моделирования (AS-IS, TO-BE)

Состав стадий, которые включает в себя моделирование бизнес процессов:

1. Выявление процессов и построение исходной модели «как есть (AS-IS)». На этой стадии определяются границы процесса, выявляются его ключевые элементы, собираются данные о работе процесса. Эта модель не всегда адекватно отражает работу процесса, поэтому модель этой стадии можно назвать «первым драфтом» или исходной моделью «как есть».

2. Пересмотр, анализ и уточнение исходной модели. На этой стадии выявляются противоречия и дублирование действий в процессе, определяются ограничения процесса, взаимосвязи процесса, устанавливается необходимость изменения процесса. В результате формируется окончательный вариант модели «как есть».

3. Разработка модели «как должно быть (TO-BE)». После анализа существующей ситуации, необходимо определить желаемое состояние процесса. Это желаемое состояние представляется в модели «как должно быть». Такая модель показывает, как процесс должен выглядеть в будущем, включая все необходимые улучшения. В ходе этой стадии моделирования бизнес процессов и разрабатываются такие модели.

4. Тестирование и применение модели «как должно быть». Эта стадия моделирования связана с внедрением разработанной модели в практику деятельности организации. Модель бизнес процесса проходит апробацию, и в нее вносятся необходимые изменения.

2.3 Виды моделирования.

- Функциональное, объектное, имитационное
- Основные подходы при разработке бизнес-моделей
 - Функциональный;
 - Процессный;
 - Ментальный (с применением ментальных карт).
- Языки разработки бизнес-моделей VS Языки проектирования систем

Функциональное, объектное, имитационное

Наиболее часто, для целей совершенствования процесса применяют следующие виды моделирования:

• **Функциональное моделирование.** Этот вид моделирования подразумевает описание процессов в виде взаимосвязанных, четко структурированных функций. При этом строгая временная последовательность функций, в том виде, как она существует в реальных процессах, не обязательна.

• **Объектное моделирование** - подразумевает описание процессов, как набора взаимодействующих объектов – т.е. производственных единиц. Объектом является какой-либо предмет, преобразуемый в ходе выполнения процессов.

• **Имитационное моделирование** – при таком виде моделирования бизнес-процессов подразумевается моделирование поведения процессов в различных внешних и внутренних условиях с анализом динамических характеристик процессов и с анализом распределения ресурсов.

Для одного и того же процесса могут быть применены различные виды моделирования. Это позволяет работать с одним видом моделей независимо от других.

Основные подходы при разработке бизнес-моделей

Функциональный

Функциональное моделирование рассматривает бизнес как функцию или иными словами «черный ящик». В функциональной модели функция не имеет временной последовательности, а только точку входа и точку выхода. Функциональное моделирование помогает рассматривать бизнес-модель с точки зрения результативности, т.е. при моделировании мы исходим из того, что имеем на входе, и того, что желаем получить на выходе.

Вся работа при моделировании направлена на поиск оптимального решения для достижения цели.

Разделяя одну функцию на несколько, вы не получите описание процесса.

Функциональное моделирование оптимально реализовано в нотации IDEF0.

Процессный

Одним из наиболее распространенных стандартов процессного моделирования является BPMN

Процесс с точки зрения бизнес-модели — это последовательность каких-то событий и действий, которые имеют начало и конец.

В этом кроется основное отличие процессного моделирования от функционального.

Функциональное моделирование рассматривает бизнес-модель с точки зрения входа и выхода (имеющихся ресурсов и желаемого результата). А процессное основано на последовательности действий в определенных границах, в случае BPMN это будут начало и конец события.

Все процессы могут разбиваться (детализироваться) на подпроцессы вплоть до детализации на уровне задач, т.е. действий, дальнейшая детализация которых невозможна.

Процесс — это некая последовательность действий, которую необходимо выполнить, чтобы получить определенный результат. Необходимо отметить что в модели бизнеса как процесса результат может и не быть явным в отличии от функциональной модели. Принципиальное отличие процессного моделирования от функционального заключается в том, что при процессном моделировании основное внимание уделяется не тому, что мы хотим получить, а тому, что нужно сделать для получения результата, т.е. не итогам той или иной деятельности, а самой последовательности действий.

Например, в BPWin в процессе детализации каждой функции происходит переход от функционального подхода к процессному. Т.е. в общем, мы рассматриваем модель с точки зрения – возможностей и желаемого результата, а когда переходим к решениям для каждой функции, здесь уже практикуется явно процессный подход, т.е. пошаговый алгоритм действий для достижения результата.

Есть и еще одно очень важное отличие. Функциональную модель невозможно использовать при реализации какой-то либо системы, только для проектирования. А процессный подход позволяет создавать исполняемые модели, т.е. описания последовательности действий, которые мы можем в дальнейшем перевести в какую-то среду для создания системы совместной работы предприятия, основанной на процессном подходе.

Ментальный (с применением ментальных карт)

При создании ментальных моделей специалист подходит к моделированию не как к процессу или набору функций, а как к некому набору связанных между собой понятий. Такой вариант подхода применяется, прежде всего, для себя. Рисование схемы в свободной форме помогает структурировать свои знания, так сказать, “разложить по полочкам” в свободной форме полученную информацию. Также подобные ментальные карты помогают найти решение, которое уже позже, по мере необходимости, будет воплощаться в рамках строгих правил процессного или функционального подхода.

Можно применять ментальные карты и для демонстрации клиентам: и существующей ситуации, и вариантов решения поставленной задачи. Ментальные карты помогут наглядно продемонстрировать, какие методы могут быть использованы, показать в наглядной форме различные идеи.

Языки разработки бизнес-моделей VS Языки проектирования систем

1. Языки разработки бизнес-моделей
2. Языки проектирования систем
 - a. Семейство языков UML
 - b. Ares Studios

Основное различие в их предназначении. Если языки проектирования ИТ-систем рассматривают бизнес-процессы с точки зрения возможности их автоматизации, то языки бизнес-моделирования рассматривают последовательность действий именно с точки зрения бизнеса, включая работу как ИТ-систем, так и сотрудников, движения товаров и т.д.

Соответственно, в языках проектирования систем нет элементов, которые помогут полноценно описать действия подразделений, сотрудников, взаимодействие между ними, работу с поставщиками, общение с клиентами и так далее. Инструменты этой группы языков помогут именно автоматизировать процессы бизнеса, которые поддаются автоматизации.

В то же время языки разработки бизнес-процессов охватывают максимально именно работу бизнеса как такового, а вот те или иные нюансы автоматизации и алгоритмизации систем в них описать далеко не всегда возможно с достаточной степенью детализации.

2.4 Виды диаграмм

- Work Flow Diagram (диаграмма потока работ)
- Data Flow Diagram (диаграмма потока данных)
- Role Activity Diagram (диаграмма ролей)
- IDEF (Integrated Definition for Function Modeling)

- Цветные сети Петри
- Unified Modeling Language (UML)

Work Flow Diagram (диаграмма потока работ)

Графический метод представления процесса, в котором операции, данные, оборудование процесса и пр. изображаются специальными символами. Метод применяется для отображения логической последовательности действий процесса. Главным достоинством метода является его гибкость. Процесс может быть представлен множеством способов.

Диаграмму потоков работ целесообразно использовать для описания бизнес-процессов нижнего уровня, где возникает необходимость показывать временную последовательность выполнения работ в зависимости от получающихся результатов и событий, возникающих в ходе выполнения процесса. Здесь главным объектом описания становятся действия, а не потоки данных.

Важной отличительной чертой методологии WFD от DFD является наличие инструмента для отражения процесса с учетом временных характеристик и условий, влияющих на запуск той или иной операции. При этом если представлять процесс в DFDнотации, то порядок работ и движение данных должны совпадать с представлением процесса в нотации WFD. Поэтому стрелки между операциями бизнес-процесса обозначают не потоки данных, а временную последовательность выполнения работ.

На диаграмме потоков работ появляются элементы, с помощью которых описывается процесс: логические операторы, события начала и окончания процесса, а также элементы, показывающие временные задержки.

Data Flow Diagram (диаграмма потока данных)

Диаграмма потока данных или DFD применяется для отображения передачи информации (данных) от одной операции процесса к другой. DFD описывает взаимосвязь операций за счет информации и данных. Этот метод является основой структурного анализа процессов, т.к. позволяет разложить процесс на логические уровни.

Применение DFD позволяет отразить только поток информации, но не поток материалов. Диаграмма потока данных показывает, как информация входит и выходит из процесса, какие действия изменяют информацию, где информация хранится в процессе и пр.

Таким образом, описать бизнес-процесс можно с помощью набора DFD- и WFD-диаграмм, где верхний уровень будет представлен в виде модели потоков данных, а нижний — в виде иерархически взаимосвязанных и упорядоченных моделей потоков работ. Стоит отметить, что степень декомпозиции зависит от целей и задач описания бизнес-процесса.

Role Activity Diagram (диаграмма ролей)

применяется для моделирования процесса с точки зрения отдельных ролей, групп ролей и взаимодействия ролей в процессе. Роль представляет собой абстрактный элемент процесса, выполняющий какую-либо организационную функцию. Диаграмма ролей показывает степень «ответственности» за процесс и его операции, а также взаимодействие ролей.

IDEF (Integrated Definition for Function Modeling)

представляет собой целый набор методов для описания различных аспектов бизнес-процессов (IDEF0, IDEF1, IDEF1X, IDEF2, IDEF3, IDEF4, IDEF5). Эти методы строятся на базе методологии структурного анализа SADT (Structured Analysis and Design Technique). Для моделирования бизнес процессов наиболее часто применяют методы **IDEF0** и **IDEF3**.

Цветные сети Петри

метод представляет модель процесса в виде графа, где вершинами являются действия процесса, а дугами события, за счет которых осуществляется переход процесса из одного состояния в другое. Сети Петри применяют для динамического моделирования поведения процесса.

Unified Modeling Language (UML)

объектно-ориентированный метод моделирования процессов. Он состоит из 9-ти различных диаграмм, каждая из которых позволяет моделировать отдельные статические или динамические аспекты процесса.

2.5 Бизнес-процесс

- Понятие бизнес-процесса
 - Универсальная структурная схема процесса
 - Отличительные черты процесса (хозяин, ресурсы, параметры, ...)
 - Декомпозиция процесса. Критерии декомпозиции.
 - Основные и вспомогательные процессы. Типичные основные процессы.

Бизнес-процесс – это логичный, последовательный, взаимосвязанный набор мероприятий, который потребляет ресурсы, создаёт ценность и выдаёт результат. В международном стандарте ISO 9000:2000 принят термин "процесс", однако в настоящее время эти термины можно считать синонимами.

Моделирование бизнес-процессов – это эффективное средство поиска путей оптимизации деятельности компании, позволяющее определить, как компания работает в целом и как организована деятельность на каждом рабочем месте.

Под методологией создания модели бизнес-процесса понимается совокупность способов, при помощи которых объекты реального мира и связи между ними представляются в виде модели. Для каждого объекта и связей характерны ряд параметров, или атрибутов, отражающих определённые характеристики реального объекта (номер объекта, название, описание, длительность выполнения (для функций), стоимость и др.).

Бизнес-процесс – это поток работы, переходящий от одного человека к другому, а для больших процессов, вероятно, от одного отдела к другому.

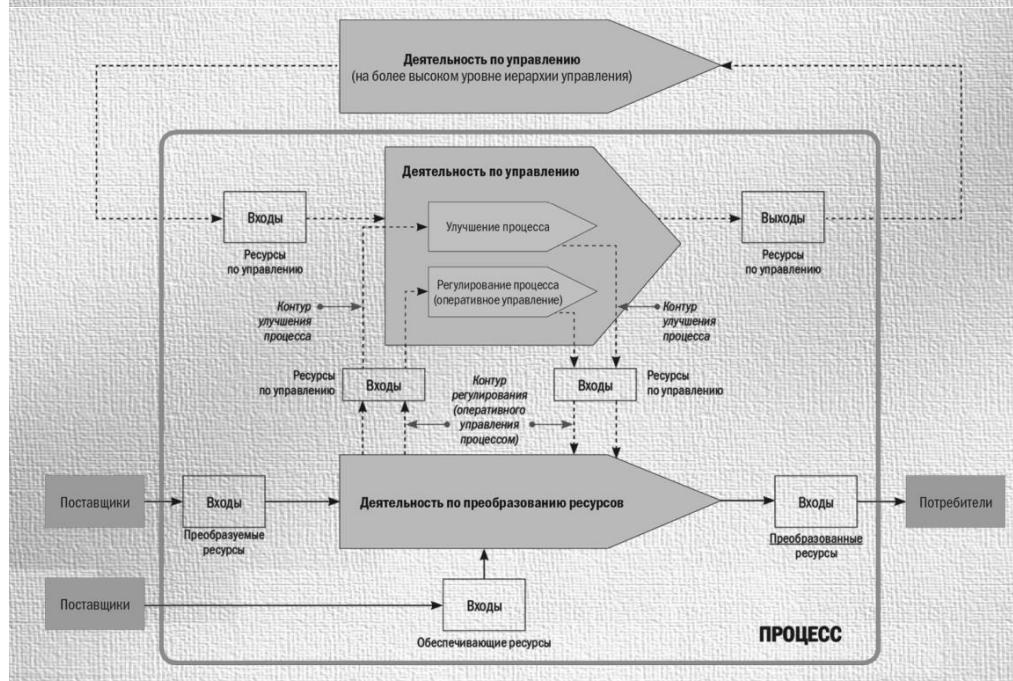
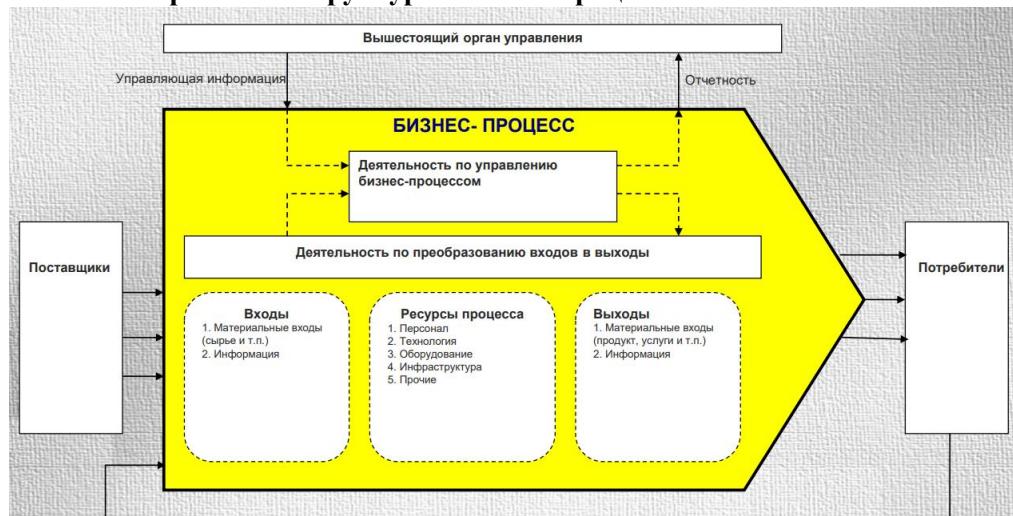
Бизнес-процесс (процесс) – устойчивая, целенаправленная совокупность взаимосвязанных видов деятельности, которая по определенной технологии преобразует входы в выходы, представляющие ценность для клиента.

Процесс – набор взаимосвязанных или взаимодействующих действий, преобразующих входы в выходы. Процессы состоят из **действий** (activities), а действия – из **задач** (tasks).

ПРОЦЕСС ► ДЕЙСТВИЯ ► ЗАДАЧИ

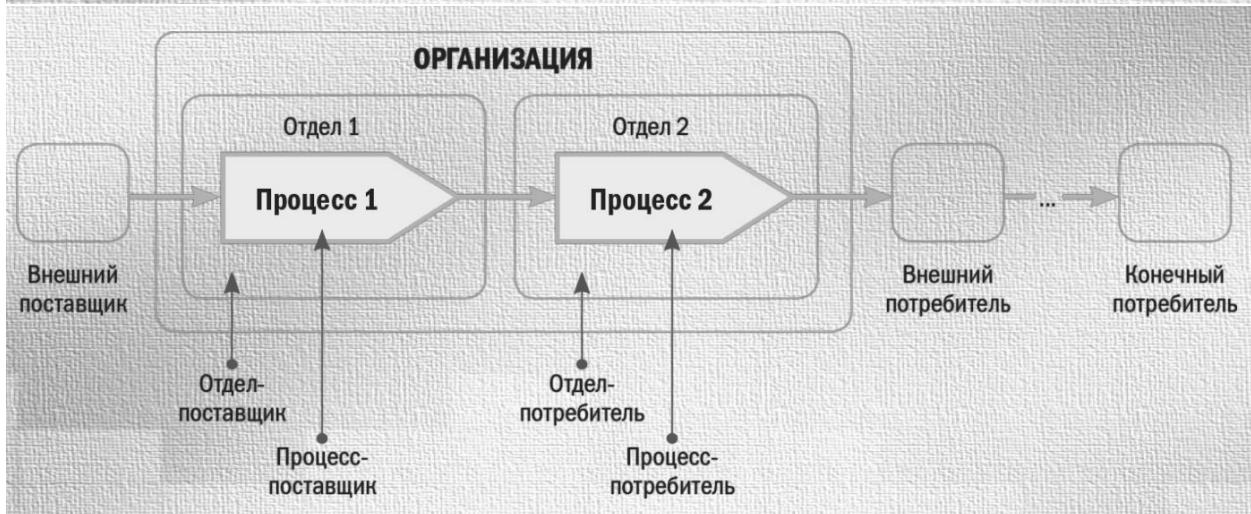
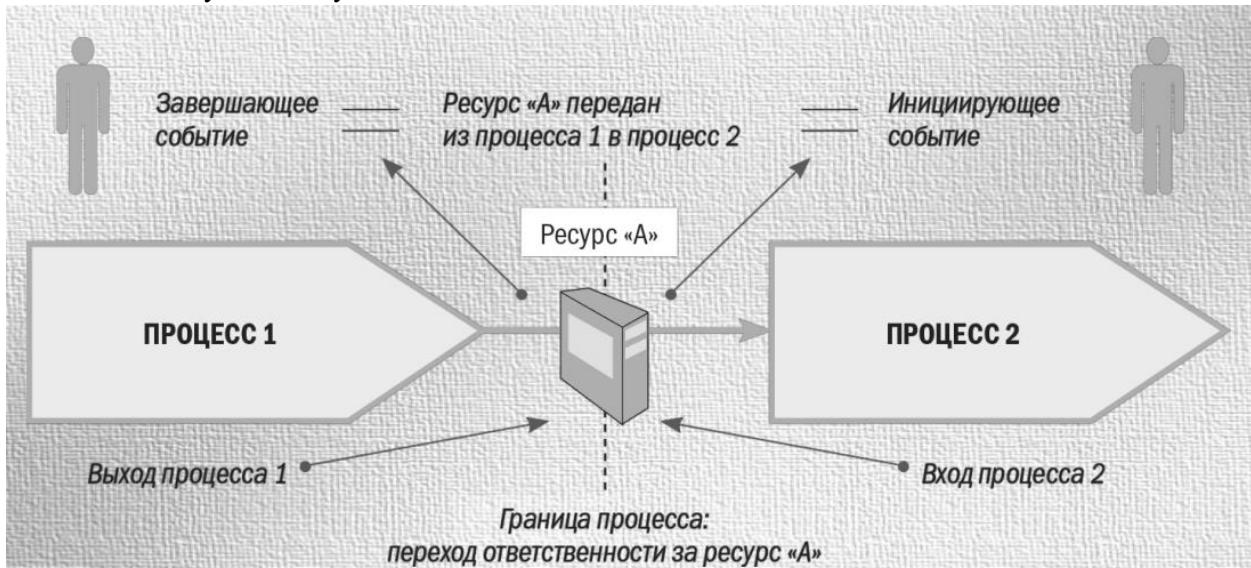
Работа не движется вверх и вниз вдоль функциональной иерархии, она течет сквозь организацию в виде набора бизнес-процессов, которые в большинстве организаций никем не управляются, и никто за них не отвечает!

Универсальная структурная схема процесса

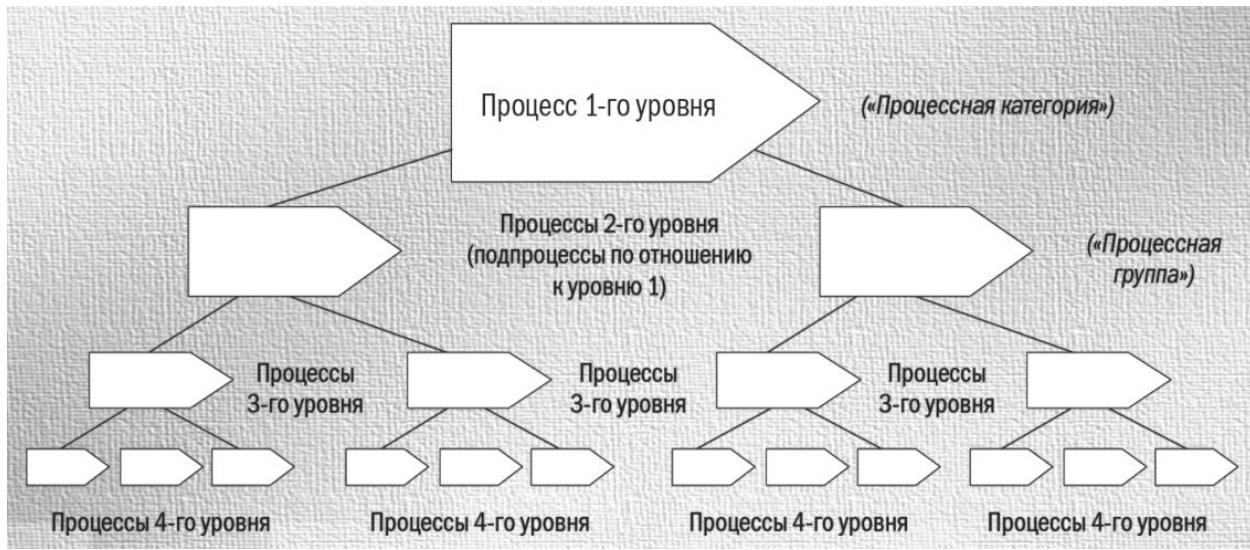


Отличительные черты процесса (хозяин, ресурсы, параметры, ...)

- **Хозяин Процесса** – должностное лицо, несущее ответственность за ход и результаты Процесса;
- **Ресурсы** – ресурсы, выделенные в распоряжение Хозяина Процесса для его проведения; могут включать – оборудование (производственное, контрольно-измерительное, офисное и др.), персонал, помещения, среду, транспорт, связь, материалы (вспомогательные), финансы, документация и т. д.;
- **Параметры Процесса** – характеристики (информация) по которым Хозяин Процесса и высший руководитель могут судить о том, насколько эффективно выполняется Процесс и достигаются ли запланированные результаты;
- **Потребитель** – потребитель результатов Процесса, степень удовлетворенности которого, также предназначена для оценки эффективности Процесса;
- **Входы Процесса** – входные объекты (сырье, продукция, комплектация, информация или услуга), которые преобразуются в Выходы Процесса, в ходе выполнения Процесса. Часто Входы одного Процесса являются выходами другого;
- **Выходы Процесса** – продукция, информация или услуга ради которой существует Процесс
- **Сеть Процессов организации** – объединение взаимосвязанных и взаимосогласованных Процессов организации в единую систему.



Декомпозиция процесса. Критерии декомпозиции.



Декомпозиция - это разбиение объекта на составные части.

Критерий декомпозиции - это характеристика, на основе которой производится разбиение.

Правило 1. На одном уровне нужно применять только один критерий декомпозиции.

Правило 2. Для одной системы можно построить несколько вариантов "деревьев" в зависимости от различной последовательности применения возможных критериев декомпозиции. При этом на верхнем уровне нужно использовать более существенные критерии декомпозиции

Основные и вспомогательные процессы. Типичные основные процессы

Основные процессы – это процессы текущей деятельности компании, результатом которых является производство выходов, требуемых внешним клиентом.

Типичные основные процессы:

- Материально-техническое обеспечение компании: обеспечение материалами и оборудованием для производства продуктов или формирования услуг
- Производственные процессы: создание продуктов или услуг
- Материально-техническое обеспечение сбыта
- Маркетинг и продажа
- Обслуживание: обеспечение поддержки клиентов после приобретения ими Ваших продуктов или услуг

Правила:

1. Данный перечень процессов не является обязательным и полным, названия процессов в конкретных организациях могут не совпадать с приведенными на данном рисунке, процессы можно объединять и исключать в зависимости от целей и особенностей конкретной организации. В данном случае важна суть дела: через Основные Процессы проходит производимая продукция и/или услуги и их компоненты (маркетинг, проект, входящие материалы и др.).

2. Основных процессов должно быть не более чем 7 ± 2 . Эта цифра исходит из того, что высший руководитель, как любой человек, не может эффективно руководить и воспринимать информацию от большего количества основных направлений деятельности.

3. При выделении Процессов необходимо назначать лиц, ответственных за их результативность (Хозяев процессов). Каждый Процесс должен иметь ТОЛЬКО ОДНОГО ХОЗЯИНА.

4. Чтобы Хозяин мог влиять на ход Процесса и его результаты, ему должны быть выделены все необходимые ресурсы и полномочия и установлены Показатели эффективности Процесса, адекватно отражающие ход Процесса. По этим показателям (в том числе экономическим) Хозяин должен регулярно отчитываться перед высшим руководителем о результатах своей деятельности.

Вспомогательные процессы обеспечивают существование первичных процессов.

Примеры вспомогательных процессов:

- Управление и обучение персонала
- Управление информацией
- Управление энергоресурсами
- Управление финансами
- Управление природными ресурсами и экологией
- Административно-хозяйственная деятельность

Правила:

1. Вспомогательные процессы, в отличие от основных, не добавляют ценности продукции, но повышают ее себестоимость.

2. Количество вспомогательных процессов не должно быть более чем 5±2. В ином случае руководитель теряет управление организацией.

Вспомогательные процессы могут быть одинаковыми для нескольких бизнес-процессов.

Как основные, так и вспомогательные процессы, в свою очередь, состоят из совокупности подпроцессов (операций, функций).

2.6 Управление бизнес-процессами

- Оптимизация организационной структуры.
- Как управлять процессами.
 - кросс-функциональное решение проблем
 - внедрение процессной оргструктуры (реинжиниринг)
 - описание и управление процессами

Оптимизация организационной структуры

Организационная структура управления – это подразделения и должности в организации, выделенные в результате распределения задач, обязанностей, полномочий и ответственности, и взаимоотношения между ними в процессе осуществления деятельности

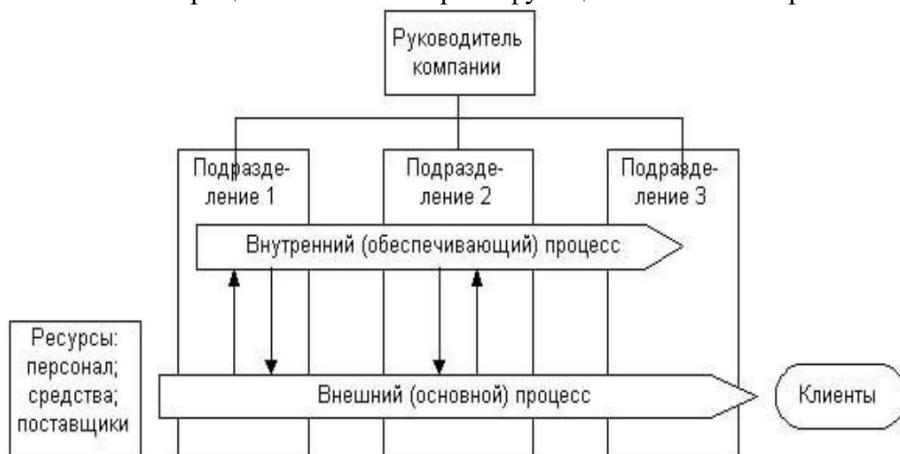
Подразделение - структурная единица, входящая в состав организации, представленная совокупностью людей, объединенных общностью деятельности в специальной области работы

Организационная структура должна разрабатываться для того, чтобы соответствовать определенному набору целей. Структура, опирающаяся на бизнес-процессы, позволяет избежать подмены общей цели организации – **удовлетворение клиентов** – целями функциональных подразделений, чувствующих себя самодостаточными структурами.

Кросс-функциональное решение проблем

Использование бизнес-процессов для идентификации нужных проблем, над которыми будут работать кросс-функциональные группы.

Бизнес-процессы являются кросс-функциональными и пронизывают организационную структуру:



Внедрение процессной оргструктуры (реинжиниринг)

Изменение организационной структуры для того, чтобы она отражала бизнес-процессы. Может иметь долгосрочные и значительные последствия.

Описание и управление процессами

Концепция управления бизнес-процессами отличается от реинжиниринга тем, что предлагает начинать управлять бизнес-процессами с имеющегося состояния, непрерывно улучшая их. Основополагающий принцип процессного управления и BPM (Business Process Management) заключается в том, что во главу угла ставится общая цель, достижение которой является смыслом деятельности.

Результат работы оценивается не по тому, как быстро каждый исполнитель выполнил свои обязанности, а по тому, как завершился весь процесс.

2.7 Бизнес-стратегии

- Проблемы функционального управления.
- Реинжиниринг бизнес-процессов.
- Процессное управление.

Проблемы функционального управления

1. Компания находится в начале своего развития: она недавно вышла на рынок и только начинает его освоение. Обычно на этой стадии количество сотрудников организации невелико (до 30 человек), организационная структура не слишком формализована, уровней иерархии не больше 3, в центре внимания руководства чаще всего находится производство и продажа Продукта/Услуги.

- отсутствие четко обозначенной стратегии дальнейшего развития (к чему и зачем идем?)
- неопределенность в разграничении обязанностей и ответственности каждого сотрудника и целых подразделений, снижающая качество выполнения работ и вызывающая внутренние конфликты
- возникновение серьезных проблем при смене работников даже на нижнем уровне компании, поскольку нет механизмов передачи знаний и навыков новым сотрудникам, из-за чего им приходится учиться на своих ошибках.
- невозможность дальнейшего развития компании без привлечения дополнительных специалистов, обладающих специальными знаниями и навыками.
- разработка регламентов и поддержка их актуальности происходит вручную

2. Компания прошла начальный этап и активно растет, в ней появляется все больше функциональных подразделений и уровней управления, количество сотрудников таково, что высшее руководство уже не знакомо с каждым лично.

○ по мере роста компании эффективность функционально-иерархической системы управления неизбежно снижается.

○ Пока уровней иерархии в подразделениях немного, взаимодействие организуется достаточно оперативно, но вот с ростом организации время, затрачиваемое на принятие решений, превышает все разумные пределы. Результатом этого является перенос ВСЕХ решений на самый высокий уровень и глобальное снижение управляемости.

Реинжиниринг бизнес-процессов.

фундаментальное переосмысление и радикальное перепроектирование бизнес-процессов для достижения максимальной эффективности производственно-хозяйственной и финансово-экономической деятельности, оформленное соответствующими организационно-распорядительными и нормативными документами. Бизнес-инжиниринг состоит из моделирования бизнес-процессов (разработка модели "как есть", её анализ, разработка модели "как надо") и разработки и реализации плана перехода к состоянию "как надо"

Смысл реинжиниринга бизнес-процессов в двух его основных этапах:

1. определение оптимального (идеального) вида бизнес-процесса (в первую очередь основного);
2. определение наилучшего (по средствам, времени, ресурсам и т. п.) способа перевода существующего бизнес-процесса в оптимальный.

Организации проводят реинжиниринг двух ключевых областей своего бизнеса.

1. Во-первых, они используют современные технологии для улучшения процессов распространения данных и принятия решений.
2. Затем они изменяют функциональные организации, чтобы сформировать функциональные команды.

Порядок проведения

1. Разработка корпоративной стратегии
2. Определение ключевых компетенций, которые необходимы для внедрения стратегии; 3.
3. Подробный анализ существующих процессов
4. Выявление процессов, требующих изменения
5. Определение ключевых показателей эффективности для бизнеспроцессов
6. Собственно реинжиниринг
7. Контроль и постоянное совершенствование новых процессов на основе ключевых показателей эффективности.

Процессное управление

Решением проблем, присущих функциональному управлению, является переход к процессному управлению: всю деятельность, безотносительно того, к какому функциональному признаку она относится, группируют в смешанные подразделения, где каждый исполнитель отвечает за свой блок операций.

Принципиальным различием этих подходов является переход от управления ФУНКЦИОНИРОВАНИЕМ организаций (и ее структурных подразделений, объединенных на основе ПРЕДМЕТА деятельности: бухгалтерия, снабжение, сбыт и т.п.) к управлению БИЗНЕС-ПРОЦЕССАМИ на основе РЕЗУЛЬТАТОВ деятельности.

- фокус внимания смещается в сторону эффективности работы организации

- все возможные ситуации при выполнении бизнес-процессов максимально подробно описываются
- для типовых ситуаций создаётся подробный регламент деятельности

В этом случае персонал в типовых ситуациях может действовать максимально эффективным образом и, что особенно важно, самостоятельно, т.е., без участия руководителя

Различие подходов BPR и BPMS

<i>Традиционный реинжиниринг</i>	<i>Автоматизация бизнес-процессов при помощи BPMS</i>
Тотальное моделирование всех бизнес-процессов, проектирование сверху-вниз	Первоочередное внимание критичным для бизнеса процессам, объектное проектирование
Длительный цикл разработки, оптимизация «большим скачком»	Короткий цикл разработки, поэтапная оптимизация
Схема бизнес-процесса актуальна только в момент сдачи работы внешними консультантами	Постоянное поддержание схем бизнес-процессов в актуальном состоянии
Средства автоматизации только для анализа бизнес-процессов	Средства автоматизации для анализа, исполнения и контроля бизнес-процессов

Для внедрения процессного подхода в компании необходимо:

- провести построение, анализ и оптимизацию цепочек создания ценности (т.е. продуктов и услуг, представляющих ценность для потребителя) организации на самом верхнем уровне;
- создать систему (сеть) процессов организации с учетом ее целей и стратегий;
- на основе системы процессов оптимизировать организационную структуру организации;
- выполнить регламентацию деятельности в той степени, которая необходима для организации управления процессами;
- разработать и внедрить систему показателей для управления компанией и процессами;
- организовывать управление процессами;
- организовывать деятельность руководителей по улучшению процессов;
- разработать и внедрить систему мотивации, ориентированную на улучшение процессов.

2.8 Методология описания бизнес-процессов

- Структуризация и описание организации
 - Основные бизнес-процессы
 - Обеспечивающие бизнес-процессы
 - Бизнес-процессы управления
 - Бизнес-процессы развития

Основные бизнес-процессы – генерируют доходы компании определяют профиль бизнеса; имеют стратегическое значение; нельзя отдавать на аутсорсинг

- Процессы, создающие добавленную стоимость продукту, который производит компания.
- Процессы, создающие продукт, представляющий ценность для внешнего клиента.
- Процессы, прямой целью которых является получение доходов.
- Процессы, за которые внешний клиент готов платить деньги.

Характеристики основных бизнес-процессов

Определения	Отличительные особенности
Бизнес-процессы, которые создают добавленную стоимость продукта, предлагаемого компанией;	Представляют "зеркальное отражение" бизнес - направлений деятельности;
Бизнес-процессы, которые создают продукт представляющий ценность для внешнего клиента;	Являются источником генерирования доходов;
Бизнес-процессы, прямой целью которых является генерирование доходов;	Определяют профиль бизнеса;
Бизнес-процессы, за которые внешний клиент готов платить деньги.	Имеют стратегическое значение;
	Могут развиваться или отмирать в зависимости от востребованности рынка и стратегии компании.

Обеспечивающие бизнес-процессы – поддерживают инфраструктуру компании обеспечивающие процессы поддерживают инфраструктуру организации. Клиент за них не готов платить деньги, но эти процессы нужны для того, чтобы компания существовала. Клиентами обеспечивающих процессов являются подразделения и сотрудники организации, которых при описании процессов называют внутренними клиентами.

- административно-хозяйственное обеспечение,
- обеспечение безопасности,
- юридическое обеспечение и т.д.

Характеристики обеспечивающих бизнес-процессов

Определения	Отличительные особенности
Бизнес-процессы, клиентами которых являются основные бизнес-процессы;	Выходы могут продаваться на внешнем рынке;
Бизнес-процессы, которые поддерживают инфраструктуру организации.	Не имеют стратегического значения ; Могут превратиться в основной бизнес-процесс; Могут отмереть в случае наличия конкурентоспособных альтернатив на внешнем рынке и передачи их исполнения на аутсорсинг.

Бизнес-процессы управления – управляют компанией

Процессы управления являются тоже обеспечивающими. Они нужны для менеджмента компании, потому что именно эти процессы позволяют управлять компанией, обеспечивая ее выживание, конкурентоспособность и развитие.

Определения	Отличительные особенности
Бизнес-процессы, которые обеспечивают выживание, конкурентоспособность и развитие организации, регулируют ее текущую деятельность;	Имеют типовую внутреннюю структуру: <ul style="list-style-type: none">• Планирование;• Организация;• Учет;• Контроль;• Регулирование.
Бизнес-процессы, прямой целью которых является управление деятельностью организации.	Различие между процессами управления определяется спецификой объектов управления, которыми процесс управляет <ul style="list-style-type: none">• Стратегия;• Деньги;• Персонал;• Потребитель;• Товарный запас;• Активы и т.д.

Типовая структура бизнес-процессов управления представляется стандартной цепочкой управленческого цикла, который состоит из следующих этапов:

1. "Планирование". На данной этапе собирается информация, проводится ее анализ и разрабатывается план действий.

2. "Организация". После разработки плана нужно обеспечить его реализацию - довести мероприятия до сотрудников, обеспечить сотрудников необходимыми для реализации плана ресурсами.

3. "Учет". По истечении установленного периода нужно собрать фактическую информацию о выполнении запланированных работ и достигнутых результатов.

4. "Контроль". После проведения учета план сопоставляется с фактической информацией и проводится анализ план-фактных отклонений.

5. "Регулирование". На последнем пятом этапе принимается решение о последующих действиях - корректировки плана, поощрении или наказании сотрудников, которые эти планы реализовывали.

Ряд "необходимых" бизнес-процессов управления, которые имеются в любой компании

- Стратегическое управление
- Управление финансами
- Управление маркетингом
- Управление персоналом.

Бизнес-процессы развития – развивают компанию.

- Бизнес-процессы, целью которых является получение прибыли в долгосрочной перспективе
- Бизнес-процессы совершенствования и развития деятельности организации.

Определения	Отличительные особенности
Бизнес-процессы целью которых является получение прибыли в долгосрочной перспективе;	На 80% представляют из себя проекты – процессы, которые выполняются один раз.
Бизнес-процессы целью которых является совершенствование и развитие деятельности организаций.	Требуют иных техник управления, которые называют технологиями управления проектами; Предъявляют иные требования к проектному менеджеру в отличие от требований к менеджеру операционному.

Отличительной особенностью бизнес-процессов развития является то, что они на 80% представляют из себя проектную деятельность.

2.9 Функциональное моделирование

- SADT – технология структурного анализа и проектирования ИС
- DFD - диаграммы потоков данных
 - Методы задания спецификации процессов
 - Структурированный естественный язык
 - Таблицы и деревья решений
 - **Визуальные языки проектирования спецификаций**. Диаграмма Насси-Шнейдермана
- IDEF3 – метод описания процессов.

SADT – технология структурного анализа и проектирования ИС

Методология IDEF-SADT представляет собой совокупность методов, правил и процедур, предназначенных для построения функциональной модели

Функциональная модель SADT отображает структуру процессов функционирования системы и ее отдельных подсистем, т. е. выполняемые ими действия и связи между этими действиями

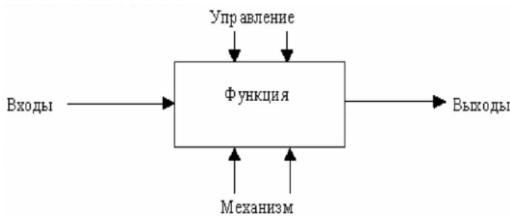
Для этой цели строятся специальные модели, которые позволяют в наглядной форме представить **последовательность** определенных действий

Исходными строительными блоками любой модели IDEF0 процесса являются **деятельность** (activity) и **стрелки** (arrows).

Деятельность представляет собой некоторое действие или набор действий, которые имеют фиксированную цель и приводят к некоторому конечному результату. Иногда деятельность называют просто **процессом**.

Стрелка служит для обозначения некоторого носителя или воздействия, которые обеспечивают перенос данных или объектов от одной деятельности к другой. Стрелки также необходимы для описания того, что именно производит деятельность и какие ресурсы она потребляет.

- I (Input) — вход, т. е. все, что поступает в процесс или потребляется процессом.
- C (Control) — управление или ограничения на выполнение операций процесса.
- O (Output) — выход или результат процесса.
- M (Mechanism) — механизм, который используется для выполнения процесса.



Под субъектом понимается сама система, при этом необходимо точно установить, что входит в систему, а что лежит за ее пределами, другими словами, мы должны определить, что мы будем в дальнейшем рассматривать как компоненты системы, а что как внешнее воздействие.

Широта подразумевает определение границ модели – мы определяем, что будет рассматриваться внутри системы, а что снаружи. **Глубина** определяет, на каком уровне детализации модель является завершенной.

Модель не может быть построена без четко сформулированной цели. Цель должна отвечать на следующие вопросы:

- Почему этот процесс должен быть смоделирован?
- Что должна показывать модель?
- Что может получить читатель?

IDEF0 - модель предполагает наличие четко сформулированной цели, единственного субъекта моделирования и одной точки зрения.

AS - IS и TO - BE

Сначала строится модель существующей организации работы - **AS-IS** (как есть). На основе модели AS-IS достигается консенсус между различными единицами бизнеса по тому, «кто что сделал» и что каждая единица бизнеса добавляет в процесс. Модель AS-IS позволяет выяснить, «что мы делаем сегодня» перед тем, как перепрыгнуть на то, «что мы будем делать завтра».

Найденные в модели AS-IS недостатки можно исправить при создании модели **TO-BE** (как будет) - модели новой организации бизнес-процессов. Модель нужна TO-BE для анализа альтернативных (лучших) путей выполнения работы и документирования того, как компания будет делать бизнес в будущем.

SADT-Модель может содержать четыре типа диаграмм:

1. контекстную диаграмму (в каждой модели может быть только одна контекстная диаграмма);
2. диаграммы декомпозиции;
3. диаграммы дерева узлов;
4. диаграммы только для экспозиции (FEO).

Контекстная диаграмма является вершиной древовидной структуры диаграмм и представляет собой самое общее описание системы и ее взаимодействия с внешней средой.

После описания системы в целом проводится разбиение ее на крупные фрагменты. Этот процесс называется функциональной декомпозицией, а диаграммы, которые описывают каждый фрагмент и взаимодействие фрагментов, называются **диаграммами декомпозиции**.

После каждого сеанса декомпозиции проводятся сеансы экспертизы – эксперты предметной области указывают на соответствие реальных бизнес-процессов созданным диаграммам.

Диаграмма дерева узлов показывает иерархическую зависимость работ, но не взаимосвязи между работами.

Диаграммы для экспозиции (FEO) строятся для иллюстрации отдельных фрагментов модели для иллюстрации альтернативной точки зрения, либо для специальных целей.

Диаграммы состоят из блоков – **работ** (activity). Работы обозначают поименованные процессы, функции или задачи, которые происходят в течение определенного времени и имеют распознаваемые результаты. Работы изображаются в виде прямоугольников. Все работы должны быть названы и определены., «Имя работы должно быть выражено глаголом или отглагольным существительным, обозначающим действие (например, «Изготовление деталиПрием заказа» и т. д.).

В IDEF0 различают пять типов связей работ.

1. Связь по входу (output-input), когда стрелка выхода вышестоящей работы (далее – просто выход) направляется на вход нижестоящей

2. Связь по управлению (output-control), когда выход вышестоящей работы направляется на управление нижестоящей. Связь по управлению показывает доминирование вышестоящей работы. Данные или объекты выхода вышестоящей работы не меняются в нижестоящей.

3. Обратная связь по входу (output-input feedback), когда выход нижестоящей работы направляется на вход вышестоящей. Такая связь, как правило, используется для описания циклов.

4. Обратная связь по управлению (output-control feedback), когда выход нижестоящей работы направляется на управление вышестоящей (стрелка «Рекомендации»). Обратная связь по управлению часто свидетельствует об эффективности бизнес-процесса.

5. Связь выход-механизм (output-mechanism), когда выход одной работы направляется на механизм другой. Эта взаимосвязь используется реже остальных и показывает, что одна работа подготавливает ресурсы, необходимые для проведения другой работы.

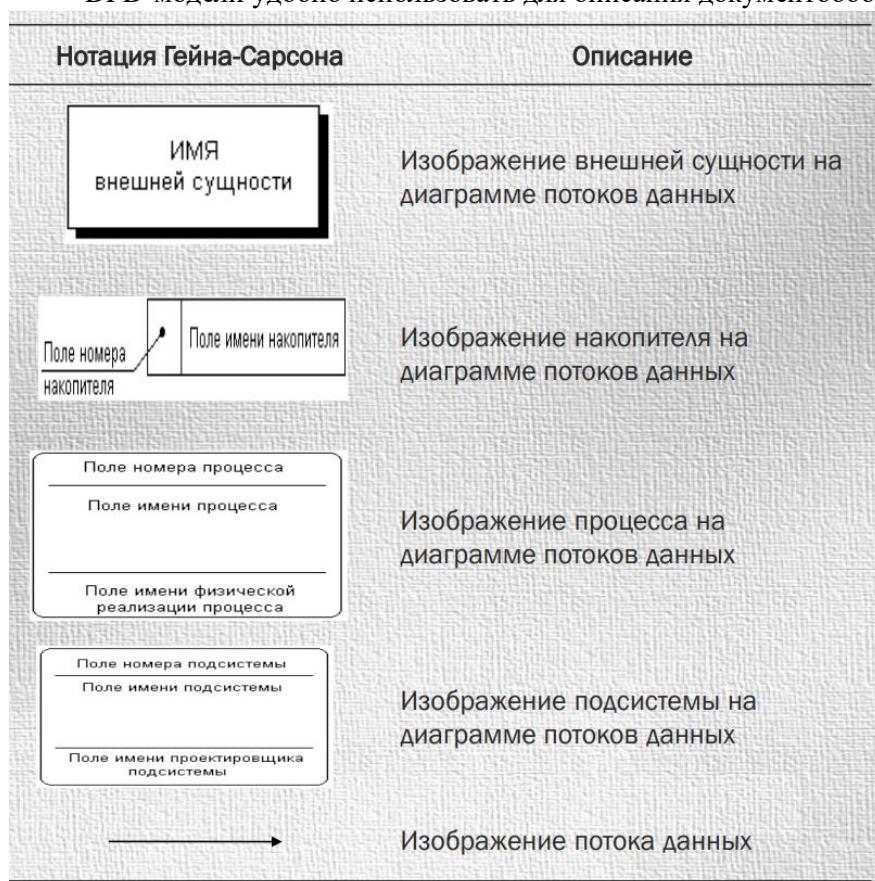
Тоннелирование стрелок. Вновь внесенные граничные стрелки на диаграмме декомпозиции нижнего уровня изображаются в квадратных скобках и автоматически не появляются на диаграмме верхнего уровня.

Тоннелирование может быть применено для изображения малозначимых стрелок. Или может быть ситуация, когда стрелка механизма мигрирует с верхнего уровня на нижний, причем на нижнем уровне этот механизм используется одинаково во всех работах без исключения.

DFD - диаграммы потоков данных

С помощью диаграмм потоков данных функциональные требования проектируемой системы разбиваются на функциональные компоненты (процессы) и представляются в виде сети, связанной потоками данных. Главная цель таких средств – продемонстрировать, как каждый процесс преобразует свои входные данные в выходные, а также выявить отношения между этими процессами.

DFD-модели удобно использовать для описания документооборота и обработки информации.



Процесс (process) – процесс представляет собой преобразование входных потоков данных в выходные в соответствии с определенным алгоритмом.

Номер процесса служит для его идентификации. В поле имени вводится наименование процесса в виде предложения с активным недвусмысленным глаголом в неопределенной форме (вычислить, рассчитать, проверить, определить, создать, получить), за которым следуют существительные в винительном падеже ("Ввести сведения о клиентах", "Выдать информацию о текущих расходах" и т.д.)

Системы и подсистемы. При построении модели сложной ИС она может быть представлена в самом общем виде на так называемой контекстной диаграмме в виде одной системы как единого целого, либо может быть декомпозирована на ряд подсистем. Номер подсистемы служит для ее идентификации. В поле имени вводится наименование подсистемы в виде предложения с подлежащим и соответствующими определениями и дополнениями

Внешний объект (external entity) – внешняя сущность представляет собой материальный предмет или физическое лицо, представляющее собой источник или приемник информации, например, заказчики, персонал, поставщики, клиенты, склад.

Накопитель (store) – накопитель данных представляет собой абстрактное устройство для хранения информации, которую можно в любой момент поместить в накопитель и через некоторое время извлечь, причем способы помещения и извлечения могут быть любыми. Накопитель данных идентифицируется буквой "D" и произвольным числом.

Поток (flow) – поток данных определяет информацию, передаваемую через некоторое соединение от источника к приемнику. Поток данных на диаграмме изображается линией, оканчивающейся стрелкой, которая показывает направление потока. Каждый поток данных имеет имя, отражающее его содержание.

Контекстная диаграмма отражает интерфейс системы с внешним миром, а именно, информационные потоки между системой и внешними сущностями, с которыми она должна быть связана.

Пример:



Главная цель построения иерархического множества DFD заключается в том, чтобы сделать требования ясными и понятными на каждом уровне детализации, а также разбить эти требования на части с точно определенными отношениями между ними.

- Размещать на каждой диаграмме от 3 до 6-7 процессов.
- Не загромождать диаграммы несущественными на данном уровне деталями
- Декомпозицию потоков данных осуществлять параллельно с декомпозицией процессов
- Выбирать ясные, отражающие суть дела, имена процессов и потоков для улучшения понимаемости диаграмм, при этом стараться не использовать аббревиатуры.
- Однократно определять функционально идентичные процессы на самом верхнем уровне, где такой процесс необходим, и ссылаться на него на нижних уровнях.
- Отделять управляющие структуры от обрабатывающих структур (т.е. процессов), локализовать управляющие структуры.

Структурированный естественный язык

Структурированный естественный язык применяется для читабельного, строгого описания спецификаций процессов. Он является разумной комбинацией строгости языка программирования и читабельности естественного языка и состоит из подмножества слов, организованных в определенные логические структуры, арифметических выражений и диаграмм.

Управляющие структуры языка имеют один вход и один выход. К ним относятся:

1. последовательная конструкция:

ВЫПОЛНИТЬ функция1

ВЫПОЛНИТЬ функция2

ВЫПОЛНИТЬ функция3

2. конструкция выбора:

ЕСЛИ <условие> **ТО**

ВЫПОЛНИТЬ функция1

ИНАЧЕ

ВЫПОЛНИТЬ функция2

КОНЕЦСЛИ

3. итерация:

ДЛЯ <условие>

ВЫПОЛНИТЬ функция

КОНЕЦДЛЯ

или

ПОКА ВЫПОЛНИТЬ функция КОНЕЦПОКА

Таблицы и деревья решений

Если действие зависит от нескольких переменных, которые в совокупности могут продуцировать большое число комбинаций, то его описание будет слишком запутанным и с большим числом уровней вложенности. Для описания подобных действий традиционно используются таблицы и деревья решений.

Проектирование спецификаций процессов с помощью **таблиц решений** (TR) заключается в задании матрицы, отображающей множество входных **условий** в множество **действий**.

Верхняя часть таблицы используется для определения условий. Обычно условие является ЕСЛИ-частью оператора ЕСЛИ-ТО и требует ответа "данет" (Д, Н).

Нижняя часть TR используется для определения действий, т.е. ТО-части оператора ЕСЛИ-ТО.

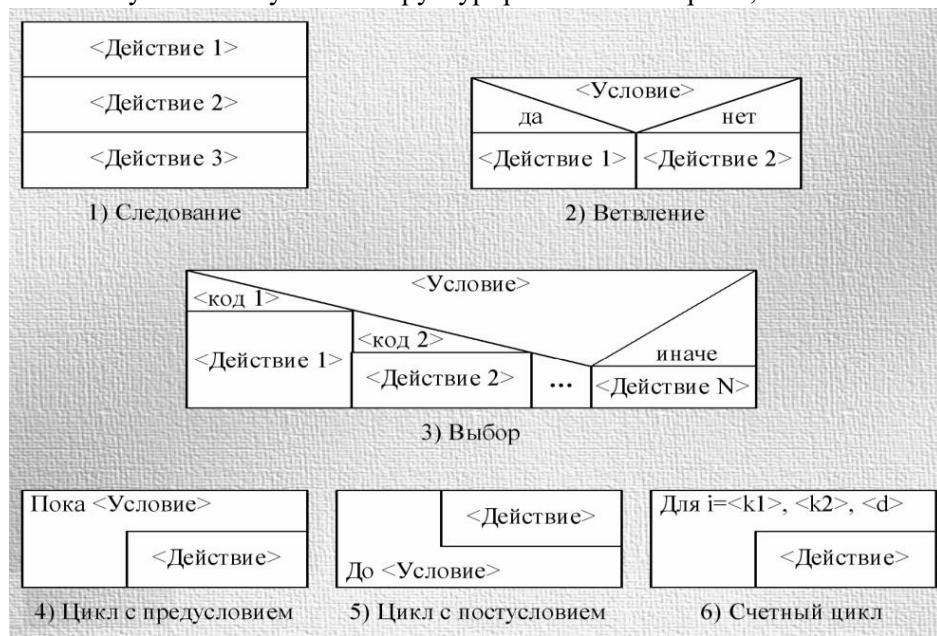
Левая часть TR содержит собственно описание условий и действий.

В правой части перечисляются все возможные комбинации условий и, соответственно, указывается, какие конкретно действия и в какой последовательности выполняются, когда определенная комбинация условий имеет место.

Диаграмма Насси-Шнейдермана

Диаграммы Насси — Шнейдермана имеют ряд преимуществ перед блоксхемами при разработке структурированных алгоритмов и программ:

- Запись является более компактной (в первую очередь за счёт отсутствия стрелок между элементами).
- Изобразив алгоритм или программу в виде диаграммы Насси — Шнейдермана, можно быть гарантировано уверенным в том, что принципы структурного программирования соблюdenы (при использовании блок-схем можно случайно получить неструктурированный алгоритм, если быть невнимательным).



Блок подпрограммы используется в случаях, когда некоторый процесс в алгоритме слишком большой, чтобы изображать его на диаграмме, или когда какие-то блоки действий используются несколько раз в разных местах одной и той же диаграммы.

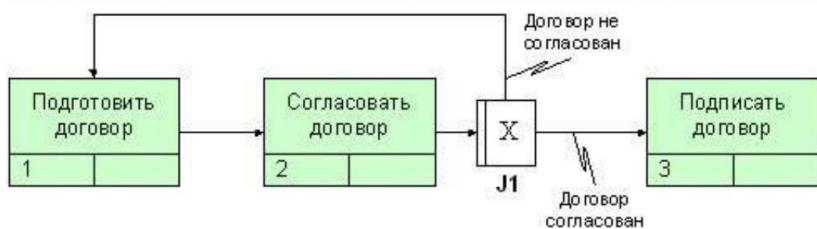


EasyCODE – среда разработки диаграмм Насси-Шнейдермана для нескольких языков программирования (C#, C++, C, Java, Cobol)

IDEF3 – метод описания процессов.

Стандарт IDEF3 является развитием WFD–подхода и предназначен для описания бизнес-процессов нижнего уровня. Он содержит объекты – логические операторы, с помощью которых показывают альтернативы и места

принятия решений в бизнес-процессе, а также объекты – стрелки с помощью которых показывают временную последовательность работ в бизнес-процессе.

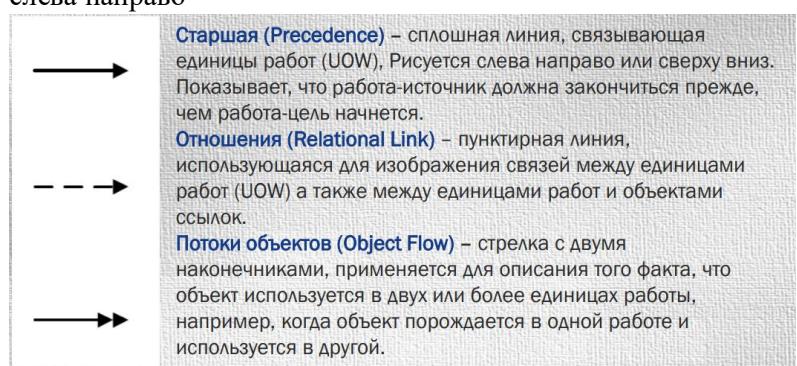


С помощью диаграмм IDEF3 можно анализировать сценарии из реальной жизни

IDEF3 – это метод, имеющий основной целью дать возможность аналитикам описать ситуацию, когда процессы выполняются в определенной последовательности, а также описать объекты, участвующие совместно в одном процессе. IDEF3 дополняет IDEF0 и содержит все необходимое для построения моделей, которые в дальнейшем могут быть использованы для **имитационного анализа**.

Единицы работы –UOW, также называемые работами (activity), являются центральными компонентами модели.

Связи. Связи показывают взаимоотношения работ. Все связи в IDEF3 односторонние и могут быть направлены куда угодно, но обычно диаграммы IDEF3 стараются построить так, чтобы связи были направлены слева направо



Типы перекрестков

Название перекрестков	Обозначение перекрестков	Смысл перекрестков	
		Схема расхождения	Схема схождения
"Исключающий ИЛИ"		Только одна последующая работа запускается	Только одна предшествующая работа должна быть завершена
"И"	Асин-хронный		Все последующие работы запускаются
	Син-хронный		Все предшествующие работы должны быть завершены одновременно
"ИЛИ"	Асин-хронный		Одна или несколько последующих работ запускаются
	Син-хронный		Одна или несколько предшествующих работ должны быть завершены одновременно

Объект ссылки в IDEF3 выражает некую идею, концепцию или данные, которые нельзя связать со стрелкой, перекрестком или работой

Тип объекта ссылки	Цель описания
OBJECT	Описывает участие важного объекта в работе.
GOTO	Инструмент циклического перехода (в повторяющейся последовательности работ), возможно на текущей диаграмме, но не обязательно. Если все работы цикла присутствуют на текущей диаграмме, цикл может также изображаться стрелкой, возвращающейся на стартовую работу. GOTO может ссылаться на перекресток.
UOB (Unit of behavior)	Применяется, когда необходимо подчеркнуть множественное использование какой-либо работы, но без цикла. Например, работа «Контроль качества» может быть использована в процессе «Изготовления изделия» несколько раз, после каждой единичной операции. Обычно этот тип ссылки не используется для моделирования автоматически запускающихся работ.
NOTE	Используется для документирования важной информации, относящейся к каким-либо графическим объектам на диаграмме. NOTE является альтернативой внесению текстового объекта в диаграмму.
ELAB (Elaboration)	Используется для усовершенствования графиков или их более детального описания. Обычно употребляется для детального описания разветвления и слияния стрелок на перекрестках.

Методология IDEF3 позволяет декомпозировать работу многократно. Это позволяет в одной модели описывать альтернативные потоки. Декомпозиция может быть сценарием или описанием. Описание включает все возможные пути развития процесса. Сценарий является частным случаем описания и иллюстрирует только один путь реализации процесса.

2.10 Процессное моделирование

- BPMN

- Основные элементы BPMN .
- Инструменты персонализации в BPMN: задачи, зона ответственности.
- BPMN: события и шлюзы.
- Практическое использование подпроцессов в BPMN.
- Средства оповещения в BPMN.
- Использование Артефактов и данных в BPMN.

BPMN отличается от остальных нотаций для моделирования бизнес-процессов:

1. во-первых, тем, что он отражает процессное моделирование.
2. во-вторых, тем, что он предназначен не только для документирования бизнес-процессов, но и для их исполнения

Основная цель BPMN — создание стандартного набора условных обозначений, понятных всем бизнес-пользователям. Бизнес-пользователи включают в себя бизнес-аналитиков, создающих и улучшающих процессы, технических разработчиков, ответственных за реализацию процессов и менеджеров, следящих за процессами и управляющих ими. Следовательно, BPMN призвана служить связующим звеном между фазой дизайна бизнес-процесса и фазой его реализации.

Разработка BPMN осуществлялась на твердой математической основе, чтобы язык реализации был максимально точен.

Основные элементы BPMN

- Объекты потока управления: события, действия и логические операторы
- Соединяющие объекты: поток управления, поток сообщений и ассоциации
- Роли: пулы и дорожки
- Артефакты: группы и текстовые аннотации
- Данные: объекты и хранилище данных.
- Действия – это точки выполнения работ в ходе Процесса.

Инструменты персонализации в BPMN: задачи, зона ответственности

Для отображения взаимодействия между участниками бизнес-процесса в нотации BPMN используются элементы **Пул и Дорожка**.

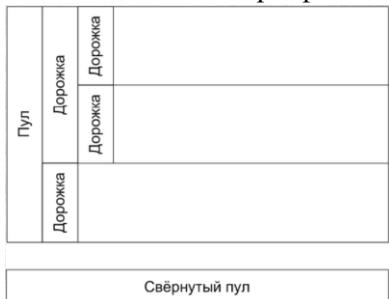
Пул и Дорожки в рамках одного потока операций в нотации BPMN называются **Оркестровкой**

Оркестровка - это описание внутреннего бизнес-процесса предприятия в виде потока взаимодействия между внутренними и внешними для организации веб-сервисами. При оркестровке существует некий центральный процесс, который управляет вызванными веб-службами и операциями. Вызванные веб-службы не знают, что они были вызваны как часть от бизнес-процесса более высокого уровня.

Хореография - это определение последовательности условий, при соблюдении которых несколько независимых участников обмениваются сообщениями с целью выполнения некоторой общей бизнес-задачи.

Пулы изображаются прямоугольником, который содержит несколько объектов потока управления, соединяющих объектов и артефактов.

Дорожки представляют собой часть пула. Дорожки позволяют организовать объекты потока управления, связывающие объекты и артефакты.



Дорожка представляет собой прямоугольник, в котором описываются все действия ответственного за выполнение задач лиц.

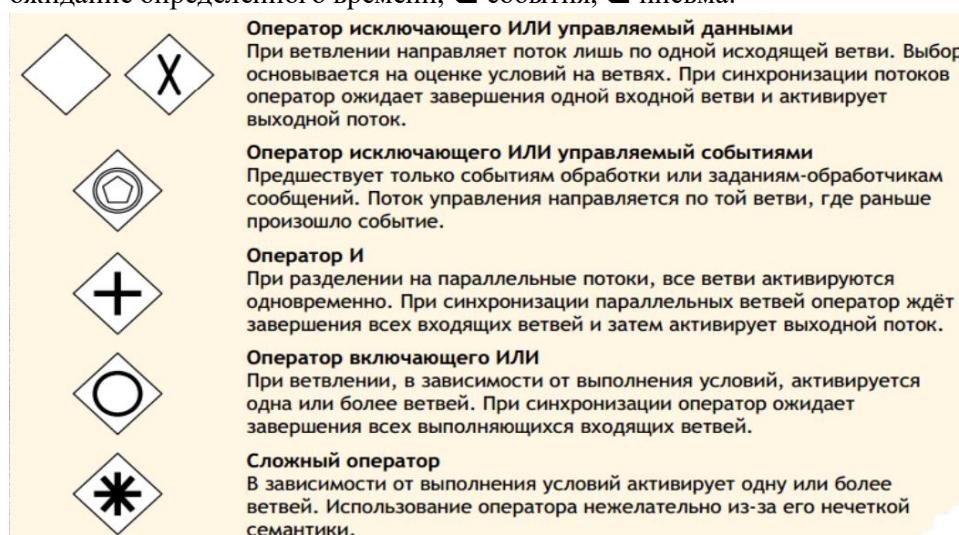
Процессы BPMN подразделяются на задачи и подпроцессы.

Задача - это простое действие (или операция), которое не имеет дальнейшей декомпозиции в рамках рассматриваемого процесса. Задачи подразделяются на типы, каждый из которых (за исключением абстрактной задачи) обозначается своим маркером в левом верхнем углу блока задачи:

1. Абстрактная задача (задача с неопределенным типом);
2. Пользовательская задача (задача, которую выполняет человек при содействии других людей или программного обеспечения);
3. Сервисная задача (задача, предназначенная для оказания услуги, которая может являться как web-сервисом, так и автоматизированным приложением);
4. Отправка сообщений (задача, суть которой заключается в отправлении сообщения внешнему участнику за пределы рассматриваемого процесса);
5. Получение сообщений (задача, суть которой заключается в получении сообщения от внешнего участника, находящегося за пределами рассматриваемого процесса);
6. Ручное выполнение (задача, выполнение которой подразумевает действия человека и исключает использование каких-либо автоматизированных механизмов исполнения или приложений);
7. Бизнес-правило (задача, суть которой заключается в выполнении бизнес-правила);
8. Задача-сценарий (задача, суть которой заключается в выполнении некоторого сценария (или скрипта) - некоторой автоматической операции).

BPMN: события и шлюзы

- Шлюзы используются для контроля расхождений и схождений потока операций в рамках процесса.
- Промежуточное событие влияет на ход процесса, однако, не может являться началом или завершением процесса и само по себе не является полноценным действием. Примерами Промежуточных событий являются: □ ожидание определённого времени, □ события, □ письма.



Практическое использование подпроцессов в BPMN

Подпроцесс - это декомпозированный процесс, включенный в состав рассматриваемого процесса, который описан более подробно на своей диаграмме.

1. Подпроцесс (подпроцесс с неопределенным типом);
2. Событийный подпроцесс (подпроцесс, не имеющий входящих и исходящих потоков управления. Событийный подпроцесс запускается всякий раз, когда его стартовое событие запускается во время выполнения родительского процесса);

3. Транзакция - подпроцесс, состоящий из набора процессов, которые в совокупности представляют некий неделимый процесс: либо весь процесс выполняется полностью, либо не выполняется вообще. Транзакции используются тогда, когда необходимо выполнить несколько процессов, но при каких-то исключительных ситуациях необходимо «откатить» выполняемые процессы;

4. Ad-Hoc процесс - подпроцесс, представляющий собой группу процессов, взаимодействие между которыми не поддаются строго регламентированным правилам. Определяется только набор процессов, однако, их последовательность и количество выполнений определяются исполнителями этих процессов.

В практике описания бизнес-процессов элемент нотации BPMN Подпроцессы используется в основном в двух случаях:

1. Для декомпозиции и повышения читаемости и наглядности схем (диаграмм);
2. Для описания повторяющихся действий. Единожды описанный Подпроцесс может многократно вызываться (использоваться) внутри различных процессов.

Действия



Транзакции



Средства оповещения в BPMN.

Основным типом бизнес-коммуникаций являются Сообщения, передаваемые и получаемые от одного участника другому в рамках процесса. **Сообщение** представляет собой содержимое (информацию) диалога между двумя участниками.

Сам же диалог (процесс обмена информацией) между двумя участниками отображается в виде Потока сообщений. **Поток сообщений** используется для отображения того порядка, в котором происходит обмен сообщениями



Поток сообщений может соединять только два разных пула (участников процесса) между собой или элементы, расположенные внутри этих пулов. Однако, Поток сообщений не может соединять два элемента, расположенные внутри одного и того же пула.

1. Поток сообщений представляет механизм запуска процесса
2. Поток сообщений используется для передачи сообщений или объектов от одного действия рассматриваемого процесса во внешний процесс
3. Поток сообщений используется для передачи сообщений или объектов из внешнего процесса (или внешней ссылки) в одно из действий рассматриваемого процесса
4. Поток сообщений используется для передачи сообщений или объектов от одного из действий внешнего процесса в одно действие рассматриваемого процесса

Использование Артефактов и данных в BPMN

Артефакты используются для введения дополнительной информации по процессу. Существует два стандартных артефакта:

- **Группа** позволяет объединять различные действия, но не влияет на поток управления в диаграмме.
- **Текстовая аннотация** используются для уточнения значения элементов диаграммы (добавления комментариев, пояснений и другой текстовой информации)

Ассоциация - используется для установки соответствия между какой-либо информацией и Артефактом и элементами потока (события, действия, шлюзы). Текстовые объекты, а также графические объекты, не относящиеся к элементам потока, могут соотноситься с элементами потока или потоком операции с помощью Ассоциации.

BPMN выделяет несколько элементов, предназначенных для хранения и передачи компонентов в ходе выполнения процесса:

- Объекты данных
- Хранилище данных.

Объект данных представляет собой информацию, которая обрабатывается в ходе процесса. Они не влияют непосредственно на последовательный поток или поток сообщений процесса, но обеспечивают информацию о том, какие действия требуют выполнения и/или что они производят.

Хранилище данных для моделирования постоянной памяти. Данный объект используется процессом для записи и извлечения данных как, например, базы данных. Сохраненная информация будет действительна даже

после завершения выполнения экземпляра процесса.

Элемент **Хранилище данных** используется для моделирования межпроцессного взаимодействия через данные, что невозможно выполнить с помощью **Объектов данных**, которые применяются только в рамках одной оркестровки

ТЕМА 3. ОСНОВЫ АНАЛИЗА ТРЕБОВАНИЙ

3.1 Роль этапа анализа требований в общем процессе разработки ПО.



Независимо от выбранной методологии разработки, можно выделить следующие этапы жизненного цикла ПО:

1. Анализ рынка, технико-экономическое обоснование проекта

2. Сбор и анализ требований к ПО
3. Проектирование архитектуры системы
4. Кодирование
5. Тестирование
6. Интеграция, развертывание
7. Эксплуатация и обслуживание (поддержка)

Наиболее активно интенсивно управление требованиями осуществляется на этапах 2 и 3, менее активно на этапах 4-5

Области применения требований

- Планирование
- Управление рисками
- Управления объемом проекта
- Проектирование
- Тестирование
- Управление изменениями

3.2 Классификация требований по предмету, назначению и уровню описания.

Классификация по предмету:

- Требования к продукту
 - В своей основе требования – это то, что формулирует заказчик. Цель, которую он преследует – получить хороший конечный продукт: функциональный и удобный в использовании.
- Требования к проекту
 - физические ресурсы, необходимые команде разработки, такие как рабочие станции, специальные аппаратные устройства, тестовые лаборатории, средства и оборудование тестирования, командные комнаты и оборудование для видеоконференций;
 - потребности в обучении персонала;
 - пользовательская документация, включая обучающие материалы, пособия, справочные руководства и информация о выпусках ПО;
 - документация для поддержки, такая как ресурсы службы технической поддержки, а также информация о техническом обеспечении и обслуживании аппаратных устройств;
 - инфраструктурные изменения, которые необходимо внести в рабочую среду;
 - требования и процедуры для выпуска продукта, установки в рабочей среде, конфигурирования и тестирования;
 - требования и процедуры для перехода со старой на новую систему, например требования по переносу и преобразованию данных, по настройке безопасности, переносу производства и обучению для восполнения недостатка квалификации — это требования иногда называют требованиями по переходу (transition requirements);
 - требования по сертификации продукта и его соответствия требованиям регулирующих органов;
 - скорректированные политики, процессы, организационные структуры и аналогичные документы;
 - сорсинг, приобретение и лицензирование ПО сторонних производителей и компонентов оборудования;
 - требования по бета-тестированию, производству, упаковке, маркетингу и дистрибуции;
 - соглашения об уровне обслуживания с клиентами;
 - требования по правовой защите (патенты, товарные знаки или авторское право) интеллектуальной собственности, связанной с разрабатываемым ПО

Классификация по типу:

- Функциональные требования
- Нефункциональные требования

Классификация по уровню:

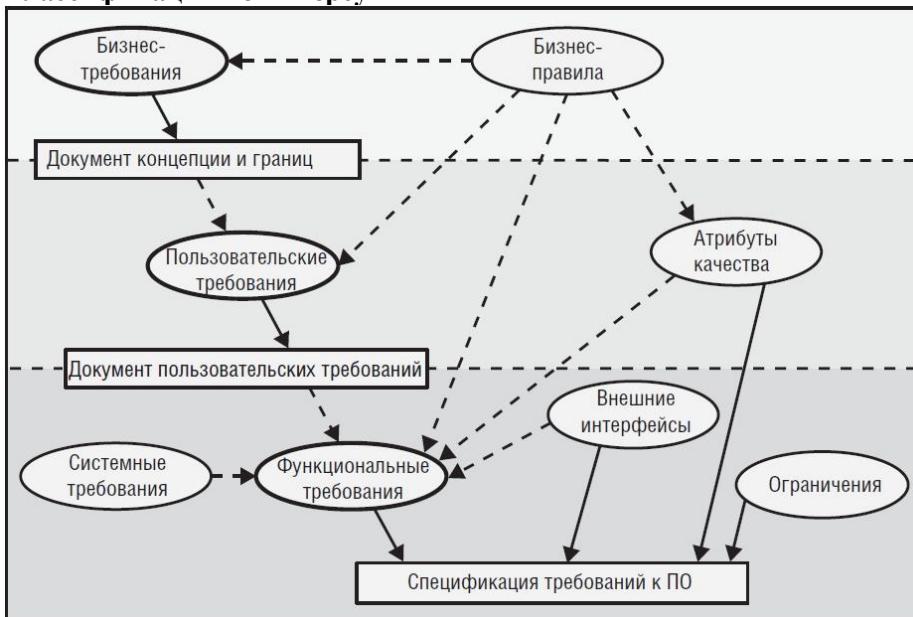
- Бизнес требования
- Требования пользователей
- Функциональные требования

Классификация по виду:

- Системные требования
- Требования программному обеспечению

3.3 Классификация требований по Вигерсу и Леффингвеллу.

Классификация по Вигерсу



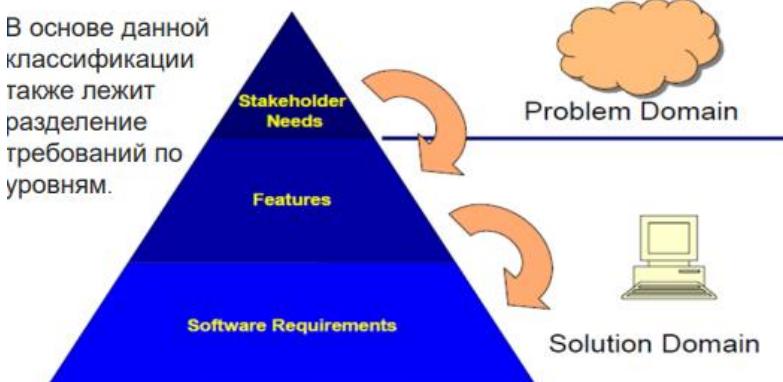
К нефункциональным требованиям, относят:

1. Бизнес правила. Набор принятых приемов организации бизнес процессов, которые должны быть учтены при разработке ПО
2. Атрибуты (характеристики) качества. Чаще всего это характеристики надежности и отказоустойчивости системы.
3. Внешние интерфейсы. Набор программных интерфейсов, которые необходимо реализовать для интеграции в существующую в организации среду программных средств.
4. Ограничения. Законодательные и другие ограничения накладываемые на создаваемое ПО

Классификация Вигерса и RUP

- RUP отождествляет понятие “требование” только с Software Requirements
- Бизнес-требования и Stakeholder/User Needs – достаточно близкие понятия, относящиеся к problem domain
- В RUP Features существует как самостоятельная классификационная единица
- Вигерс выделяет Пользовательские требования, отмечая, что они могут быть выражены вариантами использования (use cases)
- В RUP Software Requirements имеют более широкий “хват”, включая как функциональные, так и не функциональные требования
- В обоих подходах может быть создано по 3 основных документа (артефакта) на которые отображаются классификация требований

Классификация по Леффингвеллу



Уровни требований

- Потребность – отражение проблемы бизнеса, персоналии, или процесса, которое должно быть соотнесено с использованием или приобретением новой системы
- Характеристика продукта – множество логически связанных функциональных требований, которые обеспечивают возможности пользователя и удовлетворяют бизнес-целям.
 - Рассматривается как с т.з. маркетинга (a bullet item in the product description), или как “высокоуровневое выражение” поведение системы.

- Features могут быть как относящимся к функциональным, так и к нефункциональным требованиям. И могут изменяться от версии к версии продукта.

Потребности

- С точки зрения разработчиков проблемная область достаточно туманна, поэтому ее принято изображать в виде облака. Из этого облака проблем в результате анализа выявляются потребности в программном продукте.
- Анализ проблемы — это процесс осознания реальных проблем и потребностей пользователя и предложения решений для удовлетворения этих потребностей.
- Цель анализа проблемы состоит в том, чтобы добиться лучшего понимания решаемой проблемы до начала разработки.

Детализация Software Requirements

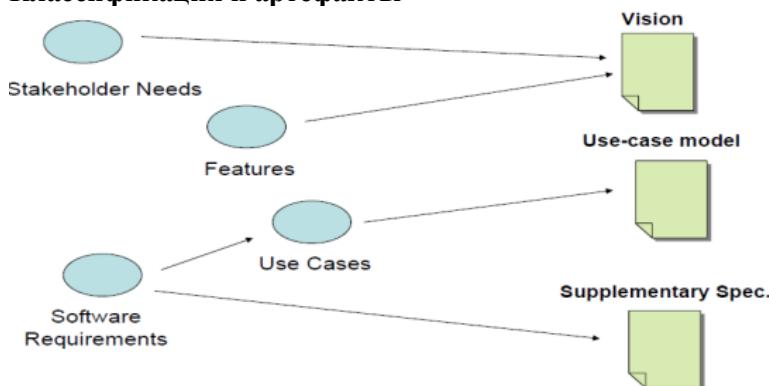


• **Функциональные требования** выражают поведение системы — входы, выходы и функции, которые система предоставляет пользователю.

• **Нефункциональные требования** (Usability, Reliability, Performance, Supportability)

• **Ограничения проектирования** — ограничения, относящиеся к дизайну или процессу создания системы, которые не влияют на внешнее поведение, но должны быть учтены для соответствия различного вида обязательствам.

Классификация и артефакты



3.4 Функциональные требования (бизнес-требования, пользовательские и функциональные).

Бизнес-требования

Назначение

Бизнес-требования описывают, почему организации нужна такая система, то есть цели, которые организация намерена достичь с ее помощью. Основное их содержание — бизнес-цели организации или клиента, заказывающих систему.

Примеры

Ускорение обработки заказов на 30 %.

Снижение издержек ведения бухгалтерии на 200 тыс. рублей в год.

Свойства

В случае с информационными системами или разработкой нестандартного приложения бизнес-требования должен определять тот, кто платит, а требования пользователей — те, кто будет стучать по клавишам и непосредственно работать с продуктом.

Всем участникам проекта рекомендуется проверить согласованность бизнес-требований и требований пользователей.

Бизнес-требования отражают организационную стратегию или бюджетные ограничения, скрытые от пользователей.

Источником бизнес-требований, как правило, является отдел маркетинга и высшее руководство.

Бизнес-требования описывают критерии успешности проекта с точки зрения спонсоров проекта.

Бизнес-требования являются требованиями высшего уровня, остальные требования подчинены им.

Пользовательские требования

Назначение

ПТ определяют набор пользовательских задач, которые должна решать ИС, а также способы (сценарии) их решения в системе. Пользовательские требования описывают, что пользователь должен делать с системой. Область

пользовательских требований также включает описания атрибутов или характеристик продукта, которые важны для удовлетворения пользователей

Примеры

система должна обеспечить возможность ввода информации о заказе, с последующим ее сохранением и возможностью передачи информации о заказе к любому сотруднику

Способы представления

фраз утверждений,

в виде вариантов использования (use case),

пользовательских историй (user story),

сценариев взаимодействия (scenario)

Свойства

Пользовательские требования отражают пожелания непосредственных пользователей системы.

Хорошей практикой является оформление пользовательских требований в виде вариантов использования (use cases).

Работе с пользовательскими требованиями следует уделять пристальное внимание, так как мнение о системе, сложившееся у непосредственных пользователей, является ключевым фактором успешности внедрения и эксплуатации системы.

Требования пользователей часто бывают плохо структурированными, дублирующимися, противоречивыми. Поэтому для создания системы важен третий уровень, в котором осуществляется формализация требований.

Важно помнить, что в большинстве проектов есть несколько классов пользователей, а также других участников, потребности которых тоже надо выявить.

Зачастую на практике «пользовательские требования» идут в разрез с «бизнес требованиями». Для примера, заказчик хочет сделать дешевую систему, пользователь, напротив, желает работать с удобным пользовательским интерфейсом, реализация которого дорого обойдется заказчику. Подобные противоречия должен идентифицировать и решать системный аналитик

Функциональные требования

Назначение

Функциональные требования определяют функциональность программного обеспечения, которую разработчики должны построить, чтобы пользователи смогли выполнить свои задачи в рамках «пользовательских» и «бизнес требований». Так как «функциональное требование» является низкоуровневым описанием действий системы, их часто описывают в виде методов (операций) класса и данных, которые обрабатываются в конкретном методе.

Пример

заказ может быть создан, отредактирован, удален и перемещен с участка на участок.

Свойства

Функциональные требования – это самый нижний уровень требований.

Функциональные требования описываются в форме традиционных утверждений со словами «должен» или «должна». Функциональные требования отвечают на вопрос «что должна делать система» в тех или иных ситуациях.

Данный вид требований детально описывает функции, которые должно выполнять ПО для удовлетворения пользовательских требований.

Функциональные требования определяют действия, которые должна выполнять система, без учета ограничений, связанных с ее реализацией.

Функциональные требования фактически являются заданиями для архитекторов и программистов.

Функциональные требования вытекают из пользовательских требований

3.5 Нефункциональные требования (бизнес-правила, атрибуты качества, внешние интерфейсы, ограничения).

Если функциональные требования задают “что” система должна делать, то нефункциональные – с соблюдением “каких условий”.

Нефункциональные требования регламентируют внутренние и внешние условия или атрибуты функционирования системы.

Атрибуты качества

Надежность (reliability) - это свойство ПС функционировать с уровнем ошибок (сбоев), не превышающем заданный (напр., д.б. обработаны не менее чем 99% всех запросов к Web-серверу);

Производительность (performance) – это свойство ПС выполнять свои основные функции с определенным уровнем быстродействия, напр.: обработка запросов к БД в диапазоне от 200 до 400 мс;

Переносимость (portability) - это характеристика ПС, которая обратно пропорционально величине затрат, необходимых на перенос системы в другую операционную среду (напр., из среды MS Windows в среду Unix), значение этой характеристики находится в интервале [0,1];

Удобство использования (usability) - это характеристика ПС, которая обратно пропорционально величине затрат, необходимых для первоначального обучения ее пользователей, и для дальнейшей ее эксплуатации, значение этой характеристики находится в интервале [0,1].

Защищенность (security) - это характеристика ПС, которая обеспечивает заданный уровень безопасности обработки данных в системе (напр., вероятность несанкционированного проникновения в систему д.б. не более 1%);

Сопровождаемость (maintainability) - это характеристика ПС, которая обратно пропорционально величине затрат, необходимых для внесение в систему требуемых изменений в ходе ее эксплуатации, значение этой характеристики находится в интервале [0,1];

Бизнес-правила — это положение, определяющее или ограничивающее какие-либо стороны бизнеса. Его назначение – защитить структуру бизнеса, контролировать или влиять на его операции. Рассмотрим различные типы правил, с которыми можно столкнуться:

1. Факты - это всего лишь верные утверждения о бизнесе. Факты также называют инвариантами – неизменными истинами о сущностях и их атрибутах.

- a. Со стоимости доставки налог с продаж не берется.
- b. Стоимость билетов не возвращается, если покупатель изменяет маршрут.

2. Ограничения определяют, какие операции может выполнять система и её пользователи. Часто применяемые слова: может, не может, должен, только.

a. При отгрузке заказа менеджер должен запросить у бухгалтера товарно-транспортную накладную и счет-фактуру.

3. Активаторы операций – правило, при определенных условиях, приводящих к какому-либо действию. Выражение вида «Если <некоторое условие верно или наступило определенное событие>, то <что-то произойдет>», — это ключ, который описывает активатор операции.

a. Если заказанный товар отсутствует на складе, то заказ передается в производственный отдел.

4. Вычисления

a. Цена единицы товара снижается на 10% при заказе от 6 до 10 единиц, на 20% — при заказе от 11 до 20 единиц и на 30% — при заказе свыше 20 единиц;

5. Выводы. Вывод создает новый факт на основе других фактов или вычислений. Выводы зачастую записывают в формате «если — то», применяемом также при записи бизнес-правил, активирующих операции; тем не менее, раздел «то» вывода заключает в себе факт или предположение, а не действие.

a. Если платеж не поступил в течение 30 календарных дней с момента отправки счета, счет считается просроченным;

b. Если поставщик не может поставить заказанный товар в течение пяти дней с момента получения заказа, заказ считается невыполненным;

Ограничения - формулировки условий, модифицирующих требования или наборы требований, сужая выбор возможных решений по их реализации. выбор платформы реализации и/или развертывания (протоколы, серверы приложений, баз данных, ...), которые, в свою очередь, могут относиться, например, к внешним интерфейсам.

Внешние интерфейсы

1. Интерфейс пользователя (User Interface, UI).

a. Ссылки на стандарты графического интерфейса пользователей или стилевые рекомендации для семейства продукта, которые необходимо соблюдать;

b. Стандарты шрифтов, значков, названий кнопок, изображений, цветовых схем, последовательностей полей вкладок, часто используемых элементов управления и т.п.;

c. Конфигурация экрана или ограничения разрешения;

d. Стандартные кнопки, функции или ссылки перемещения, одинаковые для всех экранов, например, кнопка справки;

e. Быстрые клавиши;

f. Стандарты отображения сообщений;

g. Стандарты конфигурации для упрощения локализации ПО;

h. Специальные возможности для пользователей с проблемами со зрением;

2. Интерфейсы с внешними устройствами (аппаратные интерфейсы)

3. Программные интерфейсы «Система должна взаимодействовать с существующей системой «Наша библиотека» для доступа к базе данных учебно-методической литературы»

4. Интерфейсы передачи информации (коммуникационные интерфейсы)

3.6 Системные требования.

Существуют различные понятия для “Системных требований”:

1. Классифицируются как составная часть группы функциональных требований (не путайте с как таковыми “функциональными требованиями”). Описывают высокоуровневые требования к программному

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

2. В практике компьютерной инженерии бывает другой, более узкий контекст использования данного понятия: под системными требованиями в узком смысле понимается требования, выдвигаемые прикладной программной системой (в частности – информационной) к среде своего функционирования (системной, аппаратной). Пример таких требований – тактовая частота процессора, объём памяти, требования к выбору операционной системы.

Возможные системные требования

1. требования к структуре системы;
2. требования к режимам функционирования системы;
3. требования к персоналу;
4. требования к надежности;
5. требования к безопасности;
6. требования к эргономике и технической эстетике;
7. требования к транспортабельности (для подвижных АС);
8. требования к защите информации от несанкционированного доступа;
9. ...

3.7 Свойства требований.

Хороший набор требований удовлетворяет следующим показателям качества (IEEE 830-1998 «Recommended Practice for Software Requirements Specifications»):

1. Полнота
 - а. Полнота требования означает, что текст требования не требует дополнительной детализации, то есть, в нем предусмотрены все необходимые нюансы, особенности и детали данного требования.
 - б. Различают полноту:
 - и. Полнота отдельного требования - свойство, означающее, что текст требования не требует дополнительной детализации, то есть в нем предусмотрены все необходимые нюансы, особенности и детали данного требования.
 - ii. Полнота системы требований - свойство, означающее, что совокупность артефактов, описывающих требования, исчерпывающим образом описывает все то, что требуется от разрабатываемой системы
2. Ясность (краткость, простота, точность, недвусмысленность)
3. Верифицируемость (тестируемость, возможность проверки)
4. Необходимость и полезность при эксплуатации
5. Осуществимость (выполнимость, правдоподобность, реализуемость)
6. Элементарность и трассируемость (прослеживаемость)
7. Требование считается элементарным, если оно содержит только один трассируемый элемент, который дает возможность отследить связь между ним и другими элементами информационной системы.
8. Трассируемость требования определяется возможностью отследить связь между ним и другими артефактами информационной системы (документами, моделями, текстами программ и пр.). Различают трассируемость в прямом и обратном направлениях
9. Независимость от других требований (атомарность)
10. Независимость от реализации (абстрактность)
11. Корректность (согласованность, непротиворечивость)
12. Постоянство (стабильность)
13. Наличие количественной метрики
 - а. Качественные метрики играют важную роль в верификации и аттестации информационных систем.
 - б. В первую очередь это относится к нефункциональным требованиям, которые, как правило, должны иметь под собой количественную основу.
 - с. Функциональные требования также могут расширяться количественными мерами, например, при помощи аспектов применимости.
14. Единичность - требование описывает одну и только одну вещь

3.8 Проектная документация и стандарты, регламентирующие работу с требованиями.

Стандарты, регламентирующие работу с требованиями

- Разработки IEEE:
 - IEEE 1362 "Concept of Operations Document".
 - IEEE 1233 "Guide for Developing System Requirements Specifications".
 - IEEE Standard 830-1998, "IEEE Recommended Practice for Software Requirements Specifications"

- IEEE Standard Glossary of Software Engineering Terminology/IEEE Std 610.12-1990
- IEEE Guide to the Software Engineering Body of Knowledge (1) - SWEBOK®, 2004.
- **Отечественные ГОСТ**
 - ГОСТ 34.602-89. Информационная технология. Техническое задание на создание автоматизированной системы
 - ГОСТ 19.201-78. Техническое задание. Требования к содержанию и оформлению.
 - РД 50 34.698-90. Автоматизированные системы. Требования к содержанию документов
 - ГОСТ 34.601-90. Информационная технология. Автоматизированные системы. Стадии создания.

3.9 Работа с требованиями. Основные этапы.

- Планирование требований.
- Разработка требований
 - Сбор требований (выявление)
 - Выявление бизнес- и пользовательских требований
 - Анализ проблемы на начальной стадии выявления требований
 - Препятствия при выявлении требований
 - Традиционные методы выявления требований
 - Современные методы выявления требований.
 - Проблемы, возникающие при выявлении требований
 - Анализ требований
 - Уточнение требований (полнота требований)
 - Структуризация и идентификация требований
 - Проверка на непротиворечивость требований
 - Установление рисков, приоритетов и атрибутов
 - Модели требований
 - Документирование
 - Проверка требований
- Управление изменениями требований

Планирование требований

Планирование необходимо для решения следующих проблем:

- Неучтенные требования
- Недостаточный уровень детализации требований для команды
- Смешанные уровни абстракции и типы требований
- Внезапное изменение требований
- Отсутствие опыта и навыков аналитической команды
- Сложность планирования и контроля аналитических задач
- Срыв сроков и разрастание бюджета проекта



Планирование – это этап, на котором определяются:

- Методология
- Аналитические процессы
- Регламент работ
- Описание командных ролей
- Шаблоны аналитических документов

- План управления требованиями
- План работ группы аналитиков

Процесс разработки и управления требованиями

- Используйте методологию
- Адаптируйте процесс под ваш проект/компанию
- Учитывайте предпочтения заказчика
- Согласуйте процесс с командой
- Опишите процесс в виде текста и/или диаграмм

Процесс проверки качества требований

Определите и запишите:

- Кто участвует в проверке
- В какой последовательности проверяют
- Количество итераций согласования одного документа
- Критерии качества требований
- Сроки согласования
- Протоколы замечаний

Процесс управления изменениями требований

Основные шаги:

- Определите источник изменения
- Проанализируйте влияние изменений на проект
- Примите совместное решение
- Зафиксируйте изменения в требованиях
- Реализуйте изменения
- Протестируйте изменения

Роли и обязанности аналитиков

Лидер аналитиков

- Распределяет аналитические задачи
- Контролирует выполнение задач
- Расставляет приоритеты для требований

Бизнес-аналитик

- Описывает бизнес-процессы заказчика
- Подготавливает модель предметной области



Взаимосвязи системного аналитика с представителями заказчика и проектной команды



Бизнес-аналитик отвечает за построение модели предметной области (онтологии) предприятия, бизнес-процессы которого необходимо автоматизировать с помощью разрабатываемой системы. Бизнес-аналитик занимается выявлением «бизнес требований» и др. информации, а именно:

- идентификацией заинтересованных лиц
- изучением внешней и внутренней нормативной документации (законов, методик, распоряжений, стандартов, регламентов и т.д.)
- подготовкой и проведением интервью с экспертами предметной области и заказчиками.
- подготовкой и проведением анкетирования и опросов экспертов предметной области и пользователей будущей системы.
- Бизнес-аналитик формализует и систематизирует собранную информацию. Он создает словарь терминов (глоссарий) предметной области, описывает организационно-штатную структуру предприятия, создает концептуальные функциональные модели деятельности, модели сущностей и связей (ERD), модели потоков данных, модели бизнес-процессов (как есть). Также бизнес-аналитик фиксирует ограничения бизнеса и бизнес-правила.

Системный аналитик

- Аналитик требований, или системный аналитик, отвечает за сбор и управление функциональными требованиями, характеристиками качества и ограничениями к создаваемой системе. Системный аналитик разрабатывает план управления требований, в котором определяет тип требований, их атрибуты, модели и документы, которые будут использоваться при анализе и документировании требований.
- Системный аналитик отвечает за формирование технических требований к разрабатываемой системе. После сбора требований системный аналитик проверяет их на непротиворечивость и дублирование, определяет приоритеты, устанавливает отношения прослеживаемости между требованиями. Системный аналитик также разрабатывает варианты использования системы

Спецификатор требований

- Спецификатор требований совместно с системным аналитиком разрабатывает спецификацию требований, детализирует варианты использования и другие требования.

Системный архитектор

- Системный архитектор отвечает за разработку технической части концепции создания системы. Системный архитектор разрабатывает варианты концептуальной архитектуры системы, определяет место системы в линейке продуктов организации, порядок взаимодействия с собственными и сторонними продуктами, определяет элементы архитектуры системы, разрабатывает логические модели данных. Системный архитектор также может участвовать в разработке или разрабатывать самостоятельно технико-экономическое обоснование выбранного варианта концепции

Технический писатель

- Технический писатель разрабатывает документацию на информационную систему согласно российским и международным стандартам.

Коммуникатор

- Коммуникатор отвечает за организацию, проведение, моделирование и протоколирование совещаний, фокус-групп и мозговых штурмов.

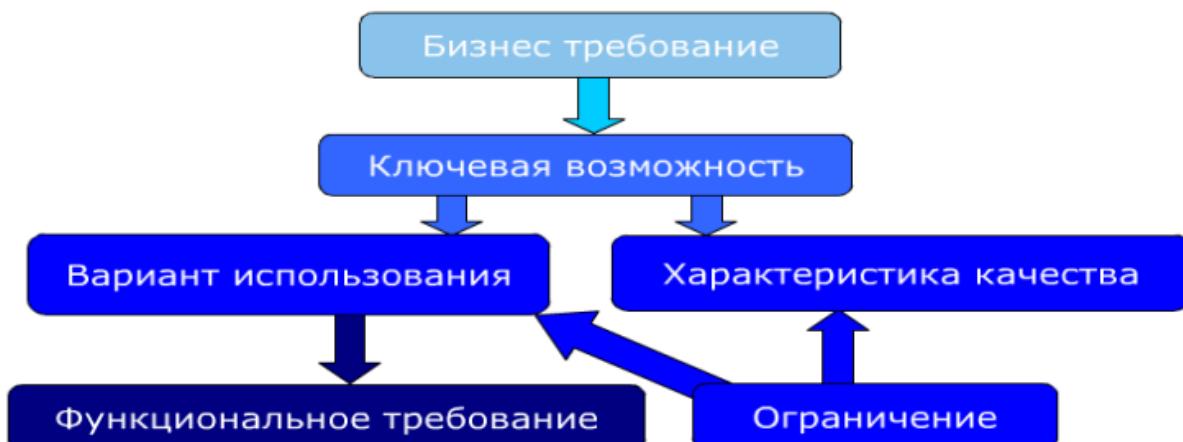
БА vs СА

Бизнес Аналитик	Системный аналитик
<ul style="list-style-type: none">• Знание (изучение) Пр. Обл.• Анализ структуры Орг.• Участие в Стратегии• Выявление З. Л.• Описание БП• Выявление Целей• Выявление Проблем• Выявление Потребностей• Оптимизация БП• Формирование Задач ПО	<ul style="list-style-type: none">• Изучение Пр. Обл.• Формулирование Задач ПО• Формулирование ПТ• Изучение ПО конкурентов• Выявление Пользователей• Формулирование ФТ и БПр• Формулирование НеФТ• Участие в разработке Арх.• Участие в Тестировании• Участие во Внедрении

План управления проектом

- Составление сметы по аналитическим работам
- Оценка трудозатрат
- Распределение задач
- Идентификация контрольных точек
- Набор команды аналитиков

Типы требований и трассировка



Атрибуты требований

- Уникальный идентификатор
- Приоритет
- Статус
- Версия ПО (build)
- Трудоемкость
- Другие

Инструменты и ПО

- Система управления требованиями
- Средство моделирования требований
- Средства коммуникации

Отчеты и метрики

- Матрицы трассировок
- Выборки требований по критериям
- Статистические данные по требованиям
- Другие отчеты

Обучение и тренинги

- План обучения команды
- Индивидуальный план обучения аналитика

- Обмен опытом внутри команды
- Семинары и конференции

Разработка требований

Процесс разработки включает 4 основных этапа:

- Сбор требований (выявление), нацеленный на выявление всех возможных источников требований, ограничений на работу системы и извлечение требований из этих источников.
- Формирование и анализ требований, целью которого является обнаружение и устранение противоречий и неоднозначностей в требованиях, их уточнение и систематизация.
- Документирование (описание) требований. В результате этой деятельности требования должны быть оформлены в виде структурированного набора документов и моделей, который может систематически анализироваться, оцениваться с разных позиций и в итоге должен быть утвержден как официальная формулировка требований к системе
- Проверка (утверждение) требований решает задачу оценки понятности сформулированных требований и их характеристик, необходимых, чтобы разрабатывать ПО на их основе, в первую очередь, непротиворечивости и полноты, а также соответствия корпоративным стандартам на техническую документацию.

Сбор требований

- Определение источников требований. В его ходе выявляются все заинтересованные лица, документы, эксперты, существующие системы, которые могут служить источниками информации для составления требований к рассматриваемой системе, а также источниками ограничений на ее работу.
- Извлечение требований — получение формулировок требований и ограничений из бесед с экспертами или заинтересованными лицами, наблюдением за работой пользователей, анализа документов, отраслевых и корпоративных стандартов, реинжиниринга существующих систем и анализа интерфейсов систем, с которыми придется взаимодействовать, и т.д.

Шаги:

- Выделяем бизнес-процесс
- Детальное изучение бизнес-процесса
- Графическое и/или текстовое описание бизнес-процесса (первичное)
- Согласование с исполнителями и владельцем бизнес-процесса
- Выделение показателей бизнес-процесса
- Окончательное документирование бизнес-процесса
- Дальнейшие действия (или их отсутствие), в зависимости от целей проекта

Источники требований

- Основным источником требований к информационной системе, безусловно, являются соображения, высказанные представителями Заказчик
- Артефакты, описывающие предметную область. Это могут быть
 - документы с описанием бизнес-процессов предприятия, выполненные консалтинговым агентством
 - либо просто документы (должностные инструкции, распоряжения, своды бизнес-правил), принятые на предприятии.
- Так называемые "лучшие практики", широко используемые в настоящее время в бизнес-консалтинге и при внедрении корпоративных информационных систем. Лучшие практики представляют собой описания моделей деятельности успешных компаний отрасли, используемые длительное время в сотнях и тысячах компаний по всему миру.

Источники требований

- Федеральное и муниципальное отраслевое законодательство (конституция, законы, распоряжения)
- Нормативное обеспечение организации (регламенты, положения, уставы, приказы)
- Текущая организация деятельности объекта автоматизации
- Модели деятельности (диаграммы бизнес-процессов)
- Представления и ожидания потребителей и пользователей системы
- Журналы использования существующих программно-аппаратных систем
- Конкурирующие программные продукты

Методы получения требований

- **Аналитические техники.** В их основе лежит анализ каких-либо документов, знаний, систем или деятельности.
 - **Размышление (introspection)** — аналитик размышляет над возможными функциями системы и выбирает те, которые выглядят достаточно привлекательными и полезными.

- **Анализ документов и моделей** (review, analysis) — анализируются нормативные документы, документы, описывающие реальность или пожелания и проблемы, модели, описывающие предметную область. Из них выделяются потребности, возможные функции и критерии их оценки.
- Анализ имеющихся аналогичных систем.
 - **Анализ выборки данных** (sampling) — анализируются большие массивы имеющихся данных (отчетов, внутренних данных системы, входной информации и пр.), выделяются сущности, их атрибуты и связи.
 - **Анализ функций** — анализируется работа одной или нескольких имеющихся систем, выделяются выполняемые ими операции и возможные правила их выполнения
- **Техники, основанные на общении.** В их основе лежит общение между аналитиком и другими людьми, нацеленное на выявление потребностей заинтересованных лиц и ограничений предметной области
 - **Индивидуальные техники.** В них используется только индивидуальное общение аналитика с другими людьми.
 - **Интервью** (interview) — заинтересованное лицо отвечает на вопросы аналитика, часть их которых подготовлена заранее, другая часть возникает в процессе диалога.
 - **Опрос** (survey) — ряд заинтересованных лиц отвечает на вопросы из заранее подготовленного списка (вопросника, questionnaire).
 - **Групповые техники.** В них используется групповое общение, повышающее отдачу каждого сеанса, но и приводящее к возникновению конфликтов, различных позиций и пр., и поэтому нуждающееся в контроле и управлении.
 - **Групповое интервью** (group interview) или **фокус-группа** (focus group) — группа экспертов собирается и обсуждает совместно с аналитиком возможные функции и требования. Обсуждение направляется вопросами аналитика по заранее подготовленному сценарию.
 - **Мозговой штурм** (brainstorming session) — несколько человек, включая аналитиков и заинтересованных лиц, некоторое время выдумывают различные возможности, которые могли бы быть полезны, не ограничивая воображение.
 - **Метод Дельфи** (Delphi method) — опрос группы экспертов на одну тему. Каждый из них сначала готовит документ с изложением своего мнения. Затем каждый из полученных документов передается для анализа и замечаний другому эксперту. Такая передача может повторяться несколько раз.
 - **Ролевая игра** (role playing game) — участники группы разыгрывают возможные сценарии работы, каждый играет свою роль. Выявляемые в ходе игры проблемы, потребности, ограничения фиксируются.
 - **Демонстрационные техники.** Они используют демонстрацию одному или группе экспертов моделей системы или возможных сценариев работы системы для получения от них откликов и замечаний, которые обычно тут же фиксируются.
 - **Раскадровка** (storyboard) — сценарий работы в некоторой ситуации расписывается по отдельным действиям и представляется пользователям и экспертам. Представляться он может в виде описания действий, набора рисунков, анимации, и пр. Собираются замечания и поправки, высказанные по поводу такого сценария
 - **Демонстрация прототипа** (prototype demo) — демонстрируется работа прототипа или отдельных модулей системы в определенной ситуации. Собираются замечания пользователей и экспертов.
 - **Модели - техники**, использующие в качестве основы общения некоторые модели (model based techniques). Таких техник много, любой вид моделей, особенно графических или симуляционных, можно использовать для общения с экспертами и пользователями и фиксации их замечаний к такой модели. В качестве моделей часто используются описания сценариев работы системы, варианты использования, диаграммы потоков данных, концептуальные модели данных системы. Отдельно можно упомянуть модели, фиксирующие выделяемые цели операций системы и связанные с ними техники (goal-driven, а также модели, фиксирующие различные точки зрения (viewpoint-based))
 - **Когнитивные техники.** Это специализированные техники, основанные на некоторых положениях когнитивной психологии. Исторически они возникли как приемы выявления структур мышления у людей с психическими отклонениями. После достижения ряда успехов в этой области техники такого рода стали применяться и для выявления мыслей и знаний в более широком контексте, в частности, для выделения требований
 - **Анализ задач** (task analysis) — анализируются задачи, решаемые пользователем, и операции, предпринимаемые им для их решения. Целью анализа является определение информации, необходимой для выполнения каждой операции, и взаимосвязей между различными используемыми в процессе решения сущностями. Обычно проводится аналитиком, присутствующим при работе пользователя и задающим вопросу по ходу его работы.
 - **Анализ протоколов** (protocol analysis) — пользователь или эксперт проговаривает вслух все выполняемые действия при конкретном или абстрактном сценарии. При этом приводятся их обоснования и возможные варианты

- **Устойчивые решетки или решетки Келли** (repertory grids). Выбирается набор объектов предметной области, для которых каждый эксперт из группы придумывает систему атрибутов и их значений. Потом эксперты пытаются найти наиболее близкие атрибуты в своей системе к атрибутам в системе другого эксперта, а также оценить значения атрибутов в чужих системах. Есть еще несколько разновидностей этой техники.
- **Сортировка карточек** (card sorting). На карточки заносятся названия объектов предметной области. Обычно один объект присутствует на одной карточке, но в некоторых случаях используется модификация — один объект может быть на нескольких карточках. После этого каждому из нескольких экспертов выдают полный набор карточек и просят разделить карточки на группы по каким-то признакам. Затем каждый формулирует обоснование своей группировки.
- **Лестница** (laddering) — используется иерархическая система вопросов-зондов (probes) для выявления знаний опрашиваемого о предметной области.
- **Масштабирование близости** (proximity scaling). Определяется набор объектов предметной области, список которых выдается экспертам. Те должны определить набор характеристик для их описания и степень близости этих объектов друг к другу по выбранному набору характеристик.
- Контекстуальные техники
 - Анализ разговоров и речи во время интервью или совещаний.
 - Анализ неверbalного поведения.
 - Запись поведения человека на видео и последующий его детальный анализ.
 - Запись и анализ физиологических параметров (пульс, давление, электропроводность кожи и пр.).

Традиционные методы выявления требований

- Интервьюирование заказчиков и экспертов проблемной области
- Анкетирование
- Наблюдение (пассивное и активное)
- Изучение документов и программных систем

Современные методы выявления требований

- Прототипирование. Прототип представляет собой демонстрационную систему — «наскоро и грубо» сделанную рабочую модель решения, которая представляет пользовательский GUI-интерфейс и моделирует поведение системы при инициировании пользователем различных событий
- Совместная разработка приложений (JAD-метод) – метод разработки, осуществляемый в ходе одного или нескольких совещаний с привлечением всех участников проекта. Проведение JAD-совещаний может занимать несколько часов, несколько дней или даже пару недель
- Быстрая разработка приложений (RAD-метод)

Прототипирование — это техника для построения быстрой и приблизительной версии желаемой системы или части этой системы. Прототип демонстрирует возможности системы пользователям и дизайнерам. Программный прототип - "зеркало", в котором видно отражение того, как понял Исполнитель требования Заказчика. В некоторых случаях, создание прототипов может создать впечатление, что разработчики зашли дальше в развитии проекта, чем есть на самом деле, что может предоставить пользователям нереалистичные ожидания окончания проекта

Горизонтальный прототип представляет собой изображения пользовательских интерфейсов, которые демонстрируют пользователю внешний вид будущей системы и характеризуют методы работы с системой. Выполнение задач может быть представлено последовательностью сменяющих друг друга пользовательских графических интерфейсов. Горизонтальные прототипы позволяют исследовать поведение системы в различных ситуациях: обработке ошибок, разветвление потоков задач и др.

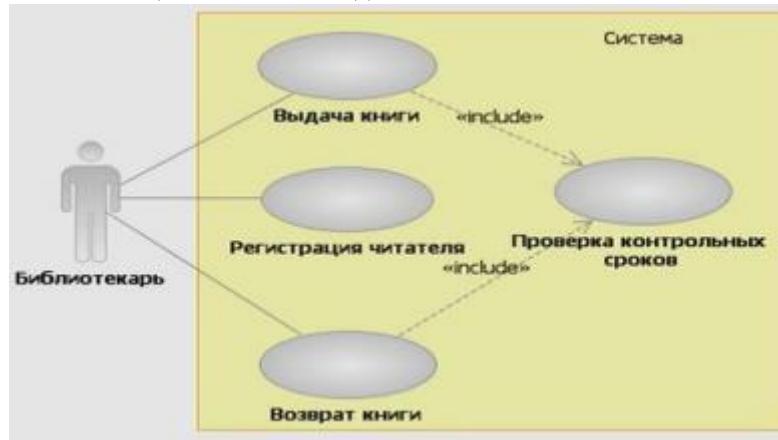
Вертикальный прототип предназначен для проверки определенной функциональности системы и исследования критически важных требований к интерфейсу, времени выполнения. Вертикальный прототип является законченной реализацией части будущей системы. С помощью вертикальных прототипов можно проверить удовлетворяет ли разработанная системная архитектура требованиям характеристик качества, произвести нагружочное тестирование приложения.

Что входит в прототипирование

- Описание целевой аудитории
- Построение пользовательских сценариев
- Обзор рынка

Варианты использования - Пользовательские требования записываются в виде модели вариантов использования, которая содержит описание внешних по отношению к системе действующих лиц и их взаимодействие с

разрабатываемой системой. Варианты использования может иметь от одного до нескольких сценариев, описывающих шаги взаимодействия системы с пользователем или другой системой.



Проблемы, возникающие при выявлении требований

1. **Проблемы классификации требований.** Сложной задачей при выявлении требований является идентификация их типов. На практике, особенно при проведении интервью, требования, формулируемые пользователем могут относиться к различным типам требований. Существуют следующие рекомендации о том, как «понимать» пользователей и классифицировать требования:
 - a. **Бизнес требования** - Любая информация, описывающая рынок или какие-либо интересы бизнеса
 - b. **Ключевые возможности** - Предположения пользователя о выполняемых системой задачах.
 - c. **Вариант использования** - Пользователь говорит о том, что ему необходимо что-то сделать
 - d. **Характеристика качества** - Необходимые характеристики системы, насколько хорошо работает система.
 - e. **Ограничения** - Пользователь говорит о том, что система или функция должна соответствовать стандарту или правилу, реализована на предопределенной платформе
 - f. **Функциональные требования** - Передаваемая информация, допустимые значения данных, поведенческие характеристики системы.
2. **Проблемы формулировки требований**
 - a. Во время интервью многие пользователи и заказчики системы сталкиваются с проблемой объяснения выполняемых задач аналитику. Они помнят исключения и забывают порядок действий, забывают рассказать о простых на их взгляд задачах. Данная ситуация может привести к тому, что один и тот же процесс будет описан по-разному несколькими пользователями, что введет в заблуждение остальных и создаст ложное единое мнение.
 - b. Для решения подобной проблемы, аналитик может наблюдать за работой пользователя и документировать все действия, выполняемые пользователем в его повседневной работе, или попросить пользователя самостоятельно документировать все выполняемые действия.
3. **Терминология.** В процессе разработки системы заказчики и пользователи выражаются с использованием терминологии, являющейся специфичной для их предметной области. Разработчики программного продукта должны понимать терминологию заказчика, для этого аналитик должен собирать термины предметной области заказчика и документировать их в гlosсарии. Гlosсарий терминов позволит не только понять требования пользователей, но и разобраться в предметной области разрабатываемой системы
4. **Предвзятые решения**
 - a. Иногда заказчики и пользователи считают, что сами знают решение проблем их деятельности, вместо описания проблемы они высказывают идею решения этой проблемы, которая могут быть неоптимальными или просто ошибочными.
 - b. Для решения данной проблемы необходимо анализировать причины, т.е. определять, почему пользователи хотят иметь некоторую функциональность системы, действительно она нужна или является субъективным желанием пользователя.

Формирование и анализ требований

Цель анализа

- Основной целью анализа требований является их систематизация и избавление от дублируемых данных. Это достигается за счет разделения пользовательских историй на отдельные пакеты по функциональному признаку и их иерархической структуризации.
- По окончании этапа анализа требований, многостраничный документ, содержащий сотни пользовательских историй, будет разбит на части. Каждая часть будет освещать только необходимую функциональность, а в её основе будет стоять диаграмма вариантов использования, на основе которой можно будет легко увидеть

все требуемые функции системы. Описание вариантов использования не будут дублироваться, а лишь дополнять друг друга

Задачи этапа

1. Уточнение требований
2. Структуризация и идентификация требований. Цели:
 - a. Определение границ системы.
 - b. Разнесение требований по функциональным областям предметной области
 - c. Декомпозиция требований позволяет разбивать требования на более детальные. Каждое родительское требование может иметь несколько дочерних требований, а одно дочернее требование принадлежать только одному родителю

Атрибуты требований

- Приоритет
- Статус
- Стоимость реализации
- Сложность реализации
- Стабильность
- Ответственный исполнитель
- № итерации (этапа) в плане

Модели визуального представления требований

- диаграммы потоков данных (data flow diagrams, DFD);
- диаграммы рабочих потоков, такие как диаграммы swimlane;
- диаграммы переходов состояний (state-transition diagrams, STD) и таблицы состояний;
- карты диалоговых окон; θ таблицы и деревья решений;
- таблицы событий и реакций;
- деревья функций
- диаграммы вариантов использования;
- диаграммы процессов;
- диаграммы «сущность-связь» (entity-relationship diagrams, ERD)

Глоссарий представляет собой документ, описывающий все основные определения, встречающиеся в проекте. Сосредоточение определений в одном документе позволяет отслеживать изменения определений, добавление новых и удаление старых определений. Глосарием могут пользоваться все участники проекта по разработке программного обеспечения: аналитики, разработчики, системные архитекторы, инженеры по тестированию и заказчики. В основном глоссарий содержит определения предметной области и дает участникам проекта возможность понимания предметной области. Также глоссарий позволяет участникам проекта разговаривать на «одном» языке. За ведение глоссария обычно отвечает системный аналитик.

Спецификация требований – процесс документирования системы в структурированном, доступном всем и управляемом формате.

Характеристики:

- Полнота. Никакие требования или необходимые данные не должны быть пропущены.
- Согласованность. Согласованные требования не конфликтуют с другими требованиями такого же типа или с высокоуровневыми пользовательскими, системными или бизнес-требованиями. Несогласованность документов следует устраниТЬ до начала процесса разработки. Вы не всегда знаете, какое именно положение некорректно (если какое-то некорректно), пока не выполните исследование. Рекомендуется записывать автора каждого требования, чтобы узнать, кто его высказал, если конфликт все-таки будет обнаружен.
- Способность к модификации. Необходимо обеспечить возможность переработки требований, если понадобится, и поддерживать историю изменений для каждого положения. Для этого все они должны быть уникально помечены и обозначены, чтобы вы могли ссылаться на них однозначно. Каждое требование должно быть записано в спецификации только единожды. Иначе вы легко получите несогласованность, изменив только одно положение из двух одинаковых. Лучше используйте ссылки на первоначальные утверждения, а не дублируйте положения. Модификация спецификации станет гораздо легче, если вы составите содержание документа и указатель. Сохранение спецификации в базе данных коммерческого инструмента управления требованиями сделает их пригодными для повторного использования.
- Трассируемость. Трассируемость, или возможность для анализа, можно реализовать как в направлении назад, к первоисточникам, так и вперед, к элементам дизайна и исходному коду, который его реализует, а также к вариантам использования, которые позволяют проверить корректность, реализации. Трассируемые требования помечены соответствующими идентификаторами. Они записаны в структурированной, детализированной форме, в противоположность параграфам в длинной повествовательной форме.

Избегайте слипания множества требований в один ком, отдельные требования можно трассировать к различным элементам дизайна и кода

Документирование требований

Правила документирования

- Используйте шаблоны спецификации требований к ПО
- Сопровождайте требования указанием их источников
- Присваивайте требованиям уникальные идентификаторы
- Указывайте количественные характеристики
- Не включайте в требования элементы дизайна
- Определяйте термины в начале спецификации
- Используйте одинаковую форму требований
- Используйте слово «должен»
- Избегайте в требованиях негативных утверждений
- Количество деталей требований может варьироваться

Основные документы

1. Глоссарий
2. Документ-концепция
3. Спецификация требований

Шаблон описания требования

- Общие сведения о требовании
- Функции, реализующие требования
- Связи между требованием и функциями
- Описание процесса выполнения функции
- Нефункциональные требования, связанные с функциональным требованием
 - Временной регламент
 - Надежность
 - Точность
 - Достоверность
 - Связи с другими функциональными требованиями

Критерий законченного набора требований

- полнота: все необходимые требования записаны
- непротиворечивость: не существует требований противоречащих друг другу
- отсутствие избыточности: каждое требование сформулировано только один раз (нет повторов)
- модульность: требования близкие друг другу содержатся в одном разделе
- структурированность: наличие ясной четкой структуры документа с требованиями
- удовлетворенность: достигнут необходимый уровень покрытия требований связями типа «удовлетворяет»
- тестируемость: достигнут необходимый уровень покрытия требований тестами

Функциональные требования

- Tag (таг) – уникальный, неизменный идентификатор
- Gist (суть) – название требования, раскрывающее его сущность
- Requirement (требование) – описание требования со всеми необходимыми деталями
- Rationale (обоснование) – обоснование необходимости требования
- Priority (приоритет) – приоритет требования (обычно используется от трех до пяти приоритетов: "высокий", "средний", "низкий" и "неопределенный")
- Stakeholders (участники) – заинтересованные лица, для которых важно требование
- Owner (владелец) – лицо, ответственное за имплементацию требования
- Author (автор) – лицо, записавшее требование
- Revision (версия) – версия требования
- Date (дата) – дата записи последней версии
- Assumptions (предположения) – все предположения о проблемах, которые связаны с требованием
- Risks (риски) – все, что может негативно сказаться на результате имплементации требования
- Defined (определения) – определение термина

Нефункциональные требования

- Ambition (назначение) – описание цели требования
- Scale (шкала) – шкала, используемая для количественной характеристики требования

- Meter (измеритель) – процесс или устройство, используемые для установления количественной характеристики
- Minimum (минимум) – минимальное допустимое значение количественной характеристики
- Target (цель) – значение количественной характеристики, при котором требование считается реализованным хорошо
- Outstanding (достижение) – наилучшее значение количественной характеристики, которое может быть достигнуто, если все будет реализовано наилучшим образом
- Wish (идеал) – значение количественной характеристики, к которому надо стремиться, но достигнуть которого практически невозможно
- Past (прошлое) – предыдущий результат, который можно использовать для сравнения с текущим.
- Trend (тенденция) – массив исторических данных по развитию требования
- Record (рекорд) – лучший достигнутый результат в индустрии

Проверка требований

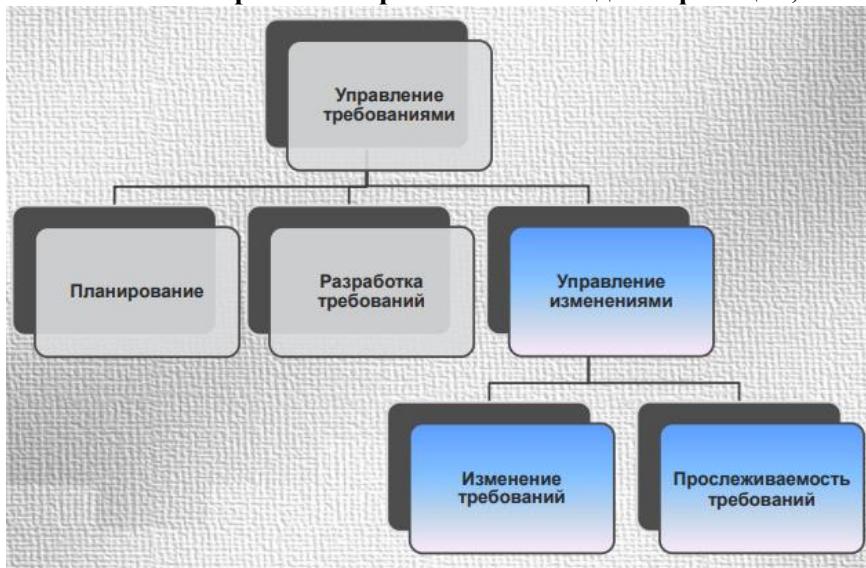
Верификация — проверка того, что продукт делался правильно, т.е. проверка того, что он разрабатывался в соответствии со всеми требованиями по отношению к процессу и этапам разработки.

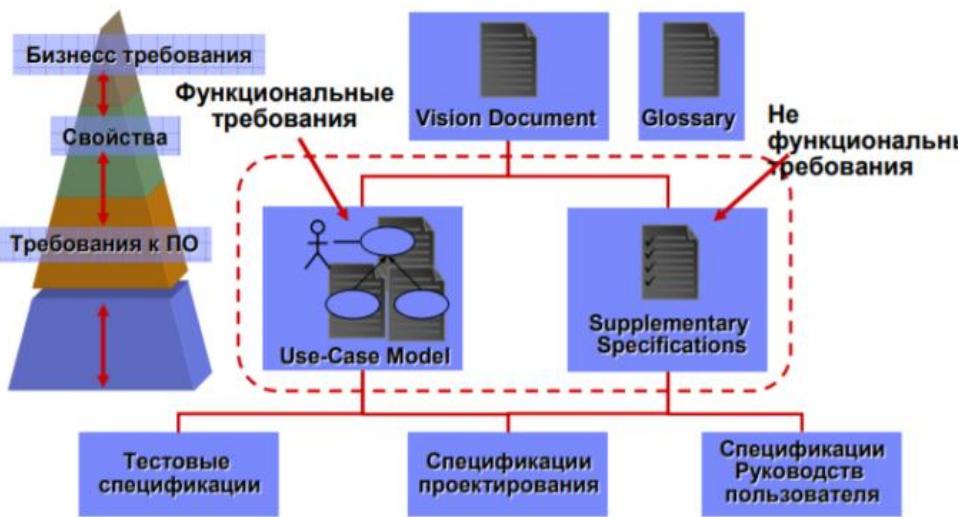
Валидация (аттестация) — проверка того, что сам продукт правилен, т.е. установление того, что он удовлетворяет требованиям, ожиданиям пользователя, заказчика и других заинтересованных сторон.

Методы и средства проверки требований

1. Неофициальные просмотры требований
2. Инспекции – это формализованный процесс, выполняемый небольшой группой специалистов. Члены группы системно анализируют объект инспекции и определяют возможные дефекты. Члены группы должны выполнять различные роли.
3. Разработка тестов
 - a. Механизм вариантов использования (uses cases) позволяет ответить на вопрос: как будет использоваться система. Чтобы проверить систему, используется аналогичный механизм: тестовых сценариев (test cases).
 - b. Тестовые сценарии (ТС) рекомендуется создавать уже на ранних стадиях работы с требованиями, в идеале - после получения запросов ЗЛ, параллельно с разработкой вариантов использования.
 - c. Тестовые сценарии, как и варианты использования, могут поддерживать разные уровни абстракции. Различаются концептуальные и детальные ТС. Концептуальный уровень предполагает проработку процедуры тестирования, инвариантную к конкретной реализации ПЛ.
4. Трассирование
 - a. Все более популярным методом верификации спецификаций требований к ПО становится трассирование спецификаций. Данная технология базируется на отслеживании взаимосвязей между требованиями различных видов и характеристиками созданного продукта. То есть между тем, что задумывалось, и тем что получилось.
5. Формирование критериев приемлемости

3.10 Управление требованиями - идентификация, изменение, прослеживаемость.





Причины изменения требований

- Внешние факторы
 - Произошли изменения проблемы, которую мы пытались решить с помощью новой системы (возникли изменения в экономике, в правительственные инструкциях, на рынке или же изменились предпочтения потребителей).
 - Пользователи изменили свое мнение о том, чего они хотят от системы, или свои предпочтения.
 - Изменилась внешняя среда, что привело к появлению новых ограничений и/или новых возможностей. Одним из наиболее очевидных примеров изменения среды является постоянно происходящее совершенствование систем аппаратного и программного обеспечения.
 - Вошла в эксплуатацию новая система. Одним из самых неожиданных внешних факторов возникновения изменений является то, что само появление новой системы приводит к тому, что меняются требования к ней
- Внутренние факторы
 - При первоначальном выявлении требований нам не удалось задать правильные вопросы нужным людям и в нужное время. Если процесс коснулся не всех заинтересованных лиц или им не были заданы правильные вопросы, это усугубляет проблему изменений, так как нет понимания истинных требований к системе, и в ходе разработки приходится производить значительные изменения, которых можно было бы избежать, если бы на более раннем этапе удалось добиться более полного понимания.
 - Нам не удалось создать практический процесс, позволяющий справиться с изменениями требований, которые являются нормой при пошаговой разработке. Возможно, мы пытались "заморозить" требования, и "латентные" необходимые изменения накапливались до тех пор, пока не "взорвались" перед лицом разработчиков и пользователей, вызвав переделки и стресс. Или процесс внесения изменений отсутствовал вовсе; все могли изменять все, что угодно и когда угодно. В таком случае на некотором этапе практически все оказывается измененным и невозможно понять, что к чему
- Просачивание требований
 - Упомянутые дистрибуторами улучшения, о которых программисты случайно услышали на переговорах.
 - Прямые запросы клиентов, обращенные к программистам.
 - Аппаратные функции, которые в итоге не вошли в продукт или не работают.
 - Изменение масштаба в ответ на действия конкурентов.
 - Функции, включенные программистом из "лучших побуждений" (в расчете, что это понравится клиенту).
 - "Сюрпризы" программистов.

Управление изменениями

1. Осознать, что изменения неизбежны, и разработать план управления изменениями.
2. Сформировать базовый уровень требований (requirements baseline).
3. Установить единый канал контроля изменений.
4. Использовать систему контроля изменений для их фиксации.
5. Обрабатывать изменения по иерархическому принципу.

Прослеживаемость требований — возможность установления связей между требованиями и их источниками, с одной стороны, а также с разделами и элементами возникающими при разработке текстов программ, документов и моделей, с другой стороны. Основано, чаще всего, на выделении элементарных требований и присвоении им идентификаторов. Каждое требование должно иметь уникальный идентификатор, который позволяет проследивать его разработку на протяжении всего жизненного цикла. В рабочих продуктах, которые появляются на более поздних этапах жизненного цикла, таких как план тестирования, каждая ссылка на свойство системы должна прослеживаться до определения и спецификации требований. Матрица прослеживаемости требований, является отличным инструментальным средством для решения упомянутой задачи.

Обеспечение прослеживаемости — определяет процедуры прослеживания требований в документах, моделях и коде, вырабатываемых при создании системы.

Анализ результатов изменений затрагивает 3 аспекта

1. Определите возможные последствия изменения. Часто они вызывают значительный волновой эффект. Включение множества функций в продукт может снизить его производительность до неприемлемого уровня, например, когда системе, запускаемой ежедневно, потребуется 24 часа для завершения одного запуска.
2. Определите все файлы, модели и документы, которые, возможно, придется изменить, если команда включит все запрошенные изменения.
3. Определите задачи, необходимые для реализации изменения, и оцените усилия, необходимые для выполнения этих задач

Трассируемость - способность соотнести какой-либо элемент проекта с другим, связанным с проектом элементом, особенно с теми, которые имеют отношение к техническим требованиям.

Трассируемость служит 2-м основным целям

1. Обеспечению качества продукта через
 - a. приданье ему всех тех возможностей, которые затребовал заказчик, а также обязательное отсутствие тех качеств, которые не были им запрошены ("создание правильного продукта").
 - b. Другой аспект качества заключается в обеспечении правильного функционирования всех затребованных возможностей ("Построение правильного продукта").
2. Анализу влияния изменений через определение затронутых требований, проектных решений и артефактов реализации, а также связанных с ними тестов.

Процедура трассирования

1. выбирается элемент (функция, фрагмент или некоторая часть) из матрицы трассирования требований, за которым проводится прослеживание на этапах ЖЦ;
2. составляется список вопросов, по которым на каждом этапе проверяются связи при реализации требований в продукте, и если изменяется какое-то звено в цепочке требований, то может модифицироваться процедура разработки этого элемента на последующем этапе ЖЦ;
3. проводится мониторинг статуса каждого требования на соответствие выполнения согласно принятому плану;
4. уточнение ресурсов выполнения проекта при необходимости проведения изменений в требования и в элементы проекта

3.11 Средства автоматизации управления требованиями.

Средства автоматизации управления требованиями представляют собой программные инструменты, которые помогают в сборе, анализе, организации и отслеживании требований в процессе разработки программного обеспечения. Они облегчают работу с требованиями, улучшают коммуникацию между участниками проекта и помогают контролировать изменения в требованиях.

Некоторые из средств автоматизации управления требованиями включают:

- Системы управления требованиями (Requirements Management Systems): Программное обеспечение, которое позволяет создавать, организовывать, отслеживать и управлять требованиями на протяжении жизненного цикла проекта. Примеры включают IBM Rational DOORS, Jama Connect, и Atlassian Jira.
- Средства моделирования и анализа требований: Инструменты, позволяющие визуализировать требования, создавать диаграммы, проводить анализ требований на соответствие и последствия изменений. Например, Enterprise Architect, Lucidchart.
- Системы отслеживания изменений (Version Control Systems): позволяют отслеживать изменения в требованиях, управлять версиями и историей изменений. Например, Git, SVN.
- Инструменты для тестирования требований (Requirements Testing Tools): Предназначены для проверки соответствия требований реальной функциональности системы. Например, HP Quality Center, TestRail.

Выбор средств автоматизации зависит от потребностей проекта, предпочтений команды разработки и особенностей требований. Они помогают оптимизировать процессы управления требованиями, улучшить их отслеживаемость и согласованность, что важно для успешной разработки программного обеспечения.

Системы управления требованиями (СУТ):

1. RequisitePro,
2. Enterprise Architect,
3. Sybase PowerDesigner.

В основном на проектах пользуются такими СУТ: Excel Word, Polarion, Enterprise Architect, TFS, Wiki, Caliber, Jira.

3.12 Системное моделирование для разработки требований. Модели структурного и объектно-ориентированного анализа.

Структурным анализом принято называть метод исследования системы, который начинается с ее общего обзора и затем детализируется, приобретая иерархическую структуру со все большим числом уровней. Основой подхода чаще всего являются методики описания бизнес-процессов с помощью диаграмм потоков данных (DFD), диаграмм структурного анализа и проектирования (SADT), диаграмм, изображающих потоки работ (IDEF3), а также диаграмм сущность-связь для моделирования данных.

Базовыми принципами структурного подхода являются:

- принцип "разделяй и властвуй";
- принцип иерархического упорядочения, когда составные части системы организуются в иерархические древовидные структуры.

В настоящее время наибольшее распространение получили методологии:

- SADT (Structured Analysis and Design Technique),
- структурного системного анализа Гейна-Сарсона (Gane-Sarson),
- структурного анализа и проектирования Йодана Де Марко (Yourdon/DeMarko),
- развития систем Джексона (Jackson), и т.д.

Несмотря на достаточно широкий спектр используемых методов и диаграммных техник, большинство методологий базируется на следующей "классической" совокупности:

- диаграммы потоков данных в нотации Йодана Де Марко или Гейна-Сарсона, обеспечивающие анализ требований и функциональное проектирование информационных систем;
- расширения Хатли и Уорда-Меллора для проектирования систем реального времени, основанные на диаграммах переходов состояний, таблицах и деревьях решений, картах и схемах потоков управления;
- диаграммы "сущность-связь" (в нотации Чена или Баркера);
- структурные карты Джексона и/или Константайна для проектирования межмодульных взаимодействий и внутренней структуры модулей, позволяющие развить модель анализа, построенную на базе перечисленных средств, до модели реализации.

Объектно-ориентированный анализ – это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

В процессе объектно-ориентированного анализа основное внимание уделяется определению и описанию объектов (или понятий) в терминах предметной области. В процессе объектно-ориентированного проектирования определяются логические программные объекты, которые будут реализованы средствами объектно-ориентированного языка программирования. Эти программные объекты включают в себя атрибуты и методы. И, наконец, в процессе конструирования или объектно-ориентированного программирования обеспечивается реализация разработанных компонентов и классов.

Рассмотрим основные принципы объектно-ориентированного анализа и проектирования.

- Сначала производится анализ требований, во время которого выделяются основные процессы, происходящие в моделируемой системе, и их формулировка в виде прецедентов.
- Шаг второй. Объектно-ориентированный анализ предметной области. Задача этого шага в определении видов деятельности участников процесса и составлении концептуальной модели, которая отражает различные категории элементов предметной области.
- Шаг третий. Разбираемся, кто, чем занимается. Эта деятельность называется объектно-ориентированным проектированием, при котором основное внимание сосредоточено на распределении обязанностей. Распределение обязанностей означает выделение задач и обязанностей различных программных объектов в приложении.

Наиболее важным моментом объектно-ориентированного анализа и проектирования является квалифицированное распределение обязанностей между компонентами программной системы. Обязанности объектов и их взаимодействия изображаются с использованием диаграмм классов и диаграмм взаимодействий.

ТЕМА 4. ОСНОВЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНЫХ СИСТЕМ

4.1 Стратегии и методы проектирования программного обеспечения

- Цель проектирования
- Виды проектирования
 - Функционально-ориентированное или структурное проектирование
 - Объектно-ориентированное проектирование
 - Компонентное проектирование
- Какие вопросы включает в себя проектирование (по SWEBOK)

Цель проектирования

Целью проектирования ПО является определение основных шагов, необходимых для разработки программы, включая этапы анализа, проектирования, реализации и тестирования.

В ходе проектирования вырабатывается подробный план того, как система будет функционировать, какие компоненты в нее входят, и как они взаимодействуют между собой. Целью является не только создание работоспособной программы, но и обеспечение ее эффективности, надежности, масштабируемости и удобства использования.

Проектирование также учитывает факторы, такие как оптимизация ресурсов, управление изменениями и обеспечение гибкости системы для будущего развития.

Главная цель успешного проектирования ПО – создать ПО, которое удовлетворяет всем функциональным и нефункциональным требованиям заказчика.

Виды проектирования

• Функционально-ориентированное или структурное проектирование

Это один из классических методов проектирования, в котором декомпозиция сфокусирована на идентификации основных программных функций и, затем, детальной разработке и уточнении этих функций “сверху-вниз”.

Структурное проектирование, обычно, используется после проведения структурного анализа с применением диаграмм потоков данных (DFD) и связанным описанием процессов.

Структурное проектирование позволяет написать код легкий для понимания и изменений, но при этом могут возникнуть трудности с внесением изменений (может потребоваться пересмотр всей структуры, что может быть неэффективно).

• Объектно-ориентированное проектирование

Представляет собой множество методов проектирования, базирующихся на концепции объектов. Данная область основывается на понятиях объекта (сущности), метода (действия) и атрибута (характеристики). Здесь главную роль играют полиморфизм и инкапсуляция. Объектно-ориентированное проектирование строится на абстракции данных, к которым добавлены поведенческие характеристики.

Преимуществами объектно-ориентированной архитектуры являются высокая тестируемость систем, расширяемость, способность к повторному использованию.

• Компонентное проектирование

Программные компоненты являются независимыми единицами, которые обладают однозначно-определенными (well-defined) интерфейсами и зависимостями (связями) и могут собираться и развертываться независимо друг от друга. Данный подход призван решить задачи использования, разработки и интеграции таких компонент с целью повышения повторного использования активов (как архитектурных, так и в форме кода).

Компонентно-ориентированное проектирование является одной из наиболее динамично развивающихся концепций проектирования и может рассматриваться как предвестник и основа сервисно-ориентированного подхода (Service-Oriented Architecture, SOA) в проектировании.

Какие вопросы включает в себя проектирование (по SWEBOK)

Параллелизм

Контроль и обработка событий

Распределение компонентов

Обработка ошибок и исключительных ситуаций и обеспечение отказоустойчивости

Взаимодействие и представление

Сохраняемость данных

4.2 Ключевые понятия архитектуры ПО

- Декомпозиция
- Компоненты
- Каркасы (Фреймворки)
- Шаблоны
- Многослойное представление
- Уровни

Декомпозиция

Декомпозиция в архитектуре программного обеспечения - это процесс разбиения сложной системы на более простые и управляемые компоненты. Этот процесс является ключевым элементом проектирования архитектуры ПО и позволяет справляться с сложностью, обеспечивать четкость структуры и повышать модульность системы.

Декомпозиция может быть выполнена по различным критериям, например, по функциональности, по компонентам, по модулям, по задачам или по времени.

Декомпозиция может проводиться на протяжении всего жизненного цикла проекта, особенно если проект является длительным и сложным. По мере того, как проект развивается, новые задачи могут появляться, требования могут изменяться, и декомпозиция может потребоваться для уточнения, изменения и дополнения структуры проекта.

Декомпозиция проекта должна проводиться до того уровня, на котором каждый элемент может быть реализован, протестирован и управляем. Оптимальный уровень декомпозиции зависит от размера, сложности и характера проекта, а также от опыта и квалификации команды разработчиков.

Компоненты

Это программный объект, спроектированный так, чтобы удовлетворять следующим требованиям:

1. Компонент допускает повторное использование в различных системах.
2. Компонент не хранит информации, специфичной для конкретного ПО, в котором он используется.
3. Допускается создание новых компонентов на основе существующих.
4. Компонент имеет известный интерфейс для взаимодействия, но скрывает детали своей внутренней реализации.
5. Компоненты проектируются так, чтобы иметь минимальные зависимости от других компонентов.

Типичным примером компонентов являются элементы пользовательского интерфейса (элементы управления). Компонент поставляется пользователю как двоичный код, скомпилированный, скомпонованный и готовый к использованию. Доступ к этому коду осуществляется через точно документированный интерфейс. Во время выполнения компоненты подключаются к другим компонентам, формируя программу.

Каркасы (Фреймворки)

Фреймворк (англ. framework - структура, каркас) - совокупность решений по архитектуре, структуре и способам объединения компонентов системы, которые могут быть применены для некоторого множества однотипных задач.

В области программирования под фреймворком понимают множество классов и способов их взаимодействия.

Фреймворк описывает начальную заготовку и свод инструкций, которые упрощают достижение конечной цели, написания программы.

Фреймворки подходят для решения многих задач: автоматизации, создания MVP (англ. minimum viable product - «минимально жизнеспособный продукт») или проверки идей.

Примеры: Django, Flask, React Native (JavaScript).

Шаблоны

Шаблон - общая, повторно используемая структура или организация компонентов, которую можно применять для проектирования различных программных систем. Шаблоны архитектуры предоставляют общие принципы для организации системы, облегчая повторное использование кода, стандартизацию и создание согласованных архитектурных решений.

Примеры шаблонов архитектуры:

- MVC (Model-View-Controller) - Разделяет систему на три основных компонента: Модель (данные и бизнес-логика), Представление (отображение данных пользователю) и Контроллер (управление взаимодействием между Моделью и Представлением);
- Microservices - Разделяет систему на небольшие, автономные и независимые службы (микросервисы), каждая из которых выполняет ограниченные функции и взаимодействует с другими микросервисами через сетевые вызовы;
- Client-Server - Разделяет систему на две основные части - клиентскую (интерфейс пользователя) и серверную (обработка данных и бизнес-логика). Клиент и сервер обмениваются данными через сетевое взаимодействие;
- и другие.

Шаблоны обеспечивают архитектурные принципы и рекомендации для создания сбалансированных и эффективных систем.

Многослойное представление

Многослойная архитектура обеспечивает группировку связанный функциональности приложения в разных слоях, выстраиваемых вертикально, поверх друг друга. Функциональность каждого слоя объединена общей ролью или ответственностью. Слои слабо связаны, и между ними осуществляется явный обмен данными. Правильное разделение приложения на слои помогает поддерживать строгое разделение функциональности, что в свою очередь, обеспечивает гибкость, а также удобство и простоту обслуживания.

Каждый слой может взаимодействовать с любым другим слоем, и информация может передаваться между слоями в разных направлениях.

Слои могут зависеть друг от друга, и изменения в одном слое могут повлиять на другие слои.

Уровни

Под уровнями архитектуры программного обеспечения понимаются логически разделенные блоки, каждый из которых включает в себя одну или несколько дисциплин, учитывать которые необходимо для достижения оптимального результата функционирования программного продукта.

Архитектура программного продукта состоит из набора иерархически подчиненных уровней, каждый из которых описывает отдельные аспекты информационной системы.

Следует выделить следующие уровни:

- Уровень приложения;
- Уровень данных;
- Уровень информации;
- Интеграционный уровень;
- Уровень безопасности;
- Сетевой уровень;
- Уровень платформы;
- Уровень системы.

Каждый уровень взаимодействует с ближайшими соседними уровнями, и обычно информация передается от одного уровня к другому.

Уровни могут быть разработаны и модифицированы независимо друг от друга, что способствует легкости сопровождения и изменений.

4.3 Ключевые вопросы проектирования

- Параллелизм
- Контроль и обработка событий
- Распределение компонентов
- Обработка ошибок и исключительных ситуаций и обеспечение отказоустойчивости
- Взаимодействие и представление
- Сохраняемость данных

Параллелизм

Параллелизм в проектировании программного обеспечения означает возможность выполнять несколько задач одновременно. Это может быть достигнуто путем использования многозадачности, многопоточности или распределенных вычислений.

Эта тема охватывает вопросы, подходы и методы организации процессов, задач и потоков для обеспечения эффективности, атомарности, синхронизации и распределения (по времени) обработки информации.

Контроль и обработка событий

Контроль и обработка событий в контексте программного обеспечения означают механизмы, которые позволяют системе реагировать на различные события, такие как действия пользователя, изменения внутреннего состояния или системные уведомления. Это важная часть архитектуры программы, которая обеспечивает адекватное взаимодействие с окружающей средой и пользователями.

Данная тема касается и неявных методов обработки событий, часто реализуемых в виде функции обратного вызова (call-back), как одной из фундаментальных концепций обработки событий.

Распределение компонентов

Распределение компонентов означает размещение различных частей приложения на различных узлах или серверах, что может быть особенно важным для распределенных систем.

Один из важнейших вопросов данной темы – использование связующего программного обеспечения (middleware).

Обработка ошибок и исключительных ситуаций и обеспечение отказоустойчивости

Обработка ошибок и исключительных ситуаций включает в себя механизмы для обнаружения, обработки и управления ситуациями, которые могут привести к некорректной работе программы. Отказоустойчивость, в свою очередь, означает способность системы продолжать корректную работу даже в условиях сбоев или непредвиденных обстоятельств.

Вопрос темы – как предотвратить сбои или, если сбой все же произошел, обеспечить дальнейшее функционирование системы.

Взаимодействие и представление

Взаимодействие и представление касаются того, как пользователи взаимодействуют с системой и как данные представляются пользователю.

Тема касается вопросов представления информации пользователям и взаимодействия пользователей с системой, с точки зрения реакции системы на действия пользователей. Речь в этой теме идет о реакции системы в ответ на действия пользователей и организации ее отклика с точки зрения внутренней организации взаимодействия, например, в рамках популярной концепции Model-View-Controller.

Сохраняемость данных

Тема касается не доступа к базам данных, как такового, а также не гарантий сохранности информации. Суть вопроса – как должны обрабатываться “долгоживущие” данные.

4.4 Горячие точки архитектуры

- Ключевые вопросы
- Ключевые проблемы

Ключевые вопросы

Аутентификация и авторизация

Как хранить идентификационные данные зарегистрированных пользователей?

Как аутентифицировать запросы?

Как авторизовывать запросы?

Как передавать идентификационные данные пользователей между слоями приложения?

Кэширование и состояние

Как выбрать эффективную стратегию кэширования?

Как увеличить производительность с помощью кэширования?

Как улучшить безопасность и стабильность с помощью кэширования?

Как поддерживать кэш в актуальном состоянии?

Как определить, когда и каким образом использовать специализированный кэш?

Как определить какие данные кэшировать и куда?

Как определить правила удаления данных из кэша из-за устаревания или переполнения?

Как загружать данные кэша?

Как следить за кэшем?

Как синхронизовать между собой множественные кэши?

Как определить какой способ кэширования даёт наибольшую производительность и масштабируемость для определённого сценария и конфигурации.

Как определить какая технология кэширования удовлетворяет требованиям приложения насчёт безопасности, лёгкости управления и мониторинга.

Взаимодействие

Как передавать информацию между слоями приложения?

Как совершать асинхронные операции?

Как передавать секретные данные?

Композиция

Как спроектировать структуру приложения?

Как спроектировать слабую связанность между модулями?

Как работать с зависимостями в слабосвязанном режиме?

Управление конфигурацией

Как определить какая информация подлежит конфигурированию?

Как определить где и каким образом сохранять конфигурационную информацию?

Как работать с секретной информацией?

Как работать с конфигурационной информацией в кластере?

Связанность и сцепление

Как разделить функциональности?

Как структурировать приложение?

Как выбрать подходящее разбиение на слои?

Как задать границы между частями системы?

Доступ к данным

Как управлять соединениями БД?

Как управлять исключениями?

Как улучшить производительность?

Как улучшить управляемость?

Как работать с бинарными данными (BLOBs)?

Как использовать пейджинг (paging) для записей?

Как управлять транзакциями?

Работа с исключениями

Как обрабатывать исключения?

Как сохранять информацию об исключениях?

Ведение логов и мониторинг

Как определить какую информацию писать в лог?

Как сделать запись в лог настраиваемой?

Взаимодействие с пользователем

Как повысить эффективность выполнения задач?

Как улучшить отзывчивость интерфейса?

Как улучшить возможности приложения для пользователя?

Как улучшить внешний вид приложения?

Проверка данных

Как определить где и когда проводить валидацию данных?

Как проверять длину, диапазон, формат и тип данных?

Как ограничить вводимые данные и сообщать о неправильных значениях?

Как обезопасить выводимые данные?

Поток операций

Как решать проблемы параллельных вычислений в потоке операций?

Как обрабатывать ошибки в потоке операций?

Как координировать процессы в потоке операций?

Ключевые проблемы

Аутентификация и авторизация

Хранение идентификационных данных в открытом виде в файлах.

Передача идентификационных данных в открытом виде по сети.

Аккаунты с излишними привилегиями.

Долгоживущие сессии.

Смешивание персонализации и аутентификации.

Использование единственного контроллера доступа.

Отсутствие ограничений на доступ к системным ресурсам для приложения.

Отсутствие ограничений на доступ к БД кроме определённых хранимых процедур.

Недекватное разделение прав доступа.

Кэширование и состояние

Промахи кэша.

Отсутствие устаревания данных в кэше.

Плохая архитектура кэша.

Отсутствие синхронизации кэша, необходимого для хорошего масштабирования системы.

Взаимодействие

Повышенный сетевой трафик и задержки из-за «тяжёлых» сообщений между слоями.

Неудачные транспортные протоколы и форматы сообщений.

Слишком большие объёмы данных в сетях ограниченной пропускной способности.

Композиция

Тесно связанные модули

Дублированный код

Управление конфигурацией

Небезопасные интерфейсы администрирования.

Небезопасные конфигурационные хранилища.

Хранение конфигурационных данных в открытом виде.

Слишком много администраторов в системе и привилегий для аккаунтов.

Связанность и сцепление

Ограниченнная масштабируемость из-за тесной привязанности к серверу и ресурсам.
Смешивание уровней презентации и бизнес-логики, ограничивающее возможность масштабирования.
Проблемы с дальнейшей поддержкой из-за излишней связанности.

Доступ к данным

Использование отдельного логина/пароля для каждого пользователя когда в этом нет необходимости.
«Тяжёлые» запросы к БД.
Рассеянная по классам бизнес-логика.

Работа с исключениями

Оставление системы/приложения в нестабильном состоянии.
Открытие секретной информации конечным пользователям.
Использование исключений для работы логики.
Запись в логи недостаточной информации об исключении.

Ведение логов и мониторинг

Отсутствие логов и мониторинга.
Слишком высокая детализация логов и мониторинга.
Отсутствие настроек логов и мониторинга во время исполнения (run-time).
Отсутствие логов для критических бизнес-операций.

Взаимодействие с пользователем

Неэффективная поддержка задач пользователя.
Плохое время отклика.
Игнорирование пользователей с ограниченными возможностями.

Проверка данных

Фильтрация ошибочных данных лишь на уровне приложения.
Небезопасный вывод данных в HTML.
Небезопасное вставление данных в SQL.
Проверка данных лишь на стороне пользователя.
Использование имён файлов, URLов или имен пользователей для принятия решений насчёт безопасности.

Поток операций

Сильная связанность.
Негибкие процессы.

4.5 Этапы архитектурного проектирования

- Структурирование системы.
- Моделирование управления. Два подхода к управлению. Типы управления в программных системах.
 - Централизованного управления
 - Шаблон вызов-возврат
 - Шаблон менеджера (диспетчера)
 - Событийное управление
 - Паттерн широковещательного управления
 - Паттерн управления на основе прерываний
- Декомпозиция систем на модули.
 - Модульность
 - Связность. Уровни связности.
 - Сцепление. Основные типы сцепления.

Архитектурное проектирование включает в себя три типа деятельности: структурирование системы, моделирование управления, декомпозицию систем на модули.

Структурирование системы.

При структурировании системы система разбивается на несколько подсистем, где под подсистемой подразумевается независимый программный компонент. Определяется взаимодействие подсистем.

Подсистема – это 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.

система → подсистема → модуль

Структурные паттерны: Репозиторий, Клиент/сервер, Объектно - ориентированный, Модель предметной области (Domain Model), модуль таблицы (Data Mapper), Многоуровневая система (Layers) или абстрактная машина, Потоки данных (конвейер или фильтр).

В модели структуры системы показаны все подсистемы, из которых она состоит. Для того, чтобы подсистемы функционировали как единое целое, необходимо управлять ими.

Моделирование управления. Два подхода к управлению. Типы управления в программных системах.

При моделировании управления разрабатывается модель связей управления между частями системы. На этом этапе необходимо организовать подсистемы согласно некоторой модели управления, которая дополняла бы имеющуюся модель структуры.

Можно выделить два основных типа управления в программных системах:

- Централизованного управления (Одна подсистема выделяется как системный контроллер. Её обязанности – руководить работой других подсистем. Управление системой определяется значениями некоторых переменных ее состояния (бинарный сигнал).)

➤ Шаблон вызов-возврат

Из одной главной программы по мере необходимости вызываются подпрограммы. Каждая из этих подпрограмм в свою очередь может вызывать другие подпрограммы и т.д. За управление отвечает та подпрограмма, которая выполняется в текущий момент. По окончании выполнения программы управление передается программе, которая ее вызвала.

Данная модель применима только для последовательных систем. Последовательные системы легче проектировать и тестировать.

Недостаток модели состоит в сложной обработке исключительных ситуаций.

Вызов программных процедур осуществляется "сверху - вниз".

➤ Шаблон менеджера (диспетчера)

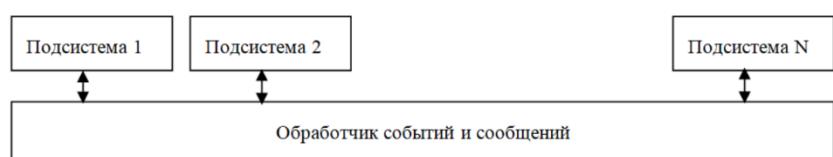
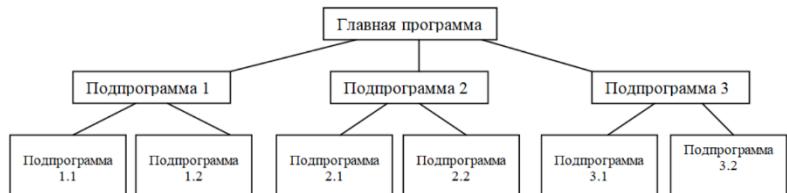
Применяется в параллельных системах. Часто используется в системах мягкого реального времени. Один системный компонент назначается диспетчером и управляет запуском, завершением и координированием других процессов системы. Процесс (выполняемая подсистема или модуль) может протекать параллельно с другими процессами. Модель такого процесса применима также в последовательных системах, где управляющая программа вызывает отдельные подсистемы в зависимости от значений некоторых переменных состояния. Обычно такое управление реализуется через оператор case.

- Событийное управление (Управление основано на внешних событиях. В данном контексте под событием подразумевается сигнал, который может принимать некоторый диапазон значений. За управление может отвечать любая подсистема)

➤ Паттерн широковещательного управления

В широковещательной модели каждая подсистема уведомляет обработчика о своем интересе к конкретным событиям. Когда событие происходит, обработчик посылает его подсистеме, которая может обработать это событие.

Преимуществом модели передачи сообщений является относительно простая модернизация систем. Новую подсистему можно интегрировать в подсистему, регистрируя ее события в обработчике событий.



Недостатком данной модели является то, что подсистемам неизвестно, когда произойдет обработка события. Генерируя событие, система не знает, какая именно система прореагирует на него. Вполне допустима ситуация, когда разные подсистемы реагируют на одинаковые события. Это может привести к конфликтам при получении доступа к результатам обработки события.

➤ Паттерн управления на основе прерываний

Для обеспечения быстрой реакции на события, необходимо использовать управление, основанное на прерываниях. Для каждого типа прерываний существует свой обработчик. Каждый тип прерывания ассоциируется с ячейкой памяти, в которой хранится адрес обработчика прерывания. При получении определенного прерывания аппаратный переключатель немедленно передает управление обработчику прерывания. В ответ на событие, вызванное прерыванием, обработчик может запустить или завершить другие процессы (это может определяться установленным приоритетом процессов).

Данная модель используется в жестких системах реального времени. Можно комбинировать эту модель с моделью централизованного управления. Центральный диспетчер обрабатывает нормальный ход выполнения системы, а в критических ситуациях используется управление, основанное на прерываниях.

Преимущество – мгновенная реакция системы на происходящие события.

Недостаток – сложность программирования и аттестации системы.

Декомпозиция систем на модули.

Каждая подсистема разбивается на модули. Определяются типы модулей и межмодульные соединения.

- Модульность

На этапе модульной декомпозиции рассматриваются две модели:

1. Объектно-ориентированная модель. Система состоит из набора взаимодействующих объектов. В такой модели модули представляют собой объекты с собственными состояниями и определенными состояниями.

2. Модель потоков данных. Система состоит из функциональных модулей, которые получают на входе данные и преобразуют их, некоторым образом, в выходные. Такой подход часто называют конвейерным.

Большая система должна быть расчленена на обозримые модули. При этом модули должны быть как можно более независимы (критерий сцепления - coupling), каждый модуль должен выполнять единственную функцию (критерий связности - cohesion).

Другими словами, модули должны быть примерно одинаковыми по размеру, обладать сильной внутренней связностью и иметь минимум взаимных пересечений (внешних связей).

- Связность. Уровни связности.

Критерий связности контролирует как действия в одном модуле связаны друг с другом.

Связность - это мера прочности соединения функциональных и информационных объектов внутри одного модуля.

Размещение сильно связанных объектов в одном и том же модуле уменьшает межмодульные взаимосвязи и взаимовлияния.

Специалисты выделяют следующие уровни связности: Функциональная ($CC=10$), Последовательная ($CC=10$), Информационная ($CC=9$), Процедурная ($CC=5$), Временная ($CC=3$), Логическая ($CC=1$), Случайная ($CC=0$).

В хорошем проекте связность каждого модуля является высокой.

- Сцепление. Основные типы сцепления.

Сцепление – мера взаимозависимости модулей по данным.

В хорошем проекте сцепления должны быть минимизированы, т.е. модули должны быть слабозависимыми (независимыми) настолько, насколько это возможно.

Слабое сцепление может быть достигнуто за счет комбинирования трех следующих способов действий:

1. удаления необязательных связей;
2. уменьшения количества необходимых связей;
3. упрощением необходимых связей.

Существуют 4 основных типа сцепления (оценивается силой сцепления СЦ):

1. нормальное сцепление ($СЦ=1$ (сцепление по данным), $СЦ=3$ (сцепление по образцу), $СЦ=4$ (сцепление по управлению));
2. сцепление по внешним ссылкам ($СЦ=5$)
3. сцепление по общей области ($СЦ=7$);
4. сцепление по содержимому ($СЦ=9$).

4.6 Архитектурные шаблоны

- Многоуровневый шаблон

- Клиент-серверный шаблон
- Ведущий-ведомый
- Каналы и фильтры
- Шаблон посредника
- Одноранговый шаблон
- Шина событий
- Модель-представление-контроллер

Многоуровневый шаблон

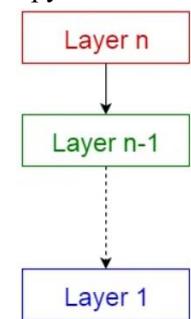
Данный шаблон используется для структурирования программ, которые можно разложить на группы неких подзадач, находящихся на определенных уровнях абстракции. Каждый слой предоставляет службы для следующего, более высокого слоя.

Чаще всего в общих информационных системах встречаются следующие 4 слоя:

- Слой представления (также известен как слой пользовательского интерфейса)
- Слой приложения (также известен как слой сервиса)
- Слой бизнес-логики (также известен как уровень предметной области)
- Слой доступа к данным (также известен как уровень хранения данных)

Использование:

- Общие десктопные приложения.
- Веб-приложения e-commerce.

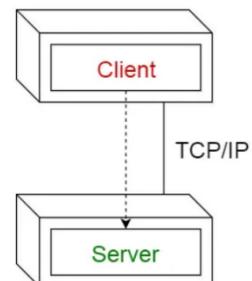


Клиент-серверный шаблон

Данный шаблон состоит из двух частей: сервера и множества клиентов. Серверный компонент предоставляет службы клиентским компонентам. Клиенты запрашивают услуги у сервера, а он, в свою очередь, оказывает эти самые услуги клиентам. Более того, сервер продолжает «подслушивать» клиентские запросы.

Использование:

- Онлайн приложения (электронная почта, совместный доступ к документам, банковские услуги).

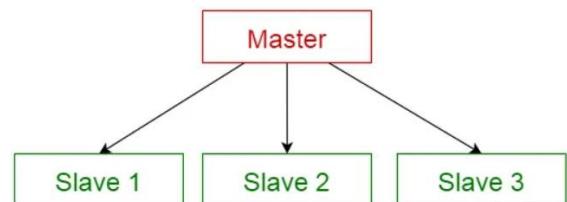


Ведущий-ведомый

В этом шаблоне также задействованы два участника — ведущий и ведомые. Ведущий компонент распределяет задачи среди идентичных ведомых компонентов и вычисляет итоговый результат на основании результатов, полученных от своих «подчиненных».

Использование:

- В репликации баз данных. Там главная БД считается авторитетным источником, а подчиненные базы с ней синхронизируются.
- Периферийные устройства, подключенные к шине в компьютере (ведущие и ведомые устройства).

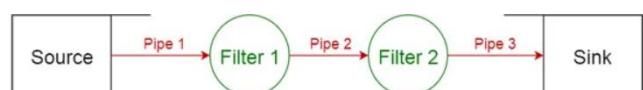


Каналы и фильтры

Этот шаблон подходит для систем, которые производят и обрабатывают потоки данных. Каждый этап обработки происходит внутри некоего компонента фильтра. Данные для обработки передаются через каналы. Эти каналы можно использовать для буферизации или синхронизации данных.

Использование:

- Компиляторы. Последовательные фильтры выполняют лексический, синтаксический, семантический анализ и создание кода.
- (Sink – приемник; pipe – канал)



Шаблон посредника

Данный шаблон нужен для структуризации распределенных систем с несвязанными компонентами. Эти компоненты могут взаимодействовать друг с другом через удаленный вызов службы. Компонент посредник отвечает за координацию взаимодействия компонентов.

Сервер размещает свои возможности (службы и характеристики) у посредника (брокера). Клиент запрашивает услугу у брокера. Затем брокер перенаправляет клиента к подходящей службе из своего реестра.

Использование:

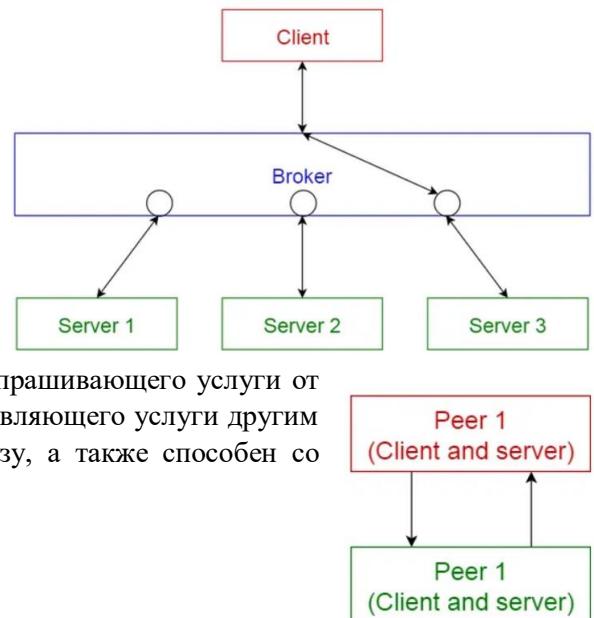
- Брокеры сообщений по типу Apache ActiveMQ, Apache Kafka, RabbitMQ и JBoss Messaging.

Одноранговый шаблон

В данном шаблоне существуют отдельные компоненты, так называемые пиры. Пиры могут выступать в роли как клиента, запрашивающего услуги от других равноправных участников (пирам), так и сервера, предоставляющего услуги другим пиром. Пир может быть клиентом или сервером, или всем сразу, а также способен со временем динамически изменять свою роль.

Использование

- Файлообменные сети (Gnutella и G2)
- Мультимедийные протоколы (P2PTV и PDTTP)
- Проприетарные мультимедийные приложения (как тот же Spotify).



Шина событий

Этот шаблон, в основном, взаимодействует с событиями и состоит из 4 главных компонентов: источник события, прослушиватель события, канал и шина событий. Источники размещают сообщения для определенных каналов нашине событий. Прослушиватели подписываются на определенные каналы. Прослушиватели получают уведомления о появлении сообщений, размещенных на каналах из их подписки.

Использование:

- Разработки на Android
- Сервисы уведомлений

Модель-представление-контроллер

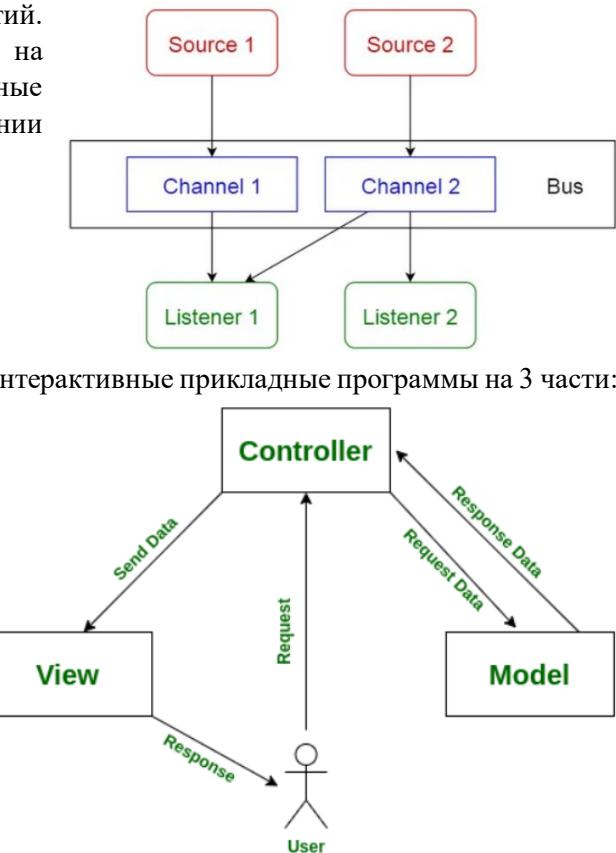
Этот шаблон также известен как MVC-шаблон. Он разделяет интерактивные прикладные программы на 3 части:

1. модель — содержит ключевые данные и функционал;
2. представление — показывает информацию пользователю (можно задавать более одного представления);
3. контроллер — занимается обработкой данных от пользователя.

Это делается с целью разграничения внутреннего представления информации от способов ее представления и принятия от пользователя. Данная схема изолирует компоненты и позволяет эффективно реализовать повторное использование кода.

Использование:

Архитектура WWW-приложений, Веб-фреймворки (Django и Rails).



ТЕМА 5. СТРУКТУРНЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

5.1 Понятие структурного анализа

- Принципы структурного анализа
- Классификация структурных методологий
- Модели AS-IS и TO-BE
- Средства структурного анализа

Структурным анализом принято называть метод исследования системы, который начинается с ее общего обзора и затем детализируется, приобретая иерархическую структуру со все большим числом уровней.

Основой структурного подхода чаще всего являются методики описания бизнес процессов с помощью диаграмм потоков данных (DFD), диаграмм структурного анализа и проектирования (SADT), диаграмм, изображающих потоки работ (IDEF3), а также диаграмм сущность-связь (ERD) для моделирования данных.

Структурный подход является функционально-ориентированным и рассматривает DFD-диаграммы в качестве движущей силы разработки проекта системы и в дальнейшем ее программной реализации. *Ориентация на функции означает, что в процессе разработки проекта системы выделяются функции, являющиеся условно независимыми частями системы. Это позволяет с одной стороны рассматривать их как независимые друг от друга части системы, а с другой стороны создавать из них более крупные фрагменты системы за счет взаимодействия функций.*

Позднее, в качестве одного из непосредственных результатов широкого распространения моделей реляционных баз данных, значение DFD-диаграмм в структурной разработке снизилось, и подход стал более ориентированным на данные, и, соответственно, акцент в разработке сместился на ERD-диаграммы.

Сочетание DFD- и ERD-диаграмм дает относительно полные модели анализа, которые фиксируют все функции и данные системы на требуемом уровне абстракции независимо от особенностей аппаратного и программного обеспечения. Затем модель анализа преобразуется в проектную модель, которая обычно выражается в понятиях реляционных баз данных. После этого следует этап реализации.

Структурный подход к анализу и проектированию отличается рядом особенностей, некоторые из них не очень хорошо увязываются с современными методами конструирования ПО:

- Подход скорее является последовательным и трансформационным, чем итеративным подходом с наращиванием возможностей (т.е. этот подход не способствует непрерывному процессу разработки, осуществляющему посредством итеративной детализации и пошаговой поставки ПО с наращенными возможностями).

- Подход направлен на поставку негибких решений, которые способны удовлетворить набор определенных бизнес-функций, но которые может быть трудно, масштабировать и расширять в будущем.

- Подход предполагает разработку «с чистого листа» и не поддерживает повторное использование уже существующих компонент.

Трансформационный характер структурного подхода является источником повышенного риска неправильно истолковать исходные требования пользователей по ходу разработки. Такой риск усиливается за счет необходимости постепенной замены относительно декларативной семантики моделей анализа на процедурные решения для проектных моделей и программного кода реализации.

Методы структурного анализа и проектирования стремятся преодолеть сложность больших систем путем расчленения их на части («черные ящики») и иерархической организации этих черных ящиков. Выгода в использовании черных ящиков заключается в том, что их пользователю не требуется знать, как они работают, необходимо знать лишь его входы и выходы, а также его назначение (т.е. функцию, которую он выполняет).

Второй важной идеей, лежащей в основе структурных методов, является идея иерархии. Для понимания сложной системы недостаточно разбиения ее на части, необходимо эти части организовать определенным образом, а именно в виде иерархических структур.

Наконец, третий момент: структурные методы широко используют графические нотации, также служащие для облегчения понимания сложных систем.

Принципы структурного анализа

Основной принцип управления любой сложной системой был известен давно: "divide et impera" - "разделяй и властвуй". Методы структурного анализа и проектирования стремятся преодолеть сложность больших систем путем расчленения их на части. Сложная программная система на верхнем уровне должна состоять из небольшого числа относительно независимых компонентов с четко определенными интерфейсами. Затем декомпозиции подвергаются выделенные на первом этапе компоненты, и так далее до заданного уровня детализации. Таким образом, система представляется иерархией с несколькими уровнями абстракции.

При этом автоматизируемая система сохраняет целостное представление, в котором все составляющие компоненты взаимоувязаны. При разработке системы "снизу-вверх" отдельных задач ко всей системе целостность теряется, возникают проблемы при информационной стыковке отдельных компонентов.

Таким образом, базовыми принципами структурного подхода являются:

- принцип "разделяй и властвуй"
- принцип иерархического упорядочения, когда составные части системы организуются в иерархические древовидные структуры.

Выделение двух базовых принципов не означает, что остальные принципы являются второстепенными, поскольку игнорирование любого из них может привести к непредсказуемым последствиям (в том числе и к провалу всего проекта). Основными из этих принципов являются следующие:

- принцип абстрагирования - заключается в выделении существенных аспектов системы и отвлечения от несущественных;
- принцип формализации - заключается в необходимости строгого методического подхода к решению проблемы;
- принцип непротиворечивости - заключается в обоснованности и согласованности элементов;
- принцип структурирования данных - заключается в том, что данные должны быть структурированы и иерархически организованы.

Классификация структурных методологий

Метод – совокупность приемов или операций для получения искомого результата.

Методология – совокупность методов, применяемых в какой-либо науке.

Роль методологии заключается в регламентации процесса разработки сложных систем. Она описывает последовательность шагов, модели и подходы, тщательное следование которым приведет к хорошо работающим системам.

Структурные методологии жестко регламентируют фазы анализа требований и проектирования спецификаций и отражают подход к разработке ПО с позиций рецептов "кулинарной книги". (Методология охватывает не только этап анализа, но и другие этапы. И в первую очередь этап проектирования)

В настоящее время наибольшее распространение получили методологии:

- SADT (Structured Analysis and Design Technique),
- структурного системного анализа Гейна-Сарсона,
- структурного анализа и проектирования Йодана Де Марко,
- развития систем Джексона,
- развития структурных систем Варнье-Оппа,
- анализа и проектирования систем реального времени Уорда-Меллора и Хатли,
- информационного моделирования Мартина.

Несмотря на достаточно широкий спектр используемых методов и диаграммных техник, большинство методологий базируется на следующей "классической" совокупности:

- диаграммы потоков данных в нотации Йодана Де Марко или Гейна-Сарсона, обеспечивающие анализ требований и функциональное проектирование информационных систем;
- расширения Хатли и Уорда-Меллора для проектирования систем реального времени, основанные на диаграммах переходов состояний, таблицах и деревьях решений, картах и схемах потоков управления;
- диаграммы "сущность-связь" (в нотации Чена или Баркера) или скобочные диаграммы Варнье-Оппа для проектирования структур данных, схем БД, форматов файлов как части всего проекта;
- структурные карты Джексона и/или Константайна для проектирования межмодульных взаимодействий и внутренней структуры модулей, позволяющие развить модель анализа, построенную на базе перечисленных средств, до модели реализации.

Современные структурные методологии по целевому использованию следует классифицировать:

1. на системы реального времени (СРВ)

2. на информационные системы (ИС)

Основная особенность систем реального времени заключается в том, что они контролируют и контролируются внешними событиями; реагирование на эти события во времени - основная и первоочередная функция таких систем. Система реального времени не должна обязательно быть быстрой, это распространенное заблуждение. Система реального времени должна выдавать реакцию в ответ на информацию, поступающую на ее вход в гарантированные промежутки времени.

Главные отличия информационных систем от систем реального времени приведены в таблице 1, средствами поддержки этих особенностей и различаются соответствующие структурные методологии.

Таблица 1

Информационные системы	Системы реального времени
Управляемы данными	Управляются событиями
Сложные структуры данных	Простые структуры данных
Большой объем входных данных	Малое кол-во входных данных
Интенсивный ввод/вывод	Интенсивные вычисления
Машинная независимость	Машинная зависимость

Необходимо отметить, что для проектирования систем реального времени используются специальные типы структурных диаграмм: диаграммы потоков управления, диаграммы переходов состояний, контекстные графы, матрицы состояний/событий, таблицы решений и др.

Информационно-управляющая система

Информационно-управляющая система (ИУС) – цифровая система контроля или управления некоторым реальным объектом, называемым обычно объект управлением. На вход ИУС поступает информация с датчиков, на выходе ИУС вырабатывается управляющее воздействие посредством исполнительного устройства.

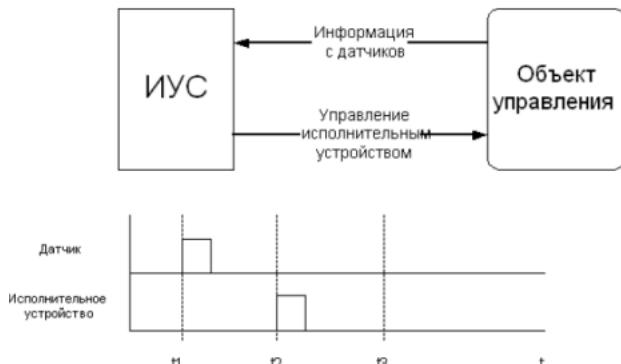


Рисунок 1. Взаимодействие ИУС и ОУ происходит в реальном масштабе времени

Информационно-управляющая система (или встроенная система) и система реального времени не являются синонимами. Отличие СРВ от ИУС состоит в том, что управляющая составляющая в СРВ не является обязательной. В класс СРВ может быть включена как информационно-управляющая система, так и система, относящаяся к информационным. Для примера к СРВ можно отнести компьютерные игры и системы резервирования авиабилетов.

Модели AS-IS и TO-BE

Концепции AS-IS (как есть) и TO-BE (как должно быть) используются в бизнес-анализе и управлении проектами для описания текущего состояния бизнес-процессов (AS-IS) и желаемого будущего состояния после внесения изменений (TO-BE).

AS-IS описывает текущие процессы, структуры, системы и методы работы в организации. Это детальное описание того, каким образом происходят бизнес-процессы на данный момент. Этот этап позволяет выявить проблемы, неэффективности, узкие места и области для улучшения. На этапе построения модели AS-IS важным считается строить максимально приближенную к действительности модель, основанную на реальных потоках процессов, а не на их идеализированном представлении.

TO-BE представляет собой желаемое состояние, которое компания стремится достичь после внедрения изменений. Это включает в себя предложения по улучшению процессов, внедрению новых технологий, оптимизации производственных процедур или изменениям в структуре управления.

Чтобы перейти от AS-IS к TO-BE, производится анализ и определение шагов, необходимых для достижения желаемого состояния. Этот процесс может включать в себя пересмотр бизнес-моделей, применение новых технологий, улучшение коммуникаций между отделами и изменения в структуре управления.

Средства структурного анализа

Для целей моделирования систем вообще, и структурного анализа в частности, используются три группы средств, иллюстрирующих:

- функции, которые система должна выполнять;
- отношения между данными;
- зависящее от времени поведение системы (аспекты реального времени).

Соответственно во всех методологиях структурного анализа используются три группы средств моделирования:

- диаграммы, иллюстрирующие функции, которые система должна выполнять, и связи между этими функциями. Для этой связи чаще всего используются DFD или SADT;
- диаграммы, моделирующие данные и их взаимосвязи ERD;
- диаграммы, моделирующие поведение системы в зависимости от времени (диаграммы потоков управления CFD, диаграммы переходов состояний и т.п.).



Рис. 2. Компоненты логической модели

Логическая DFD показывает внешние по отношению к системе источники и стоки (адресаты) данных, идентифицирует логические функции (процессы) и группы элементов данных, связывающие одну функцию с другой (потоки), а также идентифицирует хранилища (накопители) данных, к которым осуществляется доступ. Структуры потоков данных и определения их компонент хранятся и анализируются в словаре данных. Каждая логическая функция (процесс) может быть детализирована с помощью DFD нижнего уровня; когда дальнейшая детализация перестает быть полезной, переходят к выражению логики функции при помощи спецификации процесса (миниспецификации). Содержимое каждого хранилища также сохраняют в словаре данных, модель данных хранилища раскрывается с помощью ERD. В случае наличия реального времени DFD дополняется средствами описания, зависящего от времени поведения системы, раскрывающимися с помощью STD.

Перечисленные средства дают полное описание системы независимо от того, является ли она существующей или разрабатываемой с нуля. Таким образом, строится логическая функциональная спецификация – подробное описание того, что должна делать система, освобожденное насколько это возможно от рассмотрения путей реализации. Это дает проектировщику четкое представление о конечных результатах, которые следует достигать.

5.2 Анализ требований. Модели анализа: информационная, функциональная, поведенческая. Общая модель требований к системе. Структура модели требований.

○ Функциональные модели

- DFD – диаграммы потоков данных. Элементы DFD – модели. Контекстная диаграмма. Проектирование сложных ИС. Построение модели. Описание логики процесса (миниспецификация). Словарь данных. Методы задания спецификаций процессов
- SADT – диаграммы моделирования бизнес-процессов. Состав функциональной модели. Построение иерархии диаграмм. Типы связей между функциями
- IDEF3 – диаграммы потоков работ. Назначение IDEF3. Два типа диаграмм в IDEF3. Элементы IDEF3. Единицы работы, связи, перекрестки, объект ссылки. Типы объектов ссылок.

○ Модели данных. Типы сущностей. Мощность связи. Типы связей. Ключи. Правила ссылочной целостности. Типы сущностей и иерархии наследования. Нормализация данных. Прямое и обратное проектирование. Расширенные функции: UDP, триггеры и хранимые процедуры, скрипты

○ Модели управления. Взаимодействие моделей обработки данных и управления. Средства описания управляющих действий. Диаграммы переходов состояний (STD).

На этапе анализа требований необходимо понять, что предполагается сделать и задокументировать это. Для этого строится модель среды (Environmental Model), включающая:

- анализ поведения системы: определение назначения ИС, построение начальной контекстной диаграммы потоков данных (DFD) и формирование матрицы списка событий (ELM), построение функциональных диаграмм;
- анализ данных: определение состава потоков данных и построение диаграмм структур данных или схем данных (DSD), конструирование логической модели данных в виде ER-диаграммы.

Функциональные модели

Функциональные модели позволяют описать систему с точки зрения выполняемых ею функций.

Диаграммы потоков данных DFD - основной элемент структурного анализа – графическое средство для изображения информационных потоков и преобразований, которым подвергаются данные от входа к выходу системы.

Метод DFD основан на использовании трех взаимосвязанных спецификаций:

- *диаграмм потоков данных*
- *словаря данных*
- *спецификации процессов*

Основной элемент структурного анализа – диаграмма потоков данных.

Элементы DFD-модели:

Процесс (process) – преобразование входных потоков данных в выходные в соответствии с определенным алгоритмом. Идентификатором служит номер процесса. В поле имени вводится наименование в виде предложения с активным глаголом в неопределенной форме, за которым следуют существительные в винительном падеже.

Системы и подсистемы. Модель сложной ИС может быть представлена в виде одной системы, как единого целого, либо декомпозирована на ряд подсистем. Идентификатором служит номер подсистемы. В поле имени вводится наименование в виде предложения с подлежащим и соответствующими определениями и дополнениями.

Внешний объект (external entity) – Материальный предмет или физическое лицо, представляющее источник или приемник информации. Находится за пределами границ анализируемой ИС. Обозначается квадратом с тенью над диаграммой.

Накопитель (store) – Устройство хранения информации. Прообраз будущей базы данных, описание данных должно быть связано с информационной моделью. Идентифицируется буквой "D" и произвольным числом.

Поток (flow) – информация, передаваемая от источника к приемнику. Изображается линией со стрелкой – направление потока. Каждый поток имеет имя, стрелки описывают движение объектов.

Контекстная диаграмма

Первым шагом при построении иерархии DFD является построение контекстных диаграмм, где изображаются внешние сущности (терминаторы) – окружение системы, множество объектов, взаимодействующих с системой, но не входящих в ее состав (другие системы, пользователи, каналы информационного обмена, ...).

Проектирование относительно простых ИС.

На контекстной диаграмме система представляется в виде единственного процесса, связанного с внешними сущностями потоками данных. Диаграмма представляет требования к системе на самом верхнем уровне – уровне взаимодействия с окружением.

Проектирование сложных ИС.

Для сложных ИС строится иерархия контекстных диаграмм. Диаграмма верхнего уровня содержит набор подсистем, соединенных потоками данных. Диаграммы следующего уровня детализируют контекст и структуру подсистем. Иерархия диаграмм определяет взаимодействие основных функциональных подсистем проектируемой ИС как между собой, так и с внешними входными и выходными потоками данных и внешними объектами (источниками и приемниками информации), с которыми взаимодействует ИС.

Построение модели

Цель построения иерархического множества DFD – сделать требования ясными и понятными, а также разбить на части с определенными отношениями. В соответствии процесс построения модели разбивается на следующие этапы:

- 1) Расчленение требований и их организация в функциональные группы;
- 2) Идентификация внешних объектов, с которыми система должна быть связана;
- 3) Идентификация основных видов информации между системой и внешними объектами;
- 4) Разработка контекстной диаграммы, где функциональные группы представляются процессами, внешние объекты – внешними сущностями, виды информации – потоками данных между процессами и внешними сущностями;
- 5) Изучение и изменение контекстной диаграммы по результатам ответов на вопросы по ее частям;
- 6) Построение контекстной диаграммы путем объединения всех процессов предварительной диаграммы в один и группирования потоков;
- 7) Формирование DFD первого уровня на базе процессов предварительной контекстной диаграммы;
- 8) Проверка основных требований по DFD первого уровня;
- 9) Декомпозиция каждого процесса DFD с помощью детализирующей диаграммы или спецификации процесса;
- 10) Проверка основных требований по DFD соответствующего уровня;
- 11) Добавление определений новых потоков в словарь данных при каждом их появлении на диаграммах;
- 12) Параллельное (с процессом декомпозиции) изучение требований, разбиение их на элементарные и идентификация процессов или спецификаций процессов, соответствующих этим требованиям;
- 13) После построения двух-трех уровней проведение ревизии с целью проверки корректности и улучшения понимаемости модели;
- 14) Построение спецификации процесса (а не простейшей диаграммы) в случае, если некоторую функцию сложно или невозможно выразить комбинацией процессов.

Миниспецификация – описание логики процесса – формулирует его основные функции, чтобы специалист, реализующий проект, мог их выполнить, либо разработать соответствующую программу.

Миниспецификация – конечная вершина иерархии ДПД.

Условия использования:

- наличие у процесса небольшого количества входных и выходных потоков данных (2-3 потока);
- возможности описания преобразования данных процессом в виде последовательного алгоритма;
- выполнения процессом единственной логической функции преобразования входной информации в выходную;
- возможности описания логики процесса при помощи миниспецификации небольшого объема (не более 20-30 строк).

Словарь данных – организованный список элементов данных системы с их точными определениями, что позволяет различным категориям пользователей иметь понимание входных и выходных потоков и компонент хранилищ.

Для каждого потока данных необходимо хранить имя, тип и атрибуты.

Типы:

- простые (элементарные) или групповые (комплексные);
- внутренние (внутри системы) или внешние (связывают систему с другими системами);
- потоки данных или потоки управления;
- непрерывные (любые значения в диапазоне) или дискретные (определенные значения).

Атрибуты:

- имена-синонимы потока данных в соответствии с узлами изменения имени;
- БНФ-определение в случае группового потока;
- единицы измерения потока;
- диапазон значений для непрерывного потока, его типичное значение и информация по обработке экстремальных значений;
- список значений и их смысл для дискретного потока;
- список номеров диаграмм различных типов, в которых поток встречается;
- список потоков, в которые данный поток входит (как элемент БНФ-определения);
- комментарий, включающий дополнительную информацию.

Спецификация процесса (СП) используется для описания функционирования процесса, когда отсутствует необходимость детализировать его с помощью DFD (если он невелик, и описание может занимать до одной страницы текста).

СП представляют собой алгоритмы описания задач, выполняемых процессами, и содержат номер и/или имя процесса, списки входных и выходных данных и тело (описание) процесса, являющееся спецификацией алгоритма или операции, трансформирующей входные потоки данных в выходные.

Требования СП:

- для каждого процесса нижнего уровня должна существовать только одна спецификация;
- спецификация должна определять способ преобразования входных потоков в выходные;
- нет необходимости определять метод реализации этого преобразования;
- спецификация должна стремиться к ограничению избыточности – не переопределять, что уже было определено на диаграмме или в словаре данных;
- набор конструкций для построения спецификации должен быть простым и стандартным.

Методы задания спецификации процессов

1) Структурированный естественный язык. Применяется для читабельного и строгого описания спецификаций процессов. Является комбинацией строгости языка программирования и читабельности естественного языка и состоит из подмножества слов, организованных в логические структуры, арифметических выражений и диаграмм.

Состав:

- глаголы, ориентированные на действие и применяемые к объектам;
- термины (задачи, процедуры, символы данных, ...);
- предлоги и союзы в логических отношениях;
- математические, физические и технические термины;
- арифметические уравнения;
- таблицы, диаграммы, графы, ...;
- комментарии.

Управляющие структуры языка:

- последовательная конструкция;
- конструкция выбора;
- итерация.

2) Таблицы и деревья решений (ТР). Используются, если действие зависит от нескольких переменных, которые в совокупности могут продуцировать большое число комбинаций.

Проектирование заключается в задании матрицы, отображающей множество входных условий в множество действий.

Состав: верхняя часть таблицы используется для определения условий, нижняя – для определения действий. Левая часть ТР содержит описание условий и действий, а в правой части перечисляются комбинации условий и указывается, какие действия в какой последовательности выполняются, когда определенная комбинация условий имеет место.

3) Визуальные языки проектирования спецификаций. Базируются на идеях структурного программирования и позволяют определять потоки управления с помощью иерархически организованных схем.

Один из наиболее известных – подход с использованием FLOW-форм. Каждый символ FLOW-формы имеет вид прямоугольника и может быть вписан во внутренний прямоугольник другого символа. Символы помечаются с помощью предложений на естественном языке или с использованием математической нотации.

SADT – диаграммы моделирования бизнес-процессов.

Методология SADT разработана Дугласом Россом. На ее основе разработана, в частности, известная методология IDEF0 (Icam DEFinition). Методология SADT представляет собой совокупность методов, правил и процедур, предназначенных для построения функциональной модели объекта какой-либо предметной области. Функциональная модель SADT отображает функциональную структуру объекта, т.е.

Состав функциональной модели.

Результатом применения методологии SADT является модель, которая состоит из диаграмм, фрагментов текстов и глоссария, имеющих ссылки друг на друга. Диаграммы - главные компоненты модели, все функции ИС и интерфейсы на них представлены как блоки и дуги. Место соединения дуги с блоком определяет тип интерфейса. Управляющая информация входит в блок сверху, в то время как информация, которая подвергается обработке, показана с левой стороны блока, а результаты выхода показаны с правой стороны. Механизм (человек или автоматизированная система), который осуществляет операцию, представляется дугой, входящей в блок снизу(рис.1.).

Одной из наиболее важных особенностей методологии SADT является постепенное введение все больших уровней детализации по мере создания диаграмм, отображающих модель.



Рис 1. Функциональный блок и интерфейсные дуги

Построение иерархии диаграмм.

Построение SADT-модели начинается с представления всей системы в виде простейшей компоненты - одного блока и дуг, изображающих интерфейсы с функциями вне системы. Поскольку единственный блок представляет всю систему как единое целое, имя, указанное в блоке, является общим. Это верно и для интерфейсных дуг - они также представляют полный набор внешних интерфейсов системы в целом.

Затем блок, который представляет систему в качестве единого модуля, детализируется на другой диаграмме с помощью нескольких блоков, соединенных интерфейсными дугами. Эти блоки представляют основные подфункции исходной функции. Данная декомпозиция выявляет полный набор подфункций, каждая из которых представлена как блок, границы которого определены интерфейсными дугами. Каждая из этих подфункций может быть декомпозирована подобным образом для более детального представления.

Во всех случаях каждая подфункция может содержать только те элементы, которые входят в исходную функцию. Кроме того, модель не может опустить какие-либо элементы, т.е., как уже отмечалось, родительский блок и его интерфейсы обеспечивают контекст. К нему нельзя ничего добавить, и из него не может быть ничего удалено.

Модель SADT представляет собой серию диаграмм с сопроводительной документацией, разбивающих сложный объект на составные части, которые представлены в виде блоков. Детали каждого из основных блоков показаны в виде блоков на других диаграммах. Каждая детальная диаграмма является декомпозицией блока из более общей диаграммы. На каждом шаге декомпозиции более общая диаграмма называется родительской для более детальной диаграммы.

Дуги, входящие в блок и выходящие из него на диаграмме верхнего уровня, являются точно теми же самыми, что и дуги, входящие в диаграмму нижнего уровня и выходящие из нее, потому что блок и диаграмма представляют одну и ту же часть системы.

Типы связей между функциями

Одним из важных моментов при проектировании ИС с помощью методологии SADT является точная согласованность типов связей между функциями. Различают по крайней мере семь типов связывания:

(0) Тип случайной связности: наименее желательный. Случайная связность возникает, когда конкретная связь между функциями мала или полностью отсутствует. Это относится к ситуации, когда имена данных на SADT-дугах в одной диаграмме имеют малую связь друг с другом.

(1) Тип логической связности. Логическое связывание происходит тогда, когда данные и функции собираются вместе вследствие того, что они попадают в общий класс или набор элементов, но необходимых функциональных отношений между ними не обнаруживается.

(2) Тип временной связности. Связанные по времени элементы возникают вследствие того, что они представляют функции, связанные во времени, когда данные используются одновременно или функции включаются параллельно, а не последовательно.

(3) Тип процедурной связности. Процедурно-связанные элементы появляются сгруппированными вместе вследствие того, что они выполняются в течение одной и той же части цикла или процесса.

(4) Тип коммуникационной связности. Диаграммы демонстрируют коммуникационные связи, когда блоки группируются вследствие того, что они используют одни и те же входные данные и/или производят одни и те же выходные данные.

(5) Тип последовательной связности. На диаграммах, имеющих последовательные связи, выход одной функции служит входными данными для следующей функции. Связь между элементами на диаграмме является более тесной, чем на рассмотренных выше уровнях связок, поскольку моделируются причинно-следственные зависимости.

(6) Тип функциональной связности. Диаграмма отражает полную функциональную связность, при наличии полной зависимости одной функции от другой. Диаграмма, которая является чисто функциональной, не содержит чужеродных элементов, относящихся к последовательному или более слабому типу связности.

IDEF3 – диаграммы потоков работ.

Назначение IDEF3.

IDEF3 является стандартом документирования технологических процессов, происходящих на предприятии, и предоставляет инструментарий для наглядного исследования и моделирования их сценариев. Сценарием называется описание последовательности изменений свойств объекта, в рамках рассматриваемого процесса (например, описание последовательности этапов обработки детали в цеху и изменение её свойств после прохождения каждого этапа). Исполнение каждого сценария сопровождается соответствующим документооборотом, который состоит из двух основных потоков:

1. документов, определяющих структуру и последовательность процесса (технологических указаний, описаний стандартов и т.д.),

2. документов, отображающих ход его выполнения (результатов тестов и экспертиз, отчетов о браке, и т.д.).

Для эффективного управления любым процессом, необходимо иметь детальное представление об его сценарии и структуре сопутствующего документооборота. Средства документирования и моделирования IDEF3 позволяют выполнять следующие задачи:

- Документировать имеющиеся данные о технологии процесса, выявленные, скажем, в процессе опроса компетентных сотрудников, ответственных за организацию рассматриваемого процесса.

- Определять и анализировать точки влияния потоков сопутствующего документооборота на сценарий технологических процессов.

- Определять ситуации, в которых требуется принятие решения, влияющего на жизненный цикл процесса, например, изменение конструктивных, технологических или эксплуатационных свойств конечного продукта.

- Содействовать принятию оптимальных решений при реорганизации технологических процессов.

- Разрабатывать имитационные модели технологических процессов, по принципу "КАК БУДЕТ, ЕСЛИ..."

Два типа диаграмм в IDEF3.

Существуют два типа диаграмм в стандарте IDEF3, представляющие описание одного и того же сценария технологического процесса в разных ракурсах:

1. Диаграмма «Описания Последовательности Этапов Процесса» (PFDD). Необходима для описания последовательности выполнения этапов в процессе. PFDD используются для визуализации последовательности шагов, которые происходят в рамках процесса. Они отображают этапы, действия, решения и переходы между ними.

Элементы: включают действия (работы), решения, и входы/выходы для каждого этапа. Также могут отображаться контекстные условия и альтернативные пути.

2. Диаграмма «Состояния Объекта и его Трансформаций в Процессе» (OSTN). Необходима для описания состояния объектов и изменения состояний во времени в рамках процесса. OSTN сосредотачиваются на изменениях состояний объекта или системы в процессе выполнения.

Элементы: объекты, состояния, переходы между состояниями и события, которые вызывают изменения состояний объекта. Также может включать условия для переходов между состояниями.

Элементы IDEF3.

Диаграмма IDEF3 PFD может состоять из отдельных и самостоятельных описательных блоков:

- блоки действия или работы,
- стрелки или связи,
- перекрёстки,
- объекты ссылок.

Действие или работа. В IDEF3 действия изображаются прямоугольниками с прямыми углами. Действия предпочтительнее именовать по тем же правилам, что и в модели IDEF0, то есть глаголом, обычно отображающим основной результат работы.

Каждому действию присваивается уникальный номер (идентификатор), который никогда не меняется. Обычно номер действия состоит из номера родительского действия и порядкового номера на текущей диаграмме.

Нотация IDEF3 позволяет декомпозировать (детализировать) действие многократно, т.е. включить в одну модель альтернативные описания процессов. Поэтому в номере действия стоит и порядковый номер декомпозиции родительского действия. Действия имеют входы и выходы, но не поддерживают управления и механизмы, как функции в нотации IDEF0.

Связи показывают существенные взаимоотношения между действиями. Все связи в IDEF3 односторонние, могут начинаться и заканчиваться на любой стороне блока. Обычно диаграммы IDEF3 стараются построить так, чтобы связи были направлены слева направо, сверху вниз. В IDEF3 различают три типа стрелок, изображающих связи

Изображение	Название	Назначение
→	Временное предшествование	Сплошная стрелка, связывающая единицы работ. Показывает, что работа-источник должна полностью закончиться прежде, чем работа-цель начнется. Во многих случаях завершение одного действия инициирует начало выполнения другого. Связь именуют так, чтобы была понята причина ее появления. Например, прежде чем найти вхождение текста в документе, необходимо его ввести (рис. 2). Временная шкала выполнения действий показана на рис. 3. Вертикальными линиями показано начало и окончание действий. Время окончания A1.1.1 и время начала A1.1.2 может совпадать, может не совпадать
→→	Объектный поток	Стрелка с двумя наконечниками. Применяется для описания того, что результатом выполнения исходного действия является некоторый объект, который необходим для выполнения конечного действия. Временная семантика объектных связей аналогична связям предшествования. Связь именуют так, чтобы четко определить передающийся объект. Например, файл является результатом выполнения действия A1.1.3 (рис. 4)
→→→	Нечеткое отношение	Пунктирная линия. Используется, когда невозможно описать связи с использованием предшественных или объектных связей. Значение такой связи должно быть четко определено с помощью названия и описания стрелки, так как связи такого типа сами по себе не предполагают никаких ограничений. Применение нечетких отношений: отображение задержки между действиями; отображение взаимоотношений между параллельно выполняющимися действиями. На рис. 5 показано нечеткое отношение между действиями "Вывод пользователю сообщения об ошибке" и "Обработка ошибки"

Перекрестки используются для отображения логики взаимодействия стрелок при слиянии и разветвлении. В IDEF3 стрелки могут сливаться и разветвляться только через перекрестки. Различают перекрестки для слияния и разветвления стрелок. Перекресток не может использоваться одновременно для слияния и для разветвления.

Различают синхронные и асинхронные соединения. Используются для изображения соответственно синхронных действий, т.е. начинающихся и заканчивающихся одновременно, и для изображения асинхронных действий.

Все перекрестки на диаграмме нумеруются, каждый номер имеет префикс J.

Обозначение	Наименование	Смысл в случае слияния стрелок (Fan-in Junction)	Смысл в случае разветвления стрелок (Fan-out Junction)
	Asynchronous AND	Все предшествующие процессы должны быть завершены	Все следующие процессы должны быть запущены
	Synchronous AND	Все предшествующие процессы завершены одновременно	Все следующие процессы запускаются одновременно
	Asynchronous OR	Один или несколько предшествующих процессов должны быть завершены	Один или несколько следующих процессов должны быть запущены
	Synchronous OR	Один или несколько предшествующих процессов завершены одновременно	Один или несколько следующих процессов запускаются одновременно
	XOR (Exclusive OR)	Только один предшествующий процесс завершен	Только один следующий процесс запускается

Объект ссылки. Объект ссылки в IDEF3 выражает некую идею, концепцию или данные, которые нельзя связать со стрелкой, перекрестком или работой. Объект ссылки изображается в виде прямоугольника, похожего на прямоугольник работы. В качестве имени можно использовать имя какой-либо стрелки с других диаграмм или имя сущности из модели данных. Объекты ссылки должны быть связаны с единицами работ или перекрестками пунктирующими линиями. Официальная спецификация IDEF3 различает три стиля объектов ссылок - безусловные, синхронные и асинхронные.

Типы объектов ссылок.

Тип объекта ссылки	Цель описания
ОБЪЕКТ	Описывает участие важного объекта в действии
ССЫЛКА	Инструмент циклического перехода (в повторяющейся последовательности действий), переход возможен как на действие текущей диаграммы, так и на действие любой другой, но не обязательно. Если все действия цикла присутствуют на текущей диаграмме, цикл может также изображаться стрелкой, возвращающейся на стартовое действие. ССЫЛКА может ссылаться и на перекресток
ЕДИНИЦА ДЕЙСТВИЯ	Применяется для многократного отображения на диаграмме одного и того же действия, т.е. действия, которое используется в процессе несколько раз, но не в цикле. В этом случае в первый раз действие создается как единица работы, а последующие его появления на диаграмме оформляются объектами ЕДИНИЦА ДЕЙСТВИЯ
ЗАМЕТКА	Используется для документирования важной информации общего характера, относящейся к изображеному на диаграмме. ЗАМЕТКА является альтернативой внесению текстового объекта в диаграмму
УТОЧНЕНИЕ	Используется для уточнения или более детального описания, изображенного на диаграмме. Обычно употребляется для детального описания логики ветвления у соединений

Модели данных. Типы сущностей. Мощность связи. Типы связей. Ключи. Правила ссылочной целостности. Типы сущностей и иерархии наследования. Нормализация данных. Прямое и обратное проектирование. Расширенные функции: UDP, триггеры и хранимые процедуры, скрипты

Есть два уровня представления модели - логический и физический. Логический уровень - это абстрактный взгляд на данные, на нем данные представляются так, как выглядят в реальном мире, и могут называться так, как они называются в реальном мире. Объекты модели, представляемые на логическом уровне, называются сущностями и атрибутами. Логическая модель данных может быть построена на основе другой логической модели, например, на основе модели процессов. Логическая модель данных является универсальной и никак не связана с конкретной реализацией СУБД.

Физическая модель данных, напротив, зависит от конкретной СУБД, фактически являясь отображением системного каталога. В физической модели содержится информация о всех объектах БД. Поскольку стандартов на объекты БД не существует (например, нет стандарта на типы данных), физическая модель зависит от конкретной реализации СУБД. Следовательно, одной и той же логической модели могут соответствовать несколько разных физических моделей. Если в логической модели не имеет значения, какой конкретно тип данных имеет атрибут, то в физической модели важно описать всю информацию о конкретных физических объектах - таблицах, колонках, индексах, процедурах и т. д.

Уровни логической модели

Различают три уровня логической модели, отличающихся по глубине представления информации о данных:

- Диаграмма сущность-связь (Entity Relationship Diagram, ERD) – представляет собой модель данных верхнего уровня. Она включает сущности и взаимосвязи, отражающие основные бизнес-правила предметной области. Такая диаграмма не слишком детализирована, в нее включаются основные сущности и связи между ними, которые удовлетворяют основным требованиям, предъявляемым к ИС;
- Модель данных, основанная на ключах (Key Based model, KB) – более подробное представление данных. Она включает описание всех сущностей и первичных ключей и предназначена для представления структуры данных и ключей, которые соответствуют предметной области;
- Полная атрибутивная модель (Fully Attributed model, FA) – наиболее детальное представление структуры данных: представляет данные в третьей нормальной форме и включает все сущности, атрибуты и связи.

Типы сущностей

независимая, если каждый ее экземпляр может быть однозначно идентифицирован без определения его отношений с другими сущностями.

- зависимая, если однозначная идентификация ее экземпляра зависит от его отношения к другой сущности
Независимые от идентификатора сущности

Имя сущности/Номер сущности

Служащий/44

Зависимые от идентификатора сущности

Имя сущности/Номер сущности

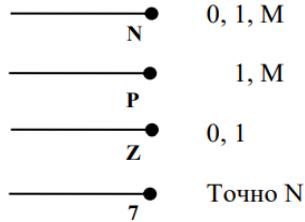
Проектное задание/56

имя	независимая сущность
имя	зависимая сущность

Мощность связи – количество экземпляров сущности-потомка, которое может существовать для каждого экземпляра сущности-родителя. Виды мощности связей:

- каждый экземпляр сущности-родителя может иметь ноль, один или более связанных с ним экземпляров сущности-потомка;
- каждый экземпляр сущности-родителя должен иметь не менее одного связанного с ним экземпляра сущности-потомка;
- каждый экземпляр сущности-родителя должен иметь не более одного связанного с ним экземпляра сущности-потомка;
- каждый экземпляр сущности-родителя связан с некоторым фиксированным числом экземпляров сущности-потомка.

Связь изображается линией, проводимой между сущностью-родителем и сущностью-потомком с точкой на конце линии у сущности-потомка (мощность – N).



Типы связей

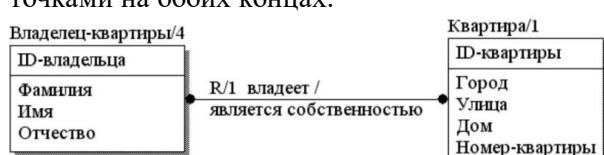
1) Идентифицирующая связь между сущностью-родителем и сущностью-потомком изображается сплошной линией. Сущность-потомок является зависимой от идентификатора сущностью. Сущность-родитель может быть как независимой, так и зависимой от идентификатора сущностью (определяется связями с другими сущностями).



2) Неидентифицирующая связь изображается пунктирной линией. Сущность-потомок независима от идентификатора, если не является сущностью-потомком в какой-либо идентифицирующей связи.



3) Неспецифическая связь (неконкретная, неопределенная, non-specific, многие-ко-многим) – связь, при которой экземпляр каждой сущности может быть связан с некоторым количеством экземпляров другой сущности. Мощность равна ноль-один или много-к-нулю- одному-или-многим. Изображается сплошной линией с точками на обоих концах.



Неспецифические связи могут быть полезны в начале работы над информационной моделью, когда сложно определить конкретные связи между экземплярами сущностей. На более поздних этапах неспецифическую связь

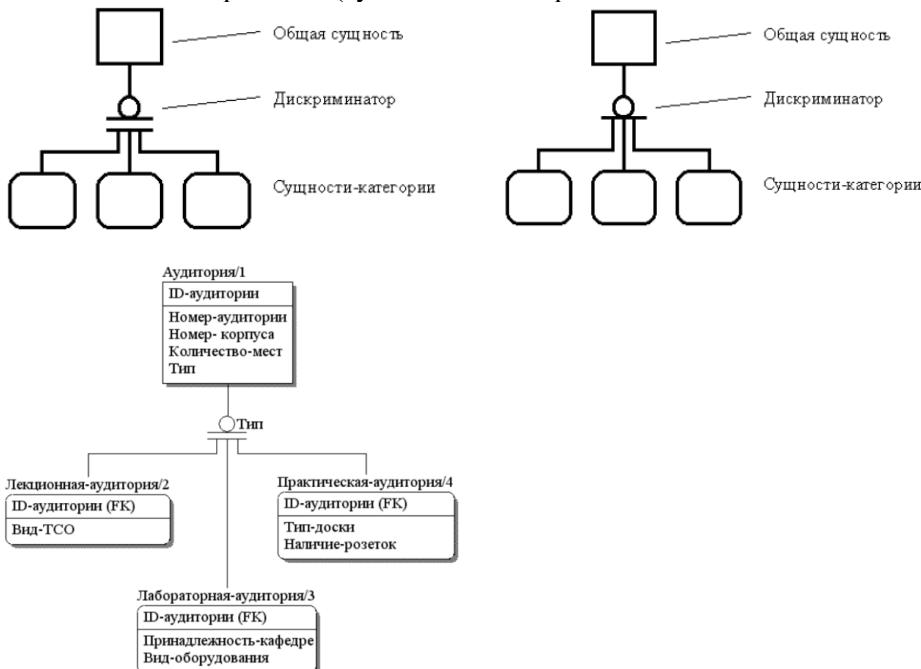
следует заменить на пару специфических соединительных связей. С этой целью создается третья – общая дочерняя сущность, связанная соединительными связями с двумя исходными сущностями.

4) Связи категоризации в IDEF1X. Понятие «отношение категоризации» по смыслу эквивалентно иерархической связи.

Виды отношений:

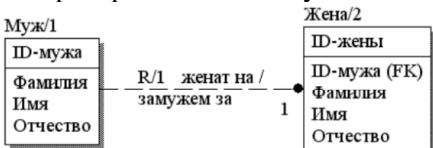
- полной категоризации (сущности-категории составляют полное множество потомков родительской сущности);

- неполной категоризации (сущности-категории составляют неполное множество потомков общей сущности).



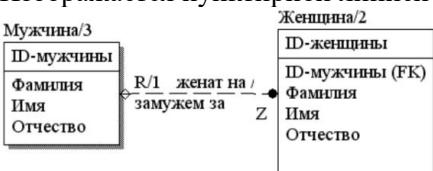
5) Обязательные и необязательные неидентифицирующие связи.

Связь обязательная (mandatory), если каждый экземпляр дочерней сущности связан точно с одним экземпляром родительской сущности.



Связь необязательная (optional), если каждый экземпляр дочерней сущности может быть связан с нулем или одним экземпляром родительской сущности.

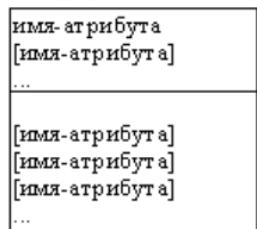
Изображается пунктирной линией с ромбом со стороны родительской сущности.



Ключи

Атрибуты изображаются списком имен внутри блока сущности. Определяющие первичный ключ размещаются наверху и отделяются от других атрибутов горизонтальной чертой.

Имя_Сущности/Номер_Сущности



Атрибуты
первичного
ключа

При выборе первичного ключа разработчики часто используют дополнительный (суррогатный) ключ, т.е. произвольный номер, который определяет запись в сущности.

Внешние ключи (Foreign Key, FK) – ключи, которые могут использоваться в качестве части или целого первичного ключа или неключевого атрибута.

Внешний ключ изображается с помощью помещения внутрь блока сущности имен атрибутов, после которых следуют буквы FK в скобках.

Пример внешнего ключа - неключевого атрибута

Брокер/12

номер брокера
номер отдела (FK)

Пример внешнего ключа - атрибута первичного ключа

Заявка-на-покупку/2

номер-заказа (FK)
номер-товара

Потенциальные ключи, которые не выбраны первичными, могут быть использованы в качестве вторичных или альтернативных ключей (АК). С помощью альтернативных ключей часто отображают различные индексы доступа к данным в конечной реализации реляционной базы.

Правила ссылочной целостности.

Правила ссылочной целостности – логические конструкции, которые выражают бизнес-правила использования данных и представляют собой правила вставки, замены и удаления.

При генерации схемы БД на основе опций логической модели, задаваемых в закладке Rolename/RI Actions, будут сгенерированы правила декларативной ссылочной целостности, которые должны быть предписаны для каждой связи, и триггеры, обеспечивающие ссылочную целостность. Триггеры представляют собой программы, выполняемые всякий раз при выполнении команд вставки, замены или удаления (INSERT, UPDATE или DELETE).

Пример 1. Существует идентифицирующая связь между сущностями Команда и Игрок. Что будет, если удалить команду? Экземпляр сущности Игрок не может существовать без команды, следовательно, нужно либо запретить удаление команды, пока в ней числится хотя бы один игрок, либо сразу удалять вместе с командой всех ее игроков. Такие правила удаления называются "ограничение" и "каскад". Выполнение команды на удаление одной строки реально может привести к удалению тысячи строк в БД, поэтому использовать правило удаления каскадом следует с осторожностью. В том случае, если установлено правило ограничения удаления, при попытке выполнить удаление команды, в которой есть хотя бы один игрок, сервер реляционной СУБД возвратит ошибку.

Пример 2. Есть необязательная неидентифицирующая связь между сущностями Отдел и Сотрудник. Экземпляр сущности Сотрудник может существовать без ссылки на отдел (атрибут внешнего ключа Где работает. Номер отдела может принимать значение NULL). В этом случае возможно установление правила установки в нуль - SET NULL. При удалении отдела атрибут внешнего ключа сущности Сотрудник - Где работает. Номер отдела примет значение NULL. Это означает, что при удалении отдела сотрудник остается работать в организации не будучи приписан к какому-либо отделу и информация о нем сохраняется.

Правила удаления управляют тем, что будет происходить в БД при удалении строки. Аналогично правила вставки и обновления управляют тем, что будет происходить с БД, если строки изменяются или добавляются. Например, можно установить правило, которое разрешает вносить новую Команду только в том случае, когда в нее зачислен хотя бы один Игрок.

ERwin автоматически присваивает каждой связи значение ссылочной целостности, устанавливаемой по умолчанию, прежде чем добавить ее в диаграмму. Режимы RI, присваиваемые ERwin по умолчанию, могут быть изменены в редакторе Referential Integrity Default, который вызывается, если щелкнуть по кнопке RI Defaults диалога Target Server (меню Server/Target Server).

Типы сущностей и иерархии наследования.

Связи определяют, является ли сущность независимой или зависимой. Различают несколько типов зависимых сущностей.

Характеристическая - зависимая дочерняя сущность, которая связана только с одной родительской и по смыслу хранит информацию о характеристиках родительской сущности.

Ассоциативная - сущность, связанная с несколькими родительскими сущностями. Такая сущность содержит информацию о связях сущностей.

Именующая - частный случай ассоциативной сущности, не имеющей собственных атрибутов (только атрибуты родительских сущностей, мигрировавших в качестве внешнего ключа).

Категориальная - дочерняя сущность в иерархии наследования.

Иерархия наследования (или иерархия категорий) представляет собой особый тип объединения сущностей, которые разделяют общие характеристики.

Обычно иерархию наследования создают, когда несколько сущностей имеют общие по смыслу атрибуты, либо, когда сущности имеют общие по смыслу связи, либо, когда это диктуется бизнес-правилами.

Для каждой категории можно указать дискриминатор - атрибут родового предка, который показывает, как отличить одну категориальную сущность от другой (*атрибут Tip на рис. 2 35*).

Иерархии категорий делятся на два типа - полные и неполные. В полной категории одному экземпляру родового предка обязательно соответствует экземпляр в каком-либо потомке. *Т. е. в примере служащий обязательно является либо совместителем, либо консультантом, либо постоянным сотрудником.*

Если категория еще не выстроена полностью и в родовом предке могут существовать экземпляры, которые не имеют соответствующих экземпляров в потомках, то такая категория будет неполной. На рис. 2.35 показана **неполная категория - сотрудник может быть не только постоянным или совместителем, но и консультантом, однако сущность Консультант еще не внесена в иерархию наследования.**



Рис. 2.35. Неполная категория

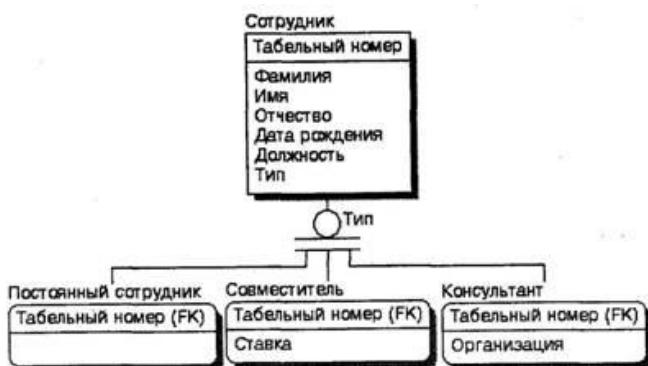


Рис. 2.36. Полная категория

Возможна комбинация полной и неполной категорий.



Рис. 2.37. Комбинация полной и неполной категорий

Нормализация данных.

Нормализация - процесс проверки и реорганизации сущностей и атрибутов с целью удовлетворения требований к реляционной модели данных. Нормализация позволяет быть уверенным, что каждый атрибут определен для своей сущности, значительно сократить объем памяти для хранения информации и устраниТЬ аномалии в организации хранения данных. В результате проведения нормализации должна быть создана структура данных, при которой информация о каждом факте хранится только в одном месте. Процесс нормализации сводится к последовательному приведению структуры данных к нормальным формам - формализованным требованиям к организации данных. Известны шесть нормальных форм:

- первая нормальная форма (1NF);
- вторая нормальная форма (2NF);
- третья нормальная форма (3NF);
- нормальная форма Бойса - Кодда (усиленная 3NF);
- четвертая нормальная форма (4NF);
- пятая нормальная форма (5NF).

На практике обычно ограничиваются приведением данных к третьей нормальной форме.

Нормальные формы основаны на понятии функциональной зависимости.

Функциональная зависимость (FD) – атрибут В сущности Е функционально зависит от атрибута А сущности Е тогда и только тогда, когда каждое значение А в Е связало с ним точно одно значение В в Е, т. е. А однозначно определяет В.

Полная функциональная зависимость – атрибут В сущности Е полностью функционально зависит от ряда атрибутов А сущности Е тогда и только тогда, когда В функционально зависит от А и не зависит ни от какого подряда А.

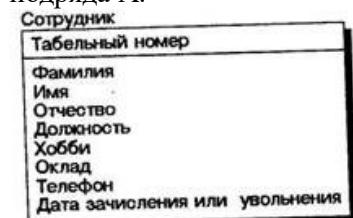


Рис. 2.47. Ненормализованная сущность "Сотрудник"

В сущности Сотрудник значение атрибутов Фамилия, Имя и Отчество однозначно определяются значением атрибута Табельный номер, т. е. атрибуты Фамилия, Имя и Отчество зависят от атрибута Табельный номер.

Функциональные зависимости определяются бизнес-правилами предметной области. Так, если оклад сотрудника определяется только должностью, то атрибут Оклад зависит от атрибута Должность; если оклад зависит еще, например, от стажа, то такой зависимости нет. В нижеследующих примерах будем считать для определенности, что такая зависимость есть.

Нормальные формы.

Первая нормальная форма (1NF). Сущность находится в первой нормальной форме тогда и только тогда, когда все атрибуты содержат атомарные значения. Среди атрибутов не должно встречаться повторяющихся групп, т. е. несколько значений для каждого экземпляра. На рис. 2.47 атрибуты Телефон и Хобби являются нарушением первой нормальной формы.

Другой ошибкой нормализации является хранение в одном атрибуте разных по смыслу значений. На рис. 2.47 атрибут Дата зачисления или увольнения хранит информацию как о зачислении, так и об увольнении сотрудника. Если хранится только одно значение, то невозможно понять, какая именно дата внесена.

Для приведения сущности к первой нормальной форме следует:

- разделить сложные атрибуты на атомарные,
- создать новую сущность,
- перенести в нее все "повторяющиеся" атрибуты,
- выбрать возможный ключ для нового РК (или создать новый РК).
- установить идентифицирующую связь от прежней сущности к новой, РК прежней сущности станет внешним ключом (FK) для новой сущности.



Рис. 2.49. Сущность "Сотрудник", приведенная к первой нормальной форме

Вторая нормальная форма (2NF). Сущность находится во второй нормальной форме, если она находится в первой нормальной форме и каждый неключевой атрибут полностью зависит от первичного ключа (не должно быть зависимости от части ключа). Вторая нормальная форма имеет смысл только для сущностей, имеющих сложный первичный ключ.

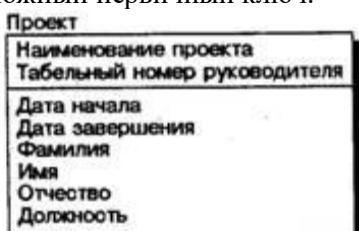


Рис. 2.50. Сущность "Проект"

Предположим, сущность Проект содержит информацию о проекте, которым руководит сотрудник, причем информация содержится как непосредственно о проекте, так и о руководителе проекта (рис. 2.50). Атрибуты Фамилия, Имя, Отчество и Должность зависят только от атрибута Табельный номер руководителя, но вовсе не от Наименования проекта. Другими словами, имеется зависимость только от части ключа.

Для приведения сущности ко второй нормальной форме следует:

- выделить атрибуты, которые зависят только от части первичного ключа, создать новую сущность;
- поместить атрибуты, зависящие от части ключа, в их собственную (новую) сущность;
- установить идентифицирующую связь от прежней сущности к новой (рис. 2.51).

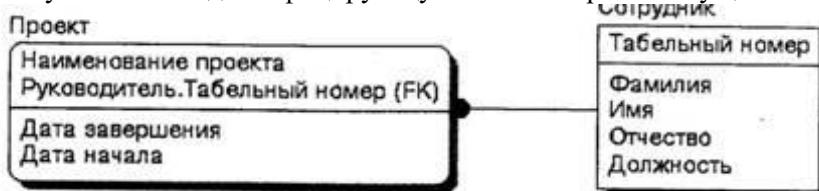


Рис. 2.51. Сущность "Проект", приведенная ко второй нормальной форме

Третья нормальная форма (3NF). Сущность находится в третьей нормальной форме, если она находится во второй нормальной форме и никакой неключевой атрибут не зависит от другого неключевого атрибута (не должно быть взаимозависимости между неключевыми атрибутами).

На рис. 2.49 сущность Сотрудник находится во второй нормальной форме (имеется только один атрибут первичного ключа, поэтому не может быть зависимости неключевых атрибутов от части ключа), но неключевой атрибут Оклад зависит от другого неключевого атрибута - Должности.

Для приведения сущности ко второй нормальной форме следует:

- создать новую сущность и перенести в нее атрибуты с одной и той же зависимостью от неключевого атрибута;
- использовать атрибут(ы), определяющий эту зависимость, в качестве первичного ключа новой сущности;
- установить неидентифицирующую связь от новой сущности к старой (рис. 2.52).



Рис. 2.52. Сущность "Сотрудник", приведенная к третьей нормальной форме

В третьей нормальной форме каждый атрибут сущности зависит от ключа, от всего ключа целиком и ни от чего другого, кроме как от ключа.

Поддержка нормализации в ERwin. ERwin не содержит полного алгоритма нормализации и не может проводить нормализацию автоматически, однако его возможности облегчают создание нормализованной модели данных. Запрет на присвоение неуникальных имен атрибутов в рамках модели облегчает соблюдение правила "один факт - в одном месте". Имена ролей атрибутов внешних ключей и унификация атрибутов также облегчают построение нормализованной модели.

Денормализация. В результате нормализации все взаимосвязи данных становятся правильно определены, исключаются аномалии при оперировании с данными, модель данных становится легче поддерживать. Однако часто нормализация данных не ведет к повышению производительности ИС в целом.

В целях повышения производительности при переходе на физический уровень приходится сознательно отходить от нормальных форм для того, чтобы использовать возможности конкретного сервера или ИС в целом. В отличие от процесса нормализации денормализация не может быть представлена в виде четко сформулированных правил. В каждом конкретном случае приходится искать конкретные решения, которые используют специфику ИС и предметной области и не могут быть тиражированы.

Денормализация, как правило, проводится на уровне физической модели. ERwin позволяет сохранить на уровне логической модели нормализованную структуру, при этом построить на уровне, физической модели структуру (возможно, денормализованную), которая обеспечивает лучшую производительность, используя особенности конкретной СУБД и бизнес-правил предметной области.

Прямое и обратное проектирование.

Прямое проектирование (forward engineering) – процесс трансформации модели в код путем ее отображения на язык реализации.

Обратное проектирование (reverse engineering) – процесс трансформации кода в модель посредством его отображения из определенного языка реализации.

Прямое проектирование предполагает перевод с UML на некий язык программирования, обратное – с языка программирования на язык UML-диаграмм.

UDP (User Defined Properties). Пользовательские свойства, которые могут быть присвоены объектам в модели данных. Позволяют расширять стандартные метаданные, добавляя пользовательские атрибуты к таблицам, столбцам, отношениям и другим элементам модели.

Триггеры. Блоки кода, которые выполняются автоматически при наступлении определенного события в базе данных. Могут включать вставку, обновление или удаление данных. Используются для автоматизации определенных действий в базе данных.

Хранимые процедуры. Предварительно скомпилированные наборы инструкций SQL, которые могут быть вызваны из клиентского приложения или других хранимых процедур. Облегчают выполнение сложных операций и повышают эффективность базы данных.

ERWin реализует собственный макроязык для подготовки прототипов триггеров и процедур. Базовые шаблоны встроены в ERWin, но пользователь может определить свои собственные и использовать их вместо стандартных.

Скрипты. Программные инструкции, которые могут быть выполнены для выполнения определенных задач в модели данных. Используются для автоматизации процессов, не поддерживаемых стандартным пользовательским интерфейсом. Могут включать создание объектов, применение изменений, генерацию SQL-скриптов и т.д.

2.3 Модели управления. Взаимодействие моделей обработки данных и управления. Средства описания управляющих воздействий. Диаграммы переходов состояний (STD).

Модели управления.

Диаграммы потоков управления отражают распределение и трансформацию потоков управления.

Модель управления содержит информацию о том, какие решения принимает система в ответ на изменение внешних или внутренних воздействий или режимов функционирования. При принятии решения – ответной реакции – происходит изменение состояния системы, влекущее включение/выключение процессов диаграммы. Помимо воздействий модель описывает, какую информацию о состоянии внешних сущностей необходимо получить системе и какие сигналы о собственном состоянии передать окружению.

Средства описания:

- диаграмма потоков управления (Control Flow Diagram - CFD);
- спецификация управляющего процесса, определяющая управление в системе.

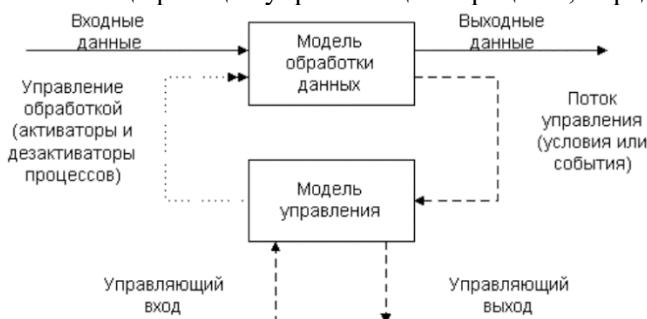


Рис. 2.8 Взаимодействие моделей обработки данных и управления

Модель обработки включает диаграммы потоков данных (DFD) и набор спецификаций процессов.

Модель управления представляет описание функционирования конечного автомата/описание модели состояний. Образует диаграммы управляющих потоков (CFD) и набор управляющих спецификаций.

Описание обрабатывающего процесса – PSpec.

Описание управляющего процесса – CSpec.

Состав описания:

- таблицы событий;
- таблицы решений;
- таблицы переходов в состояния;
- диаграммы переходов в состояния (STD – State Transition Diagrams).

Диаграммы переходов состояний, словарь данных (требований) и таблицы активизации процессов образуют модель управления системы.

Общая модель требований описывает следующие аспекты системы:

- потоки данных в системе (DFD);
- спецификации процессов обработки данных (PSPEC);
- потоки управления в системе (CFD);
- спецификации управляющих процессов в системе (STD и таблицы активизации процессов).

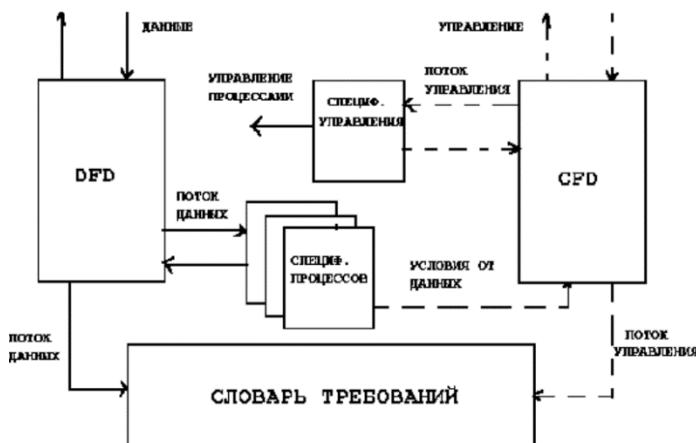


Рис.2.12 Структура модели требований
Диаграмма потоков управления (CFD)

Процессы CFD повторяют DFD и имеют такие же номера и имена. Отличие в том, что в DFD процессы взаимодействуют посредством передачи данных (сплошные стрелки), а в CFD – посредством передачи управления (пунктирные стрелки). Стрелки показывают направление и способ передачи данных или управления (синхронный/асинхронный).

Спецификации управления

Спецификации управления предназначены для моделирования и документирования аспектов систем, зависящих от времени или реакции на событие. Они позволяют осуществлять декомпозицию управляющих процессов и описывают отношения между входными и выходными управляющими потоками на управляющем процессе-предке.

Для этого используются **диаграммы переходов состояний (STD)** – графическая форма представления конечного автомата – математической абстракции, используемой для моделирования детерминированного поведения технических объектов или объектов реального мира.

На этапе анализа требований и определения спецификации диаграмма демонстрирует поведение системы при получении управляющих воздействий (сигналов) – управляющей информации извне. Получив такое воздействие, система должна выполнить определенные действия, а затем или остаться в том же состоянии, или перейти в другое, зафиксировав некоторые изменения.

Для построения диаграммы переходов состояний необходимо в соответствие с теорией конечных автоматов определить основные состояния, управляющие воздействия (условия перехода), выполняемые действия и возможные переходы разрабатываемого ПО.

С помощью STD можно моделировать последующее функционирование системы на основе предыдущего и текущего функционирования. Система находится точно в одном из конечного множества состояний. С течением времени она может изменить его, при этом переходы между состояниями должны быть точно определены.

Состав STD:

- состояние – условие устойчивости для системы;
- начальное состояние – узел STD, являющийся стартовой точкой начального перехода;
- переход – перемещение системы из одного состояния в другое;
- условие – событие (события), вызывающее переход и идентифицируемое именем перехода. Если в

Состояния представляются узлами, переходы – дугами. Условия идентифицируются именем перехода. Действия или отклики на события привязываются к переходам и записываются под соответствующим условием. Начальное состояние должно иметь входной переход, изображаемый потоком из подразумеваемого стартового узла (иногда стартовый узел изображается небольшим квадратом и привязывается к входному состоянию).



Способы построения:

- идентификация возможных состояний и исследование небессмысленных связей (переходов) между ними;
- построение начального состояния, затем следующего за ним и т.д.

Результат – предварительная STD, для которой затем осуществляется контроль состоятельности, заключающийся в ответе на вопросы:

- все ли состояния определены и имеют уникальное имя?
- все ли состояния достижимы?
- все ли состояния имеют выход?

- (для каждого состояния) реагирует ли система соответствующим образом на все возможные условия (особенно на ненормальные)?
 - все ли входные (выходные) потоки управляющего процесса отражены в условиях (действиях) на STD?
- В ситуации, когда число состояний и/или переходов велико, для проектирования спецификаций управления могут использоваться таблицы и матрицы переходов состояний.

Текущее состояние	Условие	Действие	Следующее состояние
начальное состояние	активируется каждый раз		ОЖИДАНИЕ
ОЖИДАНИЕ	введенная кред. карта	получить пароль	ОБРАБОТКА
ОБРАБОТКА	некорректный пароль	удалить кред. карту	ОЖИДАНИЕ
ОБРАБОТКА	корректный пароль	обеспечить треб. сервис удалить кред. карту	ОЖИДАНИЕ

5.3 Проектирование

- Задачи этапа проектирования. Модель реализации
- Функциональная модель проектирования
 - Модель системных процессов
 - Иерархия экранных форм и меню
 - Структурные схемы программ

Проектирование ПО — этап ЖЦ ПО, на котором устанавливается, как будут реализованы требования пользователя, выявленные на этапе анализа. Осуществляется построение модели реализации проектируемой прикладной системы.

Задачи этапа проектирования. Модель реализации

Модели реализации (физическая модель) является расширением модели требований и демонстрирует, как система будет удовлетворять требованиям без технических подробностей.

Компоненты модели реализации:

- диаграммы потоков данных (модели системных процессов);
- модель данных (ER-диаграммы необходимых параметров приложений);
- модель пользовательского интерфейса (иерархия диаграмм последовательности экранных форм);
- структурные схемы программ (модели межмодульных и внутримодульных взаимодействий с использованием техники структурных карт).

Функциональная модель проектирования

Функциональные модели, используемые на стадии проектирования ПО, предназначены для описания функциональной структуры проектируемой системы.

• Модель системных процессов

Для DFD переход от модели бизнес-процессов организации к модели системных процессов может происходить следующим образом:

- внешние сущности на контекстной диаграмме заменяются или дополняются техническими устройствами (рабочими станциями, принтерами и т.п.)
- для каждого потока данных определяется, посредством каких технических устройств информация передается или производится;
- процессы на диаграмме нулевого уровня заменяются соответствующими процессорами – обрабатывающими устройствами. Процессорами могут быть как технические устройства (ПК, рабочие станции, серверы БД), так и служащие-исполнители;
- определяется и изображается на диаграмме тип связи между процессорами (например, локальная сеть – LAN)
- определяются задачи для каждого процессора (приложения, необходимые для работы системы), для чего строятся соответствующие диаграммы. Определяется тип связи между задачами.
- устанавливаются ссылки между задачами и процессами диаграмм потоков данных следующих уровней.

• Иерархия экранных форм и меню (модель пользовательского интерфейса)

На диаграммах потоков данных среди процессов нижнего уровня выделяются интерактивные (диалоговые) и не интерактивные процессы. После этого строят диаграммы последовательности форм (FSD - Form Sequence Diagrams).

FSD показывает, какие формы в каком порядке появляются в приложении, фиксируется набор и структура вызовов экранных форм.

Диаграммы последовательности форм образуют иерархию, на вершине которой находится главная форма приложения, реализующего систему/подсистему. На втором уровне находятся формы, реализующие процессы

нижнего уровня на диаграммах потоков данных. Показывается взаимосвязь между каждой формой и определенным процессом, взаимосвязь между каждой формой и одной или более сущностями ER-диаграмм.

Описание экранных форм и отчетов должно содержать:

- назначения формы;
- данные навигации;
- список ошибок и реакция на них;
- ограничения доступа.

Построение диаграмм последовательностей экранных форм:

- на DFD выбираются интерактивные процессы нижнего уровня;
- определяется структура меню: интерактивные процессы группируются в меню;
- меню изображаются над формами, соответствующими интерактивным процессам, и соединяются стрелками, направленными от меню к формам;
- определяется верхняя (главная) форма приложения, связывающая все формы с меню.

• Структурные схемы программ

В структурных схемах программ определяются главные модули, маршруты связи по данным и по управлению между модулями, основные подпрограммы внутри каждого модуля, состав и взаимосвязь элементов (структуры) данных, спецификации форматов входных и выходных файлов.

Техника структурных карт используется, чтобы продемонстрировать, каким образом системные требования будут отражаться комбинацией программных структур:

- структурные карты Константайна (Constantine) описывают отношения между модулями;
- структурные карты Джексона (Jackson) для описывают внутреннюю структуру модулей.

Проектирование спецификаций:

- разделение проекта на 10-15 модулей;
- организация иерархии модулей;
- определение маршрутов данных между модулями;
- определение форматов внешних файлов;
- определение способов доступа к внешним файлам;
- определение структур данных;
- проектирование ключевых алгоритмов;
- определение подпрограмм каждого модуля.

ТЕМА 6. ОСНОВЫ ОБЪЕКТНОЙ ТЕХНОЛОГИИ

6.1 Принципы объектно-ориентированного представления программных систем

Рассмотрение сложной системы подразумевает декомпозицию — разбиение на составляющие элементы — алгоритмическую или объектно-ориентированную.

В основе алгоритмической — разбиение по действиям (алгоритмам).

В объектно-ориентированной — по автономным лицам — объектам реального/виртуального мира, которые имеют описания действий и данных.

Принципы объектно-ориентированного представления:

• **Абстрагирование.** Сводится к формированию абстракций. Каждая фиксирует основные характеристики объекта, которые отличают его от других, отвлекая (абстрагируя) от несущественных характеристик, и обеспечивают понятийные границы.

• **Инкапсуляция.** Служит для отделения интерфейса абстракции от ее реализации. Достигается с помощью информационной закрытости: скрываются структура объектов и реализация их методов.

• **Модульность.** Определяет способность системы подвергаться декомпозиции на модули — физические контейнеры, в которых объявляются классы и объекты логической разработки — с целью уменьшения сроков и стоимости разработки.

• **Иерархическая организация.** Задает размещение абстракций на различных уровнях описания системы. Упрощается понимание и решение проблем заказчика, система становится более обозримой.

Инструменты иерархической организации:

◦ структура из классов («is a»-иерархия) строится с помощью наследования, определяющего отношение между классами, где класс разделяет структуру/поведение, определенные в одном (единичное наследование) или нескольких (множественное наследование) других классах;

◦ структура из объектов («part of»-иерархия) базируется на отношении агрегации, которая полезна в сочетании с наследованием, поскольку:

- агрегация обеспечивает физическую группировку логически связанных структур;
- наследование позволяет использовать эти группы в других абстракциях.

6.2 Объекты

- Общая характеристика объектов

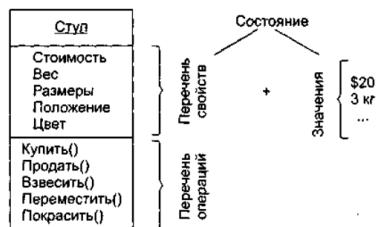
- Виды отношений между объектами

Объект – конкретные сущности, которые существуют во времени и пространстве/конкретное представление абстракции.

Общая характеристика объектов:

Индивидуальность – характеристика, которая отличает объект от других.

Состояние – перечень свойств объекта с текущими значениями каждого.



Поведение – как объект воздействует на другие объекты (или подвергается воздействию). Является функцией состояния объекта и выполняемых им операций.

Состояние объекта – суммарный результат его поведения.

Операция – обслуживание, которое объект предлагает своим клиентам:

- модификатор (изменяет состояние объекта);
- селектор (дает доступ к состоянию, не изменяя его);
- итератор (дает доступ к содержанию объекта по частям, в строго определенном порядке);
- конструктор (создает объект и инициализирует его состояние);
- деструктор (разрушает объект и освобождает занимаемую им память).



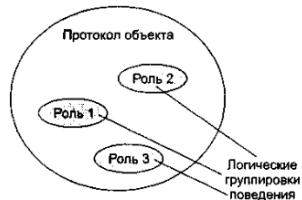
Наличие состояния подразумевает определенный порядок выполнения операций.

Объекты – независимые автоматы – могут быть активные и пассивные. Активный объект автономен и может проявлять свое поведение без воздействия других объектов. Пассивный объект, наоборот, может изменять свое состояние только под воздействием других объектов



Протокол – образование методов, ассоциированных с конкретным объектом. Определяет допустимое поведение объекта и заключает его представление.

Роли, которые может играть объект, – разделенные на логические группы поведения протокол.



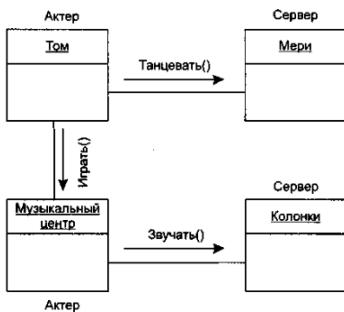
Обязанности – обязательства объекта обеспечить определенное поведение.

Виды отношений между объектами:

Связи. Физическое или понятийное соединение между объектами. Обеспечивает:

- вызов объекта-клиента операции объекта-поставщика;
- перемещение данных от одного объекта к другому.

Сообщения изображаются стрелками, отмечающими их направления, и помечаются именами вызываемых операций.



Роли участника связи:

- актер может воздействовать на другие объекты, но сам воздействию не подвержен;
- сервер используется другими объектами, но не воздействует на них;
- агент может как воздействовать на другие объекты, так и использоваться ими. Создается для выполнения работы от имени актера или другого агента.

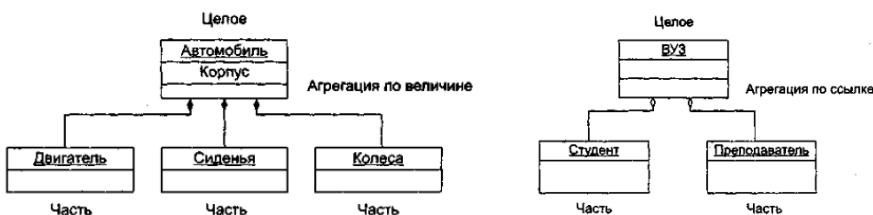
Чтобы два объекта могли взаимодействовать, они должны быть видимыми друг для друга.

Формы видимости:

- объект-поставщик (сервер) глобален для клиента;
- объект-поставщик (сервер) является параметром операции клиента;
- объект-поставщик (сервер) является частью объекта-клиента;
- объект-поставщик (сервер) является локально объявленным объектом в операции клиента.

Агрегация. Обозначает отношения объектов в иерархии «целое/часть». Обеспечивает возможность перемещения от целого (агрегата) к его частям (свойствам).

Агрегация может обозначать физическое (по величине) или нефизическое (по ссылке) включение части в целое.



6.3 Классы

- Общая характеристика классов
- Виды отношений между классами
- Виды классов

Класс – описание множества объектов, которые разделяют одинаковые свойства, операции, отношения и семантику (смысл)/абстракция существенных характеристик объекта.

Объект – экземпляр класса.

Общая характеристика классов:

Интерфейс – внешнее представление класса. Объявляет возможности (услуги) класса, но скрывает его структуру и поведение. Состоит из объявлений операций, применимых к экземплярам, может включать объявления типов, переменных, констант и исключений, необходимых для полноты абстракции.

КЛАСС	
Части	интерфейсные
	Публичная
	Защищенная
	Приватная
Реализация	

Части интерфейса:

- публичная (public), объявления которой доступны всем клиентам;
- защищенная (protected), объявления которой доступны только самому классу, его подклассам и дружественным классам;
- приватная (private), объявления которой доступны только самому классу и дружественным классам.

Реализация – внутреннее представление класса. Описывает секреты поведения класса и включает реализации всех операций, определенных в интерфейсе.

Виды отношений между классами:

Основные:

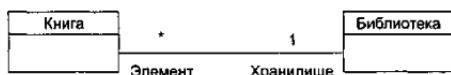
- ассоциация (фиксирует связи между экземплярами классов);

- зависимость (отображает влияние одного класса на другой);
- обобщение-специализация («is a»-отношение);
- целое-часть («part of»-отношение).

Для покрытия основных:

- ассоциация (взаимодействие объектов, принадлежащих разным классам);
- наследование (разновидность отношения обобщение-специализация);
- агрегация (отношения целое-часть для экземпляров классов);
- зависимость (отношение между клиентом и сервером);
- конкретизация (другая разновидность отношения обобщение-специализация);
- метакласс (класс классов, позволяющий обращаться с классами, как с объектами);
- реализация (отношение, при котором класс-приемник обеспечивает собственную реализацию интерфейса другого класса-источника).

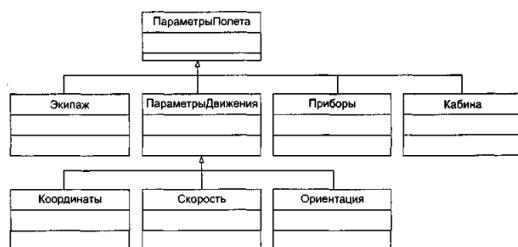
Ассоциации классов. Обозначают семантическое соединение классов, при этом не указывают направление и точную реализацию отношения. С помощью ассоциаций достигается понимание участников связей, их ролей и мощности (количества элементов).



Мощности ассоциации:

- один-к-одному;
- один-ко-многим;
- многие-ко-многим.

Наследование. Отношение, при котором класс разделяет структуру и поведение, определенные в одном (простое наследование) или многих (множественное наследование) других классах. Подкласс является специализацией его суперкласса за счет дополнения/переопределения существующей структуры или поведения.



Полиморфизм. Возможность с помощью одного имени обозначать операции из различных классов, относящихся к общему суперклассу. Вызов обслуживания по полиморфному имени приводит к исполнению одной из некоторого набора операций.

Агрегация. Аналогична отношениям агрегации между объектами.



Рис. 9.14. Агрегация классов

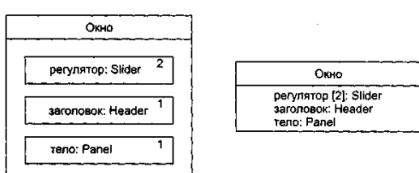


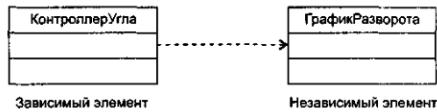
Рис. 9.15. Формы представления композиции

Обозначает отношения объектов в иерархии «целое/часть». Обеспечивает возможность перемещения от целого (агрегата) к его частям (свойствам).

Агрегация может обозначать физическое (по величине) или нефизическое (по ссылке) включение части в целое.



Зависимость. Показывает, что изменение в одном классе (независимом) может влиять на другой класс (зависимый), который использует его. С помощью зависимости уточняют, какая абстракция является клиентом, а какая — поставщиком определенной услуги.



Конкретизация. Процесс наполнения шаблона (родового или параметризованного класса) с целью получение класса, от которого возможно создание экземпляров. Родовой класс служит заготовкой, параметры которого могут настраиваться другими классами, типами, объектами, операциями.



Метакласс. Класс классов, позволяющий обращаться с классами, как с объектами.

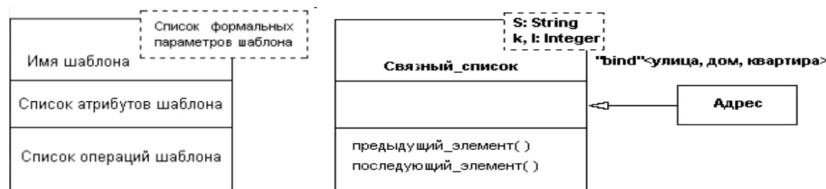
Реализация. Отношение между классификаторами, при котором один определяет «контракт», а другой гарантирует его выполнение:

- между интерфейсами и реализующими их классами или компонентами;
- между прецедентами и реализующими их кооперациями.

Графически представляется, как наследование, но с пунктирной линией.

Виды классов:

- **Конкретный класс.** Класс, на основе которого могут быть созданы экземпляры или объекты;
- **Абстрактный класс.** Класс, который не имеет экземпляров или объектов. Позволяют определить поведение по умолчанию и заставить подклассы обеспечить конкретное поведение;
- **Параметризованный класс (шаблон).** Предназначен для обозначения класса, который имеет нефиксированные формальные параметры. Он определяет множество классов, каждый из которых может быть получен связыванием этих параметров с действительными значениями;

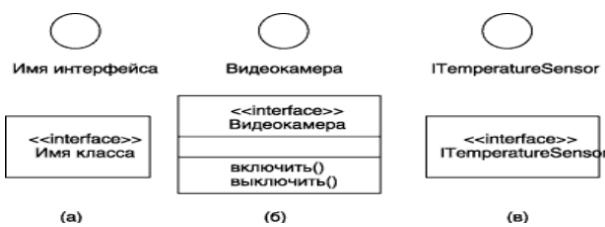


• Интерфейс. Именованное множество операций, которые характеризуют поведение отдельного элемента модели. В контексте UML является специальным случаем класса, у которого имеются операции, но отсутствуют атрибуты.

В концепции интерфейса лежит абстрактный класс, но интерфейс предотвращает реализации методов или функций, оставляя эту обязанность классу, который реализует данный интерфейс.

Преимущества интерфейса:

- множественное наследование;
- слабая связанность;
- программа-интерфейс не реализуется;
- полиморфизм с динамическим связыванием;
- абстрактные уровни, разделение функциональности.



6.4 Базис языка визуального моделирования

- Унифицированный язык моделирования (UML). Диаграммы UML.
- Механизмы расширения в UML.

Для создания моделей анализа и проектирования ООП-систем используют языки визуального моделирования.

UML – стандартный язык для написания моделей анализа, проектирования и реализации объектно-ориентированных программных систем. Может использоваться для визуализации, спецификации, конструирования и документирования результатов программных проектов.

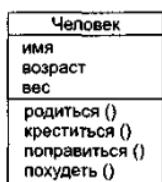
Словарь UML образуют предметы, отношения и диаграммы. Предметы – абстракции, которые являются элементами модели, отношения связывают эти предметы, диаграммы группируют коллекции предметов.

Виды предметов:

- Структурные предметы – существительные в UML-моделях. Представляют статические части модели – понятийные или физические элементы.

Виды структурных предметов:

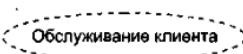
- Класс – описание множества объектов, которые разделяют одинаковые свойства, операции, отношения и семантику (смысл);



- Интерфейс – набор операций, которые определяют услуги класса или компонента. Описывает поведение элемента, видимое извне;



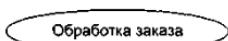
- Кооперация (сотрудничество) определяет взаимодействие и является совокупностью ролей и других элементов, которые работают вместе для обеспечения коллективного поведения более сложного, чем простая сумма всех элементов.



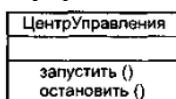
- Актер – набор согласованных ролей, которые могут играть пользователи при взаимодействии с системой (ее элементами Use Case)



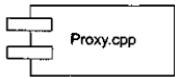
- Элемент Use Case (Прецедент) – описание последовательности действий, выполняемых системой в интересах отдельного актера и производящих видимый результат.



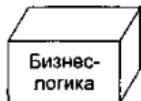
- Активный класс – класс, чьи объекты имеют один или несколько процессов (или потоков) и могут инициировать управляющую деятельность. Его объекты действуют одновременно с объектами других классов.



- Компонент – физическая заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает его реализацию.



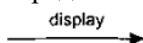
- Узел – физический элемент, который существует в период работы системы и представляет ресурс, имеющий память и возможности обработки.



- **Предметы поведения** – динамические части UML-моделей. Являются глаголами, представлением поведения во времени и пространстве.

Виды предметов поведения:

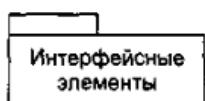
- Взаимодействие – сообщения, которыми обменивается набор объектов в конкретном контексте для достижения определенной цели.



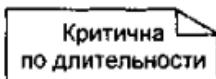
- Конечный автомат – последовательность состояний объекта или взаимодействия, выполняемые в ходе его существования в ответ на события с учетом обязанностей по этим событиям.



- **Группирующие предметы** – организационные части UML-моделей – пакеты. В пакет могут помещаться структурные предметы, предметы поведения и другие группировки. В отличие от компонента, который существует в период выполнения, пакет – концептуальное понятие – существует только в период разработки.



- **Поясняющие предметы** – разъясняющие части UML-моделей, которые можно применить для описания, объяснения и комментирования элемента модели – примечания. Примечание – символ для отображения ограничений и замечаний, присоединяемых к элементу или совокупности элементов.

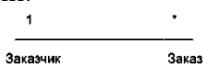


Виды отношений:

- Зависимость – семантическое отношение между предметами, где изменение в одном (независимом) может влиять на семантику другого (зависимого).



- Ассоциация – структурное отношение, которое описывает набор связей, являющихся соединением между объектами. Агрегация – разновидность ассоциации, представляющая структурное отношение между целым и его частями.



- Обобщение – отношение специализации/обобщения, в котором объекты специализированного элемента (потомка, ребенка) могут заменять объекты обобщенного элемента (предка, родителя).



- Реализация – семантическое отношение между классификаторами, где один классификатор определяет контракт, который другой классификатор обязуется выполнять.



Диаграмма – графическое представление множества элементов – связный граф из вершин (предметов) и дуг (отношений). Рисуются для визуализации системы с разных точек зрения.

Виды диаграмм:

- Диаграмма классов показывает набор классов, интерфейсов, сотрудничеств и их отношений. Обеспечивают статическое проектное представление системы или статическое представление процессов системы, если включают активные классы.

- Диаграмма объектов показывает набор объектов и их отношения. Представляет статический «моментальный снимок» экземпляров предметов, которые находятся в диаграммах классов. Так же обеспечивают статическое проектное представление или статическое представление процессов системы, но с точки зрения реальных или фототипичных случаев.

- Диаграмма Use Case (диаграмма прецедентов) показывает набор элементов Use Case, актеров и их отношений. Создается статическое представление Use Case. Используются для организации и моделирования поведения системы, задания требований заказчика.

- Диаграмма взаимодействия показывает взаимодействие, включающее набор объектов и их отношений, а также пересылаемые между ними сообщения. Обеспечивают динамическое представление системы. Разновидности диаграмм взаимодействия:

- Диаграмма последовательности выделяет упорядочение сообщений по времени.

- Диаграмма сотрудничества (диаграмма кооперации) выделяет структурную организацию объектов, посылающих и принимающих сообщения.

- Диаграмма схем состояний показывает конечный автомат, представляет состояния, переходы, события и действия. Обеспечивают динамическое представление системы. Используются для моделирования поведения интерфейса, класса или сотрудничества.

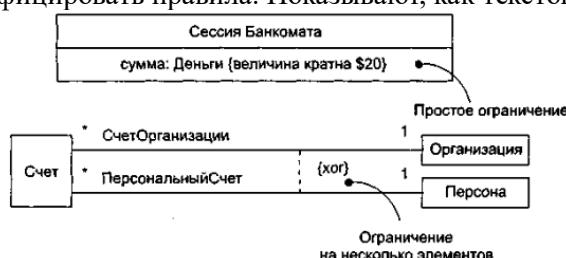
- Диаграмма деятельности показывает поток от действия к действию внутри системы. Обеспечивают динамическое представление системы. Используются для моделирования функциональности и выделяют поток управления между объектами.

- Компонентная диаграмма показывает организацию набора компонентов и зависимости между ними. В компонент отображается один или несколько классов, интерфейсов или коопераций. Обеспечивают статическое представление реализации системы.

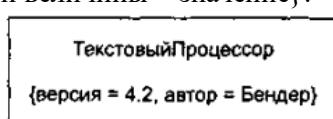
- Диаграмма размещения (диаграмма развертывания) показывает конфигурацию обрабатывающих узлов периода выполнения, а также компоненты, живущие в них. Узел включает один или несколько компонентов. Обеспечивают статическое представление размещения системы.

Расширения UML:

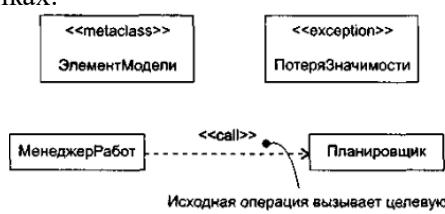
- Ограничение (constraint) расширяет семантику строительного UML-блока, позволяя добавить или модифицировать правила. Показывают, как текстовую строку, заключенную в фигурные скобки {}.



- Теговая величина (tagged value) расширяет характеристики строительного UML-блока, позволяя создать новую информацию в спецификации конкретного элемента. Показывают, как строку в фигурных скобках {имя теговой величины = значение}.



- Стереотип (stereotype) расширяет словарь языка, позволяя создавать новые виды строительных блоков, производные от существующих. У них могут быть дополнительные ограничения и теговые величины, а также другое визуальное представление. Отображают как имя, указанное в двойных угловых скобках/угловых кавычках.



ТЕМА 7. ДИАГРАММЫ UML

7.1 Унифицированный язык моделирования (UML). Диаграммы UML.

UML – стандартный язык для написания моделей анализа, проектирования и реализации объектно-ориентированных программных систем. Может использоваться для визуализации, спецификации, конструирования и документирования результатов программных проектов.

Словарь UML образуют предметы, отношения и диаграммы. Предметы – абстракции, которые являются элементами модели, отношения связывают эти предметы, диаграммы группируют коллекции предметов.

Унифицированный язык моделирования (UML) представляет собой стандартный язык для визуального представления и проектирования программных систем. Он разработан с целью обеспечения единого подхода к проектированию сложных систем, а также для обеспечения стандартизации и понимания между разработчиками.

Виды диаграмм UML:

- 1) Диаграмма классов (Class Diagram) - предназначена для описания структуры системы с точки зрения классов, их атрибутов и методов
- 2) Диаграмма вариантов использования (Use Case Diagram) - используется для описания функциональности системы с точки зрения взаимодействия с внешними акторами.
- 3) Диаграмма последовательности (Sequence Diagram) - позволяет визуализировать взаимодействие объектов в определенном порядке времени.
- 4) Диаграмма состояний (State Diagram) - используется для моделирования поведения объекта в различных состояниях.
- 5) Диаграмма активностей (Activity Diagram) - предназначена для моделирования бизнес-процессов, алгоритмов или действий в системе. Визуализирует последовательность действий, включая разветвления и объединения.
- 6) Диаграмма компонентов (Component Diagram) - позволяет описать структуру и взаимосвязи компонентов системы.
- 7) Диаграмма развертывания (Deployment Diagram) - используется для описания размещения компонентов и узлов вычислительной инфраструктуры системы.

7.2 Диаграммы прецедентов (сценариев) Use Cases. Определение требований к системе.

- Способы взаимосвязи прецедентов.
- Реализация вариантов использования

Диаграмма прецедентов (Использования) – наиболее общее представление функционального назначения системы. Рассматривается, как главное средство для первичного моделирования динамики системы, используется для выяснения требований к разрабатываемой системе и фиксации этих требований в форме, которая позволит проводить дальнейшую разработку

Цели разработки диаграммы:

- определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы;
- сформулировать общие требования к функциональному поведению проектируемой системы;
- разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей;
- подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями.

Основные элементы диаграммы:

- 1) Use Case (Вариант использования (ВИ), сценарий использования, прецедент) - представляет собой последовательность шагов, описывающих взаимодействие между пользователем и системой.
- 2) Actor (Пользователь, субъект, действующее лицо, исполнитель, актер, актант) - представляет пользователя, организацию или внешнюю систему, взаимодействующую с используемым приложением или системой.
- 3) Отношения (Зависимости, ассоциации, обобщения) - указывает, что субъект принимает участие в варианте использования.

Определение требований к системе.

Типы связей между элементами диаграммы

- 1) **Ассоциация (Communication)** - между действующим лицом и вариантом использования показывает, что действующее лицо тем или иным способом взаимодействует (предоставляет исходные данные, получает результат) с вариантом использования.
- 2) **Зависимости:**
 - a. **Включения (Include)** - специфицирует тот факт, что некоторый вариант использования содержит поведение, определенное в другом варианте использования.
 - b. **Расширения (Extend)** - определяет взаимосвязь одного варианта использования с некоторым другим вариантом использования, функциональность или поведение которого задействуется первым не всегда, а только при выполнении некоторых дополнительных условий.
- 3) **Обобщения (Generalization)** - предназначено для спецификации того факта, что один элемент модели является специальным или частным случаем другого элемента модели.

Реализация вариантов использования(ВИ):

Вариант использования (ВИ) – последовательность действий (транзакций), выполняемых системой в ответ на событие, инициируемое некоторым внешним объектом (действующим лицом). Описывает типичное взаимодействие между пользователем и системой: определяет поведение классификатора (подсистемы/системы) и описывает услугу, которую предоставляет своим пользователям классификатор. К взаимодействию относятся только коммуникации между системой и актантами, внутреннее поведение и реализация скрыты.

ВИ включает:

- описание основного поведения, осуществляемого в ответ на запрос пользователя;
- варианты этого поведения: альтернативные последовательности, исключительное поведение, обработка ошибок.

Цель создания ВИ – описать часть функциональности во всех ее вариантах, включая ошибочные. Все варианты использования классификатора специфицируют способы его использования.

7.3 Диаграммы классов (ДК). Общие сведения. Уровни ДК. Работа с атрибутами: стереотипы, видимость, метод локализации. Операции

Диаграмма классов (ДК) – структурная диаграмма UML, демонстрирующая общую структуру иерархии классов системы, их коопераций, атрибутов (полей), методов, интерфейсов и взаимосвязей (отношений) между ними.

Классом называется именованное описание совокупности объектов с общими атрибутами, операциями, связями и семантикой. У каждого класса должно быть имя (текстовая строка), уникально отличающее его от всех других классов.

Цель – графическое представление статической структуры декларативных элементов системы (классов, типов и т. п.). ДК также содержит элементы поведения (например, операции) и представления пакетов.

Уровни классов:

- **аналитический.** Класс содержит только набросок системы и работает, как логическая концепция предметной области или программного продукта;
- **проектирования.** Класс отражает основные проектные решения касательно распределения информации и функциональности, объединяя сведения о состоянии и операциях;
- **реализации.** Класс дорабатывается для воплощения в среде разработки, при этом не воспрещается опустить общие свойства, которые не применяются на выбранном языке программирования.

Элементы ДК:

- класс – определение объектов, совместно обладающих данными характеристиками структуры и поведения;
- классификатор – общее имя класса, интерфейса или перечисления;
- элемент управления "развернуть/свернуть" – расширитель в верхней левой части классификатора или сегмента;
- атрибут – типизированное значение, прикрепленное к экземпляру классификатора;
- операция – метод или функция, которую можно выполнить с помощью экземпляров классификатора;
- ассоциация – отношение между членами двух классификаторов:
 - агрегат – ассоциация, представляющая отношение совместного владения;
 - композиция – ассоциация, представляющая отношение целого и части;
- имя ассоциации – имя ассоциации;
- имя роли – имя роли, одного из окончаний ассоциации. Может использоваться для ссылки на связанный объект;
- кратность – сколько объектов на этом окончании можно связать с объектами на другом;
- обобщение – конкретный классификатор наследует часть своего определения от общего классификатора;
- интерфейс – определение части внешне видимого поведения объекта;
- перечисление – классификатор, состоящий из набора строковых литералов;
- пакет – группа классификаторов, ассоциаций, действий, линий жизни, компонентов и пакетов;
- импорт – отношение между пакетами, указывающее, что один включает определения другого;
- зависимость – определение или реализация зависимого классификатора может измениться, если изменяется классификатор на окончании с наконечником стрелки;
- реализация – класс реализует операции и атрибуты, определенные интерфейсом.

Стандартные стереотипы:

- «actor» – действующее лицо;
- «auxiliary» – вспомогательный класс;
- «enumeration» – перечислимый тип данных;
- «exception» – исключение (UML1);
- «focus» – основной класс;
- «implementationClass» – реализация класса;
- «interface» – составляющие абстрактные;
- «metaclass» – экземпляры-классы;

- «powertype» – метакласс, экземпляры которого – наследники данного класса (UML1);
- «process» – активный класс;
- «thread» – активный класс (UML1);
- «signal» – класс, экземпляры которого – сигналы;
- «stereotype» – новый элемент на основе существующего;
- «type» – тип данных;
- «dataType» – тип данных;
- «utility» – нет экземпляров, служба.

Работа с атрибутами.

Формат записи атрибута: <квантор видимости> <имя атрибута> [кратность] :<тип атрибута> = <исходное значение> {строка-свойство}.

Видимость (visibility) – качественная характеристика элементов класса, характеризующая возможность других объектов модели оказывать влияние на данный класс. Видимость определяют, чтобы скрыть детали реализации класса и показать только те особенности, которые необходимы для осуществления обязанностей, продекларированных абстракцией.

Видимость UML специфицируется с помощью квантора видимости (visibility):

- "+" – общедоступный (public). Атрибут доступен из любого другого класса пакета, в котором определена диаграмма;
- "#" – защищенный (protected). Атрибут недоступен для всех классов, за исключением подклассов данного класса;
- "-" – закрытый (private). Атрибут недоступен для всех классов без исключения;
- "~" – пакетный (package). Атрибут недоступен для всех классов за пределами пакета, в котором определен класс-владелец данного атрибута.

Имя атрибута – единственный обязательный элемент синтаксического обозначения атрибута, должно начинаться со строчной буквы и не должно содержать пробелов.

Кратность атрибута характеризует количество конкретных атрибутов данного типа, входящих в состав отдельного класса.

Тип атрибута может быть встроенным элементарным (primitive type) или другим классом.

Элементарный тип – распознаваемый и поддерживаемый базовой средой тип данных.

В UML атрибуты, типы которых обозначают классы, представляются не в секции атрибута, а с помощью ассоциации (association) между классами.

Методы локализации:

- By value (по значению) – значения атрибута хранятся в пределах адресного пространства, выделенного для класса;
- By reference (по ссылке) – значение атрибута хранится вне адресного пространства, выделенного для объекта данного класса, но у объекта имеется указатель на этот атрибут;
- Unspecified (не определен) – метод локализации данного атрибута не определен.

Операции. Методы (method) – процедуры, реализующие объявленные в классе операции.

Операция (метод) вызывается с помощью отправленного ей сообщения. Операция может содержать список параметров, которым в вызове сообщения могут быть присвоены определенные значения, и может возвращать значение вызывающему объекту.

Сигнатура (signature) – имя операции со списком типов формальных аргументов.

Формат записи операции: <квантор видимости> <имя операции> (список параметров): <выражение типа возвращаемого значения> {строка-свойство}.

Имя операции – единственный обязательный элемент синтаксического обозначения операции, должно начинаться со строчной буквы и записываться без пробелов.

Список параметров – перечень разделенных запятой формальных параметров с форматом записи <направление параметра> <имя параметра>; <выражение типа> = <значение параметра по умолчанию>.

Типы операций:

- реализаций – реализуют бизнес-функциональность;
- управления (manager operations) – управляют созданием и разрушением объектов – конструкторы и деструкторы классов;
- доступа (access operations) – управляют доступом к атрибутам класса;
- вспомогательные (helper operations) – для выполнения ответственостей, о которых другие классы не должны знать – закрытые и защищенные операции класса.

7.4 Диаграммы классов (ДК). Типы классов.

См. п. 6.3 ([ссылка](#)).

Диаграмма классов предназначена для представления статической структуры модели системы в терминологии классов объектно-ориентированного программирования.

Виды классов:

- **Конкретный класс.** Класс, на основе которого могут быть созданы экземпляры или объекты;
- **Абстрактный класс.** Класс, который не имеет экземпляров или объектов. Позволяют определить поведение по умолчанию и заставить подклассы обеспечить конкретное поведение;
- **Параметризованный класс (шаблон).** Предназначен для обозначения класса, который имеет нефиксированные формальные параметры. Он определяет множество классов, каждый из которых может быть получен связыванием этих параметров с действительными значениями;
- **Интерфейс.** Именованное множество операций, которые характеризуют поведение отдельного элемента модели. В контексте UML является специальным случаем класса, у которого имеются операции, но отсутствуют атрибуты.

В концепции интерфейса лежит абстрактный класс, но интерфейс предотвращает реализации методов или функций, оставляя эту обязанность классу, который реализует данный интерфейс.

Преимущества интерфейса:

- множественное наследование;
- слабая связанность;
- программа-интерфейс не реализуется;
- полиморфизм с динамическим связыванием;
- абстрактные уровни, разделение функциональности.

7.5 Диаграммы классов (ДК). Связи классов. Ассоциации. Кратность ассоциации. Зависимости. Агрегации. Обобщение. Определение супер- и подтипов.

Диаграмма классов — диаграмма, демонстрирующая классы системы, их атрибуты, методы и взаимосвязи между ними. Входит в UML.

Связи классов:



1) **Ассоциация.** Если два класса связаны друг с другом ассоциацией, то это означает, что их экземпляры (объекты) определенным образом связаны друг с другом, например: вызывают методы друг друга, работают с общей памятью и так далее. Ассоциация как связь между классами обязательно переходит в связь между экземплярами этих классов. Каждый конец ассоциации называется ролью (role). Она может означать следующее: 1. Один класс осуществляет взаимодействие с другим каким-либо образом. 2. Один класс включает в себя экземпляр другого класса. 3. Один класс включает в себя несколько экземпляров другого класса.

Значение кратности определяет, сколько экземпляров одного класса может быть корректно связано с экземпляром другого класса в некоторый конкретный момент времени.

2) **Зависимость.** Зависимость (Dependency) - это семантическое отношение между двумя сущностями, при котором изменение одной из них (независимой сущности) может отразиться на семантике другой (зависимой). Используется, когда некоторое изменение одного элемента модели может потребовать изменения другого зависимого от него элемента модели.

3) **Агрегация.** Агрегация (aggregation) – это ассоциация между классом A (часть) и классом B (целое), которая означает, что экземпляры (один или несколько) класса A входят в состав экземпляра класса B. Отношение агрегации показывает, из каких компонентов состоит система и как они связаны между собой. Части системы никак не обязаны наследовать ее свойства и поведение, поскольку являются вполне самостоятельными сущностями.

4) **Обобщение.** Отношение обобщения является отношением между более общим элементом (родителем или предком) и более частным или специальным элементом (дочерним или потомком). Данное отношение может использоваться для представления взаимосвязей между пакетами, классами, вариантами использования и другими элементами языка UML. Применительно к диаграмме классов данное отношение описывает иерархическое строение классов и наследование их свойств и поведения. При этом предполагается, что класс-потомок обладает всеми свойствами и поведением класса-предка, а также имеет свои собственные свойства и поведение, которые отсутствуют у класса-предка.

Супертип — это сущность, включающая разные подтипы, которые необходимо представить в модели данных. **Подтип** — это сущность, являющаяся членом супертипа, но выполняющая отдельную роль в нем.

Супертип может иметь несколько разных подтипов. Так, например, подтипы: АССИСТЕНТ, СТАРШИЙ ПРЕПОДАВАТЕЛЬ, ДОЦЕНТ, ПРОФЕССОР являются членами супертипа ПРЕПОДАВАТЕЛЬ. Это означает, что

каждый экземпляр подтипа является в то же время и экземпляром супертипа. Связь между супертипов и подтипов относится к типу "один к одному".

7.6 Диаграммы взаимодействия: диаграммы последовательности. Работа с сообщениями.

Параметры синхронизации и частоты. Работа с примечаниями.

Диаграмма последовательности (sequence diagram) – это способ описания поведения системы на основе указания последовательности передаваемых сообщений.

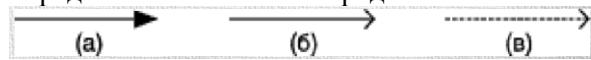
На диаграмме последовательности применяют один основной тип сущностей – экземпляры взаимодействующих классов, компонентов и действующих лиц, и один тип отношений – связи, по которым происходит обмен сообщениями.

Для обозначения самих взаимодействующих объектов применяется стандартная нотация – прямоугольник с именем экземпляра классификатора. Пунктирная линия, выходящая из него, называется линией жизни. Фигуры в виде узких полосок показывают отрезки времени, в течении которых объект владеет потоком управления или другими словами имеет место активация объекта. Составные шаги взаимодействия позволяют на диаграмме последовательности отражать алгоритмические аспекты протокола взаимодействия.



Работа с сообщениями

Процесс взаимодействия объектов реализуется посредством сообщений, которые посылаются одними объектами другим. Сообщения изображаются в виде горизонтальных стрелок с именем сообщения и образуют определенный порядок относительно времени своей инициализации.



- 1) Синхронные (Стрелка с направлением) - сообщение, при котором отправитель ждет, пока получатель выполнит операцию и вернет результат.
- 2) Асинхронные (Стрелка с открытой стрелкой на конце) - сообщение, при котором отправитель продолжает выполнение своих операций без ожидания ответа от получателя.
- 3) Возврат (Стрелка с прерывистой линией) - показывает возврат от получателя к отправителю.

Параметры синхронизации и частота используются для подчеркивания тайминга и частоты передачи сообщений или выполнения определенных операций в системе. Сообщения, расположенные на диаграмме последовательности выше, передаются раньше тех, которые расположены ниже.

В UML 2.0 диаграммы последовательности могут содержать блоки разных типов:

- 1) *alt* – несколько альтернатив (каждая альтернатива – часть блока, помеченная сторожевым условием);
- 2) *opt* – необязательный блок (взаимодействие выполняемое при истинности сторожевого условия);
- 3) *par* – блок из параллельно выполняемых разделов;
- 4) *loop* – цикл (пока истинно условие);
- 5) *region* – критический участок;
- 6) *neg* – неверное взаимодействие;
- 7) *ref* – ссылка на другую диаграмму;
- 8) *sd* – блок, включающий диаграмму последовательности целиком.

Примечание (комментарий) дает возможность прикреплять к элементам различные примечания. Комментарий не несет семантической силы, но может содержать информацию, полезную для разработчика моделей.

7.7 Диаграммы взаимодействия: кооперативные диаграммы. Основные обозначения для диаграмм кооперации.

Диаграмма кооперации предназначена для описания поведения системы на уровне отдельных объектов, которые обмениваются между собой сообщениями, чтобы достичь нужной цели или реализовать некоторый вариант использования. С точки зрения аналитика или архитектора системы в проекте важно представить структурные связи отдельных объектов между собой. Такое представление структуры модели как совокупности взаимодействующих объектов и обеспечивает диаграмма кооперации.

Основные обозначения для диаграмм кооперации:

1. Объекты делятся на две категории: пассивные и активные.

a. Пассивный объект оперирует только данными и не может инициировать деятельность по управлению другими объектами. Однако пассивные объекты могут посыпать сигналы в процессе выполнения запросов, которые они обрабатывают.

b. Активный объект (active object) имеет собственный процесс управления и может инициировать деятельность по управлению другими объектами.

2. Мультиобъекты. На диаграмме кооперации мультиобъект используется для того, чтобы показать операции и сигналы, которые адресованы всему множеству анонимных объектов. При этом стрелка взаимосвязи относится ко всему множеству объектов, которые обозначает данный мультиобъект. На диаграмме кооперации может быть явно указано отношение агрегации (композиции) между мультиобъектом и отдельным объектом из его множества.

3. Составной объект. Составной объект (composite object) или объект-композит предназначен для представления объекта, имеющего собственную структуру и внутренние потоки (нити) управления. Составной объект является экземпляром классакомпозита, который связан отношением композиции со своими частями. Аналогичные отношения связывают между собой и соответствующие объекты.

4. Сообщения. Сообщение (message) — спецификация передачи информации от одного элемента модели к другому с ожиданием выполнения определенных действий со стороны принимающего элемента.

5. Стереотипы сообщений. В языке UML предусмотрены стандартные действия, выполняемые в ответ на получение соответствующего сообщения. Они могут быть явно указаны на диаграмме кооперации в форме стереотипа перед именем сообщения, к которому они относятся, или выше его. В этом случае они записываются в угловых кавычках.

7.8 Диаграммы состояний. Описание динамики поведения системы. Понятие действий и деятельности.

Диаграмма состояний (statechart diagram) - диаграмма, которая представляет конечный автомат.

Главное назначение диаграммы состояний - описать возможные последовательности состояний и переходов, которые в совокупности характеризуют поведение моделируемой системы в течение всего ее жизненного цикла.

Диаграммы состояний чаще всего используются для описания поведения отдельных систем и подсистем. Они также могут быть применены для спецификации функциональности экземпляров отдельных классов, т. е. для моделирования всех возможных изменений состояний конкретных объектов. Диаграмма состояний по существу является графом специального вида, который служит для представления конечного автомата.

Состояния (state): Простые (simple), Составные (composite), Специальные (pseudo).

Переходы (transition): Начало (source), Событие (event), Условие (guard condition), Действие (action), Конец (target).

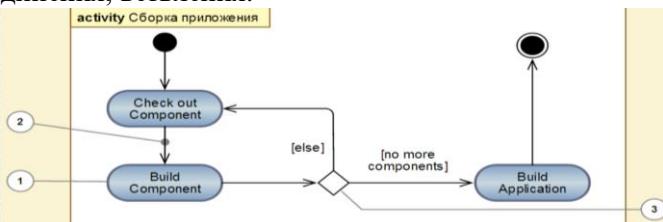
Динамика поведения системы описывает изменение ее состояний во времени и в ответ на действия, воздействия или события. Например, диаграмма состояний может показать, как система меняет свое состояние, когда на нее поступают определенные входные данные или при наступлении определенных условий.

Деятельность (Activity) - это продолжающееся неатомарное вычисление внутри автомата. **Действие (Action)** - это атомарное вычисление, которое приводит к смене состояния или возврату.

7.9 Диаграммы деятельности. Рекомендации по использованию.

Диаграмма деятельности – еще один способ описания поведения, который визуально напоминает блок-схему алгоритма. Однако за счет модернизированных обозначений, согласованных с объектно-ориентированным подходом, а главное, за счет новой семантической составляющей (свободная интерпретация сетей Петри), диаграмма деятельности UML является мощным средством для описания поведения системы.

На диаграмме деятельности применяют один основной тип сущностей – действие, и один тип отношений – переходы (передачи управления и данных). Также используются такие конструкции как разветвления, слияния, соединения, ветвлениия.



Элементы диаграммы деятельности:

- Главной сущностью этого типа диаграмм является активность (activity) - активное состояние системы, в котором она выполняет некоторую работу. После ее завершения происходит переход в другую активность.
- На диаграмме должны присутствовать символы начала (start) и конца (finish).

- Далее, на диаграмме может использоваться параллельный разветвитель (fork), который запускает несколько одновременно работающих веток. Такие ветки могут объединяться (все или только часть) конструкцией под названием параллельный соединитель (join).
- Наконец, на диаграмме могут использоваться символы логического ветвления и логического соединения (decision). На ветках, идущих из логического ветвления, обозначаются условия перехода.

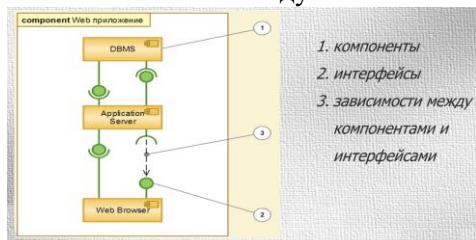
Рекомендации по построению диаграммы деятельности.

Диаграмма строится для отдельного класса, варианта использования, отдельной операции класса или целой подсистемы. С одной стороны, на начальных этапах проектирования, когда детали реализации деятельности в проектируемой системе неизвестны, построение диаграммы деятельности начинают с выделения поддеятельностей, которые в совокупности образуют деятельность подсистем. В последующем, по мере разработки диаграмм классов и состояний, эти под-деятельности уточняются в виде отдельных вложенных диаграмм деятельности компонентов подсистем, какими выступают классы и объекты.

7.10 Диаграммы компонентов.

Диаграмма компонентов (component diagram) – показывает взаимосвязи между модулями (логическими или физическими), из которых состоит моделируемая система. Основной тип сущностей на диаграмме компонентов – это сами компоненты, а также интерфейсы, посредством которых указывается взаимосвязь между компонентами. На диаграмме компонентов применяются следующие отношения:

- реализации между компонентами и интерфейсами (компонент реализует интерфейс);
- зависимости между компонентами и интерфейсами (компонент использует интерфейс).



Компонент (component) – это модульный фрагмент логического представления системы, взаимодействие с которым описывается набором обеспеченных и требуемых интерфейсов.

Интерфейс – это именованный набор составляющих, описывающий контракт между поставщиками и потребителями услуг.

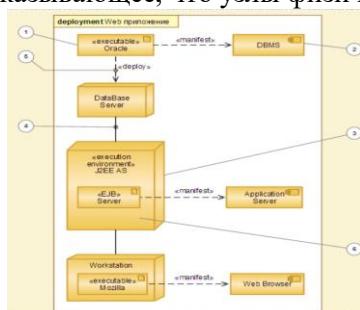
Диаграмма компонентов разрабатывается для следующих целей:

- Визуализации общей структуры исходного кода программной системы.
- Спецификации исполнимого варианта программной системы.
- Обеспечения многократного использования отдельных фрагментов программного кода.
- Представления концептуальной и физической схем баз данных.

7.11 Диаграммы размещения.

Диаграммы размещения или **диаграмма развертывания (deployment diagram)** наряду с отображением состава и связей элементов системы показывает, как они физически размещены на вычислительных ресурсах во время выполнения.

Таким образом, на диаграмме развертывания, по сравнению с диаграммой компонентов, добавляется два типа сущностей: артефакт, который является реализацией компонента и узел (может быть как классификатор, описывающий тип узла, так и конкретный экземпляр), а также отношение ассоциации между узлами, показывающее, что узлы физически связаны во время выполнения.



Артефакт (artifact) – это физическая и замещаемая часть системы, несущая физическую информацию («биты»). Обычно артефакт представляет собой физический пакет с исходным или исполняемым кодом. Изображается как прямоугольник, снабженный ключевым словом «artifact», расположенным над его именем.

Реализацией компонента или манифестацией – это отношение зависимости со стереотипом «manifest», связывающее элемент модели (например, класс или компонент) и его физическую реализацию в виде артефакта.

Узел (node) – это физический вычислительный ресурс, участвующий в работе системы.

Итак, перечислим цели, преследуемые при разработке диаграммы развертывания:

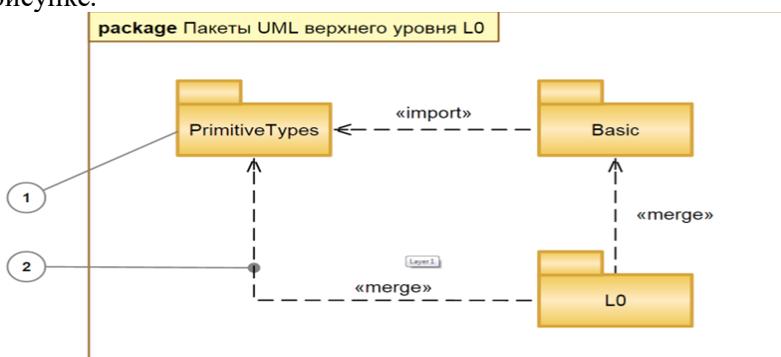
- Определить распределение компонентов системы по ее физическим узлам.
- Показать физические связи между всеми узлами реализации системы на этапе ее исполнения.
- Выявить узкие места системы и реконфигурировать ее топологию для достижения требуемой производительности.

Для обеспечения этих требований диаграмма развертывания разрабатывается совместно системными аналитиками, сетевыми инженерами и системотехниками.

7.12 Пакеты в языке UML. Отображение архитектуры с использованием пакетов UML. Механизм пакетов. Уровни и разделы многоуровневой архитектуры.

Пакет – основной способ организации элементов модели в языке UML. Каждый пакет владеет всеми своими элементами, т. е. теми элементами, которые включены в него. Про соответствующие элементы пакета говорят, что они принадлежат пакету или входят в него. При этом каждый элемент может принадлежать только одному пакету. В свою очередь, одни пакеты могут быть вложены в другие пакеты. В этом случае первые называются подпакетами, поскольку все элементы подпакета будут принадлежать более общему пакету. Тем самым для элементов модели задается отношение вложенности пакетов, которое представляет собой иерархию.

Диаграмма пакетов – единственное средство, позволяющее управлять сложностью самой модели. Основные элементы нотации – пакеты 1 и зависимости с различными стереотипами 2, применяемые на диаграмме, показаны на рисунке.



Основные концепции и механизмы пакетов в UML:

- Определение пакета (Package Definition) - пакет представляет собой группировку элементов модели в целях управления и организации.
- Элементы пакета (Package Elements) - пакет может содержать в себе различные элементы, такие как классы, интерфейсы, диаграммы, другие пакеты и т.д.
- Вложенные пакеты (Nested Packages) - пакеты могут вложаться друг в друга, образуя иерархию для логического иерархического представления системы.
- Импорт пакета (Package Import) - импорт пакета позволяет использовать элементы из одного пакета в другом пакете без необходимости копирования.

Уровни пакетов:

- Высший уровень (Top Level) - это самый верхний уровень иерархии. Здесь часто размещают общие элементы, касающиеся всей системы.
- Промежуточные уровни - эти уровни могут включать подпакеты или элементы, относящиеся к конкретным аспектам системы. Они служат для организации и структурирования элементов модели на более низком уровне абстракции.
- Низший уровень (Bottom Level) - этот уровень может содержать конкретные элементы системы, такие как классы, интерфейсы, диаграммы и другие.

Разделы пакетов:

- Вложенные пакеты (Nested Packages) - пакеты могут вложаться друг в друга, образуя иерархию и уровни. Вложенные пакеты позволяют организовывать элементы системы в структуре, удобной для понимания.
- Вложенные элементы - внутри каждого пакета могут находиться не только другие пакеты, но и конкретные элементы модели, такие как классы, диаграммы, и т.д.

ТЕМА 8. ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

8.1 Объектно-ориентированный анализ

- Анализ предметной области

- Модель анализа.
 - Идентификация классов. Приемы выделения классов.
 - Выделение ассоциаций.
 - Группировка классов в пакеты
- Модель состояний предметной области
 - Анализ приложения. Модель взаимодействия. Этапы построения.
 - Артефакты этапа анализа

Анализ предметной области – это процесс изучения и понимания основных характеристик и составляющих элементов определенной предметной области с целью создания эффективных решений в данной области.

Этап анализа делится на анализ предметной области и анализ приложения.

Статическая модель предметной области описывает общие знания о приложении. Такая модель обычно включает модели классов и состояний, но очень редко – модель взаимодействия.

Динамическая модель приложения отражает наиболее важные артефакты, видимые пользователем и требующие их одобрения. Для приложения важнее всего модель использования (взаимодействия), однако нельзя забывать модель классов и состояний.

Модель анализа.

Последовательность моделирования



Основу модели анализа составляют:

- Модель предметной области
- Модель вариантов использования (прецедентов)

В модель прецедентов, кроме непосредственно диаграммы прецедентов могут включать реализацию прецедентов и системные диаграммы последовательностей.

В описании модели анализа может быть отражено:

- Введение: Текстовое описание, которое служит в качестве краткого введения в модель.
- Пакеты анализа: Пакеты в модели, представляющие иерархию.
- Классы: Классы в модели, принадлежащие пакетам.
- Взаимосвязи: Взаимосвязи в модели, принадлежащие пакетам.
- Реализации вариантов использования: Реализации вариантов использования в модели, принадлежащие пакетам.
- Диаграммы: Диаграммы в модели, принадлежащие пакетам

Идентификация классов. Приемы выделения классов.

Классы в модели идентифицируются в результате проведения трех частично независимых процессов:

- анализа предметной области,
- согласования уже построенной модели
- применения теоретических соображений.

Ниже приведены три приема выделения классов, самых простых, а потому самых действенных и широко применимых:

1. словарь предметной области – это набор основных понятий (сущностей) данной предметной области.

2. реализация вариантов использования. Если при реализации ВИ применяются диаграммы взаимодействия, то в этом процессе в качестве побочного эффекта выделяются некоторые классы непосредственно, поскольку на диаграммах кооперации и последовательности основными сущностями являются объекты, которые по необходимости нужно отнести к определенным классам.

3. шаблоны проектирования или паттерн (design pattern) в разработке программного обеспечения — повторяющаяся архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста. Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы или объекты приложения будут использоваться.

Выделение ассоциаций

Структурные отношения между двумя и более классами являются ассоциацией. Если один класс ссылается на другой класс, это тоже ассоциация. Для ссылок на классы не следует использовать атрибуты, их правильнее представлять в модели в виде ассоциации. Ассоциации показывают отношения между классами на том же уровне абстракции, на котором находятся сами классы, тогда как атрибуты скрывают зависимости и их двухстороннюю природу.

Группировка классов в пакеты

Последний этап моделирования - группировка классов в пакеты. Пакеты делают модель более удобной с точки зрения построения, печати и просмотра. Помещая определённые классы и ассоциации в пакет, вы делаете семантическое утверждение. Классы одного пакета связаны друг с другом сильнее, чем классы в разных пакетах. Нужно стараться ограничивать ассоциации рамками одного пакета некоторые классы можно повторять в разных пакетах. Чтобы распределить классы по пакетам, ищите точки сочленения - классы, являющиеся единственным соединением двух частей модели, не имеющих между собой других связей такой класс образует мост между двумя пакетами. Например, в системе управления файлами класс Файл является точкой соединения сочленения между структурой каталогов и содержимым файла. Страйтесь выбирать пакеты так, чтобы сокращать количество пересечений на диаграммах классов.

Модель состояний предметной области

Объекты предметные области за время своей жизни сменяют несколько качественно различных состояний. В этих состояниях они могут иметь разные ограничения на значение атрибутов, разные ассоциации или кратности, выполнять разные операции или иметь разное поведение и т.д. Диаграмма состояний описывает различные состояния, в которых будет находиться объект, свойства объекта и действующие на них ограничения, события вызывающий переход объекта из одного состояния в другое. Большинство классов предметы области не требует использования диаграмм состояний. Для их описания достаточно списка операций. Модель состояний может помочь пониманию поведение тех классов, могут находиться существенно разных состояниях.

Модель состояний предметной области конструируются в несколько этапов:

- выявление классов, обладающих разными состояниями;
- выделение состояний; выделение событий;
- построении диаграмм состояний;
- проверка диаграмм состояний.

Для модели состояний предметной области важны только те события, который вызывает переходы. Информацию, содержащуюся в событии, следует представлять в форме списка его параметр.

Анализ приложения. Модель взаимодействия. Этапы построения.

Анализ приложения представляет собой процесс изучения и оценки функциональности, производительности, безопасности и других аспектов приложения. В ходе анализа проводится тестирование приложения на различных устройствах и платформах,

На этапе модели взаимодействия можно нарисовать предварительную диаграмму вариантов использования. На ней следует показать действующие лица и варианты использования, соединив их между собой. Обычно вариант использования можно соединить с действующим лицом, которое его инициирует, но вообще говоря в нём могут участвовать и другие действующие лица.

Большинство моделей предметной области являются статическими. Для сложных и неочевидных вариантов использования нужно построить диаграммы деятельности.

Модель взаимодействия для приложения строится в несколько этапов:

1. Определить границу системы.
2. Выделить действующие лица.
3. Выделить варианты использования.
4. Выделить начальные и конечные события.
5. Подготовить типовые сценарии.
6. Добавить сценарии, описывающие вариации и исключительные ситуации.
7. Выделить внешние события.
8. Построить диаграммы деятельности для сложных вариантов использования.
9. Структурировать действующие лица и варианты использования.
10. Выполнить проверку по модели классов предметный области

Артефакты этапа анализа

- Модель предметной области (диаграмма классов – ключевых абстракций)
- Модель вариантов использования
- Сценарии вариантов использования
 - типовые сценарии
 - сценарии, описывающие вариации и исключительные ситуации.
- Диаграммы деятельности для сложных и неочевидных вариантов использования

8.2 Объектно-ориентированной проектирование. Основные принципы проектирования.

- Введение
 - Ключевые понятия архитектуры ПО (декомпозиция, компоненты, ..., уровни)
 - Ключевые вопросы и проблемы архитектуры
 - Многоуровневая объектно-ориентированная архитектура
 - Типы часто используемых подсистем
- Архитектурное проектирование
 - Проектные механизмы
 - Архитектурные представления (виды)
 - Архитектурные системные паттерны
 - Этапы архитектурного проектирования
 - Структурирование системы
 - Моделирование управления
 - Декомпозиции подсистем на модули
 - Типовая архитектура приложения. Слои. Сквозная функциональность.
 - Моделирование UML
 - Отображение архитектуры с использованием пакетов UML.
 - Компоненты. Диаграмма компонентов.
 - Диаграмма размещения.
- Проектирование элементов системы (детальное проектирование).
 - Этапы детального проектирования
 - Проектирование подсистем
 - Проектирование классов
 - Проектирование базы данных
 - Проектирование интерфейса

Введение

Объектно-ориентированное проектирование — это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

Основные принципы проектирования помогают создать архитектуру, которая будет следовать проверенным подходам, обеспечит минимизацию затрат, простоту обслуживания, удобство использования и расширяемость. Рассмотрим основные принципы:

- Разделение функций. Разделите приложение на отдельные компоненты, по возможности, с минимальным перекрытием функциональности. Важным фактором является предельное уменьшение количества точек соприкосновения, что обеспечит высокую связность (high cohesion) и слабую связанность (low coupling).
- Принцип единственности ответственности. Каждый отдельно взятый компонент или модуль должен отвечать только за одно конкретное свойство/функцию или совокупность связанных функций.
- Принцип минимального знания (также известный как Закон Деметера (Law of Demeter, LoD)). Компоненту или объекту не должны быть известны внутренние детали других компонентов или объектов.
- Не повторяйтесь (Don't repeat yourself, DRY). Намерение должно быть обозначено только один раз. В применении к проектированию приложения это означает, что определенная функциональность должна быть реализована только в одном компоненте и не должна дублироваться ни в одном другом компоненте.
- Минимизируйте проектирование наперед. Проектируйте только то, что необходимо. Если требования к приложению четко не определены, или существует вероятность изменения дизайна со временем, старайтесь не тратить много сил на проектирование раньше времени. Этот принцип называют YAGNI («You ain't gonna need it»).

Ключевые понятия архитектуры ПО (декомпозиция, компоненты, ..., уровни)

Декомпозиция — операция мышления, состоящая в разделении целого на части. Разработка архитектуры системы — это процесс разбиения большой системы на более мелкие части. Для обозначения этих частей придумано множество названий: программы, компоненты, подсистемы, модули, ...

Компоненты. Программы состоят из отдельных компонент. Компонента поставляется пользователю как двоичный код, скомпилированный, скомпонованный и готовый к использованию. Доступ к этому коду осуществляется через точно документированный интерфейс. Во время выполнения компоненты подключаются к другим компонентам, формируя программу.

Каркасы (фреймворки). В самом простом смысле каркас может означать любой фрагмент разработанного и проверенного ПО, который повторно используется в нескольких проектах разработки ПО. Фреймворк — это набор всевозможных библиотек (инструментов) для быстрой разработки повседневных (рутинных) задач. Чаще всего использует одну из распространенных архитектур приложения (к примеру, MVC) для разделения проекта на логические сегменты (модули). Главная цель фреймворка — предоставить программисту удобную среду для проекта с большим и хорошо расширяемым функционалом.

Шаблоны. Шаблон проектирования или паттерн в разработке программного обеспечения — повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста. Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы или объекты приложения будут использоваться.

- Многоуровневый шаблон (Layered pattern).
- Клиент-серверный шаблон (Client-Server pattern).
- Шаблон посредника (Broker pattern).
- Шаблон «Модель-Представление-Контроллер» (Model-View-Controller pattern).

Уровни. Понятия слой (layer) и уровень (tier) очень близки. Слой используется для логического разделения, а уровень для физического разворачивания. Термин слой применяется, чтобы подчеркнуть, что слои вовсе не обязательно должны располагаться на разных машинах. Отдельный слой бизнес-логики может функционировать как на персональном компьютере “рядом” с клиентским слоем интерфейса, так и на сервере базы данных.

Ключевые вопросы и проблемы архитектуры

Аутентификация и авторизация

1. Как хранить идентификационные данные зарегистрированных пользователей?
2. Как аутентифицировать запросы?
3. Как авторизовывать запросы?
4. Как передавать идентификационные данные пользователей между слоями приложения?

Кэширование и состояние

1. Как определить правила удаления данных из кэша из-за устаревания или переполнения?
2. Как загружать данные кэша?
3. Как следить за кэшем?
4. Как синхронизовать между собой множественные кэши?
5. Как определить какой способ кэширования даёт наибольшую производительность и масштабируемость для определённого сценария и конфигурации.
6. Как определить какая технология кэширования удовлетворяет требованиям приложения насчёт безопасности, лёгкости управления и мониторинга.
1. Как выбрать эффективную стратегию кэширования?
2. Как увеличить производительность с помощью кэширования?
3. Как улучшить безопасность с помощью кэширования?
4. Как улучшить стабильность с помощью кэширования?
5. Как поддерживать кэш в актуальном состоянии?
6. Как определить когда и каким образом использовать специализированный кэш?
7. Как определить какие данные кэшировать?
8. Как определить куда кэшировать данные?

Взаимодействие

1. Как передавать информацию между слоями приложения?
2. Как совершать асинхронные операции?
3. Как передавать секретные данные?

Композиция

1. Как спроектировать структуру приложения?
2. Как спроектировать слабую связанность между модулями?
3. Как работать с зависимостями в слабосвязанном режиме?

Управление конфигурацией

1. Как определить какая информация подлежит конфигурированию?
2. Как определить где и каким образом сохранять конфигурационную информацию?
3. Как работать с секретной информацией?
4. Как работать с конфигурационной информацией в кластере.

Связанность и сцепление

1. Как разделить функциональности?
2. Как структурировать приложение?
3. Как выбрать подходящее разбиение на слои?
4. Как задать границы между частями системы?

Доступ к данным

1. Как управлять соединениями БД? 2. Как управлять исключениями? 3. Как улучшить производительность?
4. Как улучшить управляемость? 5. Как работать с бинарными данными (BLOBs)? 6. Как управлять транзакциями?

Поток операций

1. Сильная связанность. 2. Негибкие процессы. 3. Проблемы с приоритетами и взаимными блокировками (race and deadlock issues).

Ключевые проблемы

1. Хранение идентификационных данных в открытом виде в файлах. 2. Передача идентификационных данных в открытом виде по сети. 3. Аккаунты с излишними привилегиями. 4. Долгоживущие сессии. 5. Смешивание персонализации и аутентификации. 6. Использование единственного контроллера доступа (gatekeeper). 7. Отсутствие ограничений на доступ к системным ресурсам для приложения. 8. Отсутствие ограничений на доступ к БД кроме определённых хранимых процедур (stored procedures). 9. Неадекватное разделение прав доступа.

Основные вопросы проектирования:

1. Определение типа приложения
2. Выбор стратегии развертывания
3. Выбор соответствующих технологий
4. Выбор показателей качества
5. Решение о путях реализации сквозной функциональности

Многоуровневая объектно-ориентированная архитектура

Данный шаблон используется для структурирования программ, которые можно разложить на группы неких подзадач, находящихся на определенных уровнях абстракции. Каждый слой предоставляет службы для следующего, более высокого слоя. Чаще всего в общих информационных системах встречаются следующие 4 слоя:

1. Слой представления (также известен как слой пользовательского интерфейса)
2. Слой приложения (также известен как слой сервиса)
3. Слой бизнес-логики (также известен как уровень предметной области)
4. Слой доступа к данным (также известен как уровень хранения данных).

Трехуровневая архитектура - это клиент-серверный шаблон архитектуры программного обеспечения, в котором пользовательский интерфейс (представление), функциональная логика процесса ("бизнес-правила"), хранилище компьютерных данных и доступ к данным разрабатываются и поддерживаются как независимые модули, чаще всего на отдельных платформах.

Плюсы: Одним низким слоем могут пользоваться разные слои более высокого ранга. Слои упрощают стандартизацию, т.к. мы четко определяем уровни. Изменения вносятся внутри какого-то одного слоя, при этом остальные слои остаются неизменными.

Минусы: Наличие уровней снижает производительность. Этот шаблон не подходит для высокопроизводительных приложений: проходить несколько уровней архитектуры для выполнения бизнес-запроса — это неэффективно. Кроме того, добавление уровней увеличивает полную стоимость и усложняет систему.

Применение Использовать этот подход следует для небольших, простых приложений и веб-сайтов. Также этот шаблон хорошо подходит, если бюджет и время ограничены.

Типы часто используемых подсистем



Как видите, в итоге все подсистемы начинают напрямую взаимодействовать, что поднимает несколько важных вопросов: в скольких разных частях системы нужно хоть немного разбираться разработчику, желающему изменить какой-то аспект подсистемы графических операций? что будет, если вы попытаетесь задействовать данный модуль бизнесправил в другой системе? что будет, если вы захотите включить в систему новый пользовательский интерфейс (например, интерфейс командной строки, удобный для проведения тестирования)? что произойдет, если вы захотите перенести модуль хранения данных на удаленный компьютер? Стрелки между подсистемами можно рассматривать как шланги с водой. Если вам захочется «выдернуть» одну из подсистем, к ней наверняка будут подключены несколько шлангов. Чем больше шлангов вам нужно будет отсоединить и подключить заново, тем сильнее вы промокнете. Архитектура системы должна быть такой, чтобы замена подсистем требовала как можно меньше возни со шлангами.

Чтобы соединения подсистем были понятными и легкими в сопровождении, старайтесь поддерживать простоту отношений между подсистемами. Самым простым отношением является то, при котором одна подсистема вызывает методы другой. Более сложное отношение имеет место, когда одна подсистема содержит классы другой. Самое сложное отношение — наследование классов одной подсистемы от классов другой.

Придерживайтесь одного разумного правила: диаграмма системного уровня вроде той, что показана на рисунке, должна быть ациклическим графом. Иначе говоря, программа не должна содержать циклических отношений, при которых класс A использует класс B, класс B использует класс C, а класс C — класс A.

Архитектурное проектирование

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

- идентификацию подсистем;
- определение основных принципов управлениями подсистемами и их взаимодействия;

Архитектурный анализ связан с определением и удовлетворением нефункциональных (например, качественных) требований к системе в контексте функциональных требований. Архитектура предприятия описывает структуры бизнеса и его внутренние процессы (поток информации, взаимодействие между отделами и т.д.). Системная архитектура описывает элементы и их взаимодействие между собой в рамках всей системы и включает как программные, так и аппаратные элементы.

Архитектура ПО является реализацией нефункциональных требований к системе, в то время как детальное проектирование ПО является реализацией функциональных требований. Архитектура ПО, которую также можно представить себе в виде разработки стратегии — это деятельность, связанная с определением глобальных ограничений, накладываемых на проектирование системы, такие как выбор парадигмы программирования, архитектурных стилей, стандарты разработки ПО, основанные на использовании компонентов, принципы проектирования и ограничения, накладываемые государственным законодательством.

Проектные механизмы

Проектирование архитектуры системы выполняется архитектором. Определяются детали реализации архитектурных (проектных) механизмов, обозначенных в процессе анализа. Например, уточняется способ хранения данных, вводится дублирование данных для повышения надежности системы и т. п. Поскольку практически все механизмы — это некоторые типовые решения (образцы, шаблоны, каркасы), они документируются в проекте (модели) с помощью коопераций со стереотипом <>, при этом структурная часть каждого механизма описывается с помощью диаграмм классов, а поведение — с помощью диаграмм взаимодействия.

Архитектурные представления (виды)

Методика описания архитектуры стандарта базируется на трех уровнях представления архитектуры:

1. Viewpoint - точка зрения (аспект), относительно которой мы рассматриваем архитектуру или дизайн приложения.
2. View - представление, некоторая часть архитектурного описания приложения, например, функциональная, логическая или посвященная безопасности, например, security view.

Архитектурные виды можно поделить на 3 основных типа:

- Модульные виды (англ. module views) — показывают систему как структуру из различных программных блоков. Примеры: 1. Декомпозиция — состоит из модулей в контексте отношения «является подмодулем», 2. Использование — состоит из модулей в контексте отношения «использует» (т.е. один модуль использует сервисы другого модуля), 3. Вид уровней — показывает структуру, в которой связанные по функциональности модули объединены в группы (уровни), 4. Вид классов/обобщений — состоит из классов, связанные через отношения «наследуется от» и «является экземпляром»
- Компоненты-и-коннекторы (англ. component-and-connector views) — показывают систему как структуру из параллельно запущенных элементов (компонентов) и способов их взаимодействия (коннекторов). Примеры: 1. Процессный вид — состоит из процессов, соединённых операциями коммуникации, синхронизации и/или исключения, 2. Параллельный вид — состоит из компонентов и коннекторов, где коннекторы представляют собой «логические потоки», 3. Вид обмена данными — состоит из компонентов и коннекторов, которые создают, сохраняют и получают постоянные данные, 4. Вид клиент-сервер — состоит из взаимодействующих клиентов и серверов, а также коннекторов между ними (например, протоколов и общих сообщений)
- Размещение (англ. allocation views) — показывает размещение элементов системы во внешних средах. Примеры: 1. Развёртывание — состоит из программных элементов, их размещения на физических носителях и коммуникационных элементов, 2. Внедрение — состоит из программных элементов и их соответствия файловым структурам в различных средах (разработческой, интеграционной и т.д.), 3. Распределение работы — состоит из модулей и описания того, кто ответственен за внедрение каждого из них

3. Model - модель, содержащая конкретные компоненты и отражающая связи между ними. В качестве моделей можно использовать диаграмму классов для отображения представления предметной области или логического представления архитектуры приложения.

Архитектурные системные паттерны

1. Структурные паттерны:

- Репозиторий
- Клиент/сервер
- Объектно - ориентированный, Модель предметной области (Domain Model), модуль таблицы (Data Mapper)
- Многоуровневая система (Layers) или абстрактная машина
- Потоки данных (конвейер или фильтр)

2. Паттерны управления

- Паттерны централизованного управления

Вызов - возврат (сценарий транзакции - частный случай).

Диспетчер

- Паттерны управления, основанные на событиях

Передача сообщений

Управляемый прерываниями

3. Паттерны, обеспечивающие взаимодействие с базой данных

- Активная запись (Active Record)
- Единица работы (Unit Of Work)
- Загрузка по требованию (Lazy Load)
- Коллекция объектов (Identity Map)
- Множество записей (Record Set)
- Наследование с одной таблицей (Single Table Inheritance)
- Наследование с таблицами для каждого класса (Class Table Inheritance)
- Оптимистическая автономная блокировка (Optimistic Offline Lock)
- Отображение с помощью внешних ключей
- Отображение с помощью таблицы ассоциаций (Association Table Mapping)
- Пессимистическая автономная блокировка (Pessimistic Offline Lock)

(Поле идентификации (Identity Field))

- Преобразователь данных (Data Mapper)
- Сохранение сеанса на стороне клиента (Client Session State)
- Сохранение сеанса на стороне сервера (Server Session State)
- Шлюз записи данных (Row Data Gateway)
- Шлюз таблицы данных (Table Data Gateway)

Этапы архитектурного проектирования

Архитектурным проектированием называют первый этап процесса проектирования, на котором определяются подсистемы, а также структура управления и взаимодействия подсистем. Архитектурное проектирование обеспечивает: идентификацию подсистем; определение основных принципов управлениями подсистемами и их взаимодействия;

Архитектурное проектирование включает в себя три типа деятельности:

1. Структурирование системы. Система разбивается на несколько подсистем, где под подсистемой подразумевается независимый программный компонент. Определяется взаимодействие подсистем.

2. Моделирование управления. Разрабатывается модель связей управления между частями системы.

3. Декомпозиция подсистем на модули. Каждая подсистема разбивается на модули. Определяются типы модулей и межмодульные соединения

Структурирование системы

Система разбивается на несколько подсистем, где под подсистемой подразумевается независимый программный компонент. Определяется взаимодействие подсистем. Структура – это взаимосвязь или отношение между элементами некоторой материальной или концептуальной системы.

На первом этапе процесса проектирования архитектуры система разбивается на несколько взаимодействующих подсистем. На самом абстрактном уровне архитектуру системы можно изобразить в виде блоксхемы, в которой отдельные подсистемы представлены блоками. В процессе структурирования проектировщику необходимо ответить на вопросы: 1. Какие подсистемы можно выделить из системы? 2. Как подсистемы связать друг с другом? 3. Как будут взаимодействовать подсистемы?

Подсистема – это 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.

система → подсистема → модуль

Моделирование управления

В модели структуры системы показаны все подсистемы, из которых она состоит. Для того, чтобы подсистемы функционировали как единое целое, необходимо управлять ими. Разработчик архитектуры должен организовать подсистемы согласно некоторой модели управления, которая дополняла бы имеющуюся модель структуры. Можно выделить два основных типа управления в программных системах:

1. Централизованное управление (В модели централизованного управления одна подсистема выделяется как системный контроллер. Её обязанности – руководить работой других подсистем):

а. модель вызова-возврата. Классическая схема построения программных систем: из одной главной программы по мере необходимости вызываются подпрограммы. Каждая из этих подпрограмм в свою очередь может вызывать другие подпрограммы и т.д. За управление отвечает та подпрограмма, которая выполняется в текущий момент. По окончании выполнения программы управление передается программе, которая ее вызвала. Данная модель применима только для последовательных систем. Недостаток модели состоит в сложной обработке исключительных ситуаций.

б. модель диспетчера. Применяется в параллельных системах. Часто используется в системах реального времени. Один системный компонент назначается диспетчером и управляет запуском, завершением и координированием других процессов системы. Процесс (выполняемая подсистема или модуль) может протекать параллельно с другими процессами. Модель такого процесса применима также в последовательных системах, где управляющая программа вызывает отдельные подсистемы в зависимости от значений некоторых переменных состояния.

2. Управление, основанное на событиях. В моделях централизованного управления управление системой определяется значениями некоторых переменных ее состояния.

- а. модель передачи сообщений (широковещательная модель). В широковещательной модели каждая подсистема уведомляет обработчика о своем интересе к конкретным событиям. Когда событие происходит, обработчик посылает его подсистеме, которая может обработать это событие.
- б. модель, управляемая прерываниями. Для того, чтобы обеспечить быструю реакцию на события, необходимо использовать управление, основанное на прерываниях. Для каждого типа прерываний существует свой обработчик. Каждый тип прерывания ассоциируется с ячейкой памяти, в которой хранится адрес обработчика прерывания. При получении определенного прерывания аппаратный переключатель немедленно передает управление обработчику прерывания. В ответ на событие, вызванное прерыванием, обработчик может запустить или завершить другие процессы (это может определяться установленным приоритетом процессов). Данная модель используется только в жестких системах реального времени, где требуется немедленная реакция на определенные события.

Декомпозиции подсистем на модули

После этапа разработки системной структуры следует этап декомпозиции подсистем на модули. Современные процессы разработки ПО непременно являются итеративными (iterative) процессами с пошаговым наращиванием возможностей (incremental) системы. Модули системы уточняются и преобразуются на этапах анализа, проектирования и реализации. Успех итеративного процесса с пошаговым наращиванием возможностей основывается на раннем выявлении архитектурных модулей системы. На этапе модульной декомпозиции рассматриваются две модели:

1. Объектно-ориентированная модель. Система состоит из набора взаимодействующих объектов. В такой модели модули представляют собой объекты с собственными состояниями и определенными состояниями.

2. Модель потоков данных. Система состоит из функциональных модулей, которые получают на входе данные и преобразуют их некоторым образом в выходные данные (эта модель нам уже знакома). Такой подход часто называют конвейерным.

Типовая архитектура приложения. Слои. Сквозная функциональность.

К сегодняшнему дню исторически сложилось 3 типа приложений:

- Web-приложение – программа с определенным набором функционала, использующая в качестве клиента браузер. Другими словами, если приложению для осуществления бизнес-логики требуется сетевое соединение и наличие на стороне пользователя браузера, то его относят к веб-приложению.
- Десктопные приложения предполагают установку клиента на стороне пользователя. В зависимости от типа операционной системы, процессора, видеокарты и других параметров могут потребоваться разные версии программы.

- Мобильные приложения заточены исключительно для смартфонов и планшетов с учетом установленной там системы (Android, iOS и др.). Многие мобильные приложения фактически являются web-приложениями, так как возможности «движков» браузеров это позволяют. Наиболее динамично на сегодня развиваются web-приложения, так как они для своей работы требуют только установленный браузер на клиентской стороне.



Слой пользовательского интерфейса: В самом верху пирамиды находится слой пользовательского интерфейса (UIL – User Interface Layer). Основная задача этого слоя – предоставлять эргономичный интерфейс пользователю в соответствии с функционалом, описанном в техническом задании. Этот слой просто должен быть красивым, удобным и интуитивно понятным. При этом он должен знать, каким образом он взаимодействует со слоем, находящимся на уровень ниже, как он передаёт туда данные и как он их оттуда принимает для отображения. К главным функциям слоя представления относятся отображение информации и интерпретация вводимых пользователем команд с преобразованием их в соответствующие операции в контексте домена (бизнес-логики) и источника данных.

Слой сервисов: Сервисный слой инкапсулирует логику приложения от его представления и предоставляет набор доступных операций. Для слоя бизнес-логики данный слой фактически перенаправляет все запросы в слой бизнес-логики. В сервисном слое осуществляется предварительная проверка корректности введенных данных.

Слой бизнес-логики: Ниже находится слой бизнес-логики классы, описывающие сущности и основную логику работы приложения (основные алгоритмы работы приложения, вычисления, обработка и т.д.). Бизнес-логика — это реализация правил и ограничений автоматизируемых операций. Логика домена (бизнес-логика или логика предметной области) описывает основные функции приложения, предназначенные для достижения поставленной перед ним цели.

Слой доступа к данным: Ниже находится слой доступа к данным (DAL – Data Access Layer). Это работа с источником данных (базами данных разных поставщиков MSSQL, MySql, Oracle...), или XML файлами, или бинарными данными, или текстовыми файлами, всем тем, в чём можно данные хранить постоянно. Здесь тоже описана логика, но направленная именно на взаимодействие с источником и передачу полученных результатов на уровень выше.

Сквозная функциональность – это аспекты дизайна, которые могут применяться ко всем слоям, компонентам и уровням. Также это те области, в которых чаще всего делаются ошибки, имеющие большое влияние на дизайн. Приведем примеры сквозной функциональности:

- Аутентификация и авторизация.
- Кэширование.
- Связь.
- Управление конфигурацией.
- Управление исключениями.
- Протоколирование и инструментирование.
- Валидация.

Моделирование UML

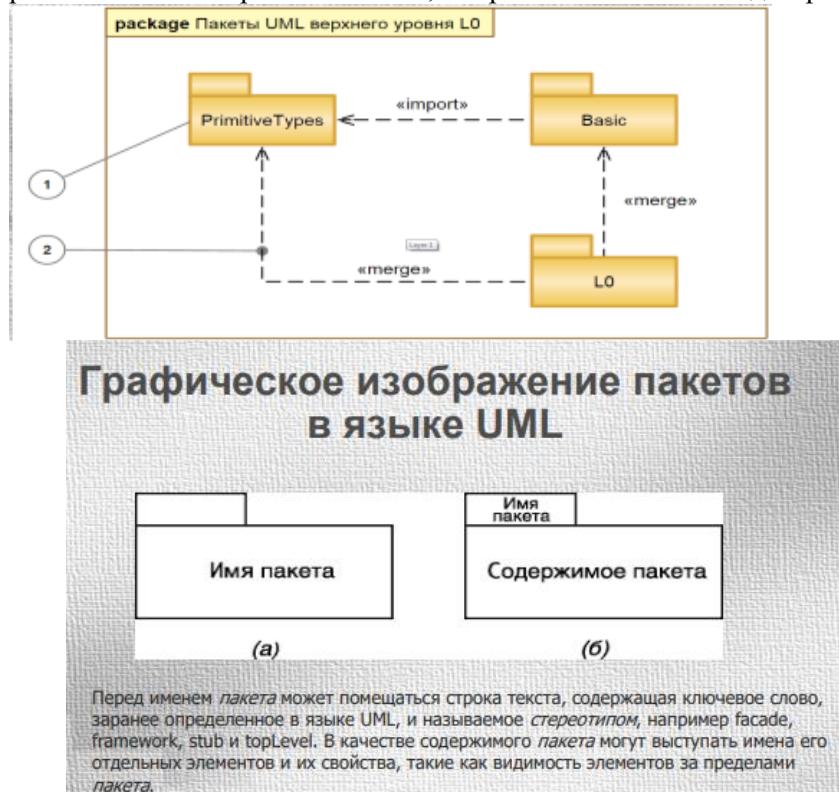
UML, или Unified Modeling Language, — это унифицированный язык моделирования. Его используют, чтобы создавать диаграммы и схемы для визуализации процессов и явлений. UML нужен, чтобы превращать абстракции в визуальные модели и схемы. Поэтому применять его можно во многих областях:

- в программировании — чтобы наглядно видеть связи между классами и другими частями приложения или чтобы построить карту поведения пользователя на сайте;
- в дизайне — чтобы создавать интерфейсы и понимать, как пользователи будут взаимодействовать с ними;
- в бизнесе — чтобы визуально представлять, как работают бизнес-процессы или ведётся документооборот в организации.

Отображение архитектуры с использованием пакетов UML.

Пакеты – это способ группировки любых элементов языка UML – диаграмм, вариантов использования, классов и пр. Обычно пакеты представляют собой: подсистемы, если речь идет о системе, компоненты или слои, если речь идет о классах, наборы бизнес-логики, если речь идет о вариантах использования и т.п. Конкретный способ группировки элементов в пакеты остается на усмотрение проектировщика. Пакеты могут включать в себя не только объекты, но и диаграммы различных типов. Т.е. в каждом пакете могут быть свои диаграммы вариантов использования, последовательностей, классов и пр.

Диаграмма пакетов – средство группирования элементов модели. Диаграмма пакетов – единственное средство, позволяющее управлять сложностью самой модели. Основные элементы нотации – пакеты 1 и зависимости с различными стереотипами 2, применяемые на диаграмме, показаны на следующем рисунке.



В основном пакеты служат для объединения классов в группы, и именно этот способ их применения здесь описывается, но помните, что пакеты могут применяться для любой другой конструкции языка UML.

Компоненты. Диаграмма компонентов.

Компонент скрывает свою реализацию за набором внешних интерфейсов. Компоненты системы, разделяющие общие интерфейсы, могут замещать друг друга, сохраняя при этом одинаковое логическое поведение. Компонент (component) – это модульный фрагмент логического представления системы, взаимодействие с которым описывается набором обеспеченных и требуемых интерфейсов. Затраты на тиражирование программ любой величины и сложности исчезающе малы по сравнению с затратами на их разработку. Отсюда возникает естественное желание не создавать каждую новую программу "с нуля", а использовать ранее созданные программы в качестве готовых строительных элементов, благо затраты на копирование столь не велики. Понятие компонента в UML не соответствует тому понятию, которое имеет в виду сборочное программирование. Компонент UML является частью модели, и описывает логическую сущность, которая существует только во время проектирования (design time), хотя в дальнейшем ее можно связать с физической реализацией (артефактом) времени исполнения (run time).

Диаграмма компонентов (component diagram) – показывает взаимосвязи между модулями (логическими или физическими), из которых состоит моделируемая система

Основной тип сущностей на диаграмме компонентов – это 1. сами компоненты, 2. интерфейсы, посредством которых указывается взаимосвязь между компонентами. На диаграмме компонентов применяются следующие отношения:

- реализации между компонентами и интерфейсами (компонент реализует интерфейс);
- 3. зависимости между компонентами и интерфейсами (компонент использует интерфейс)

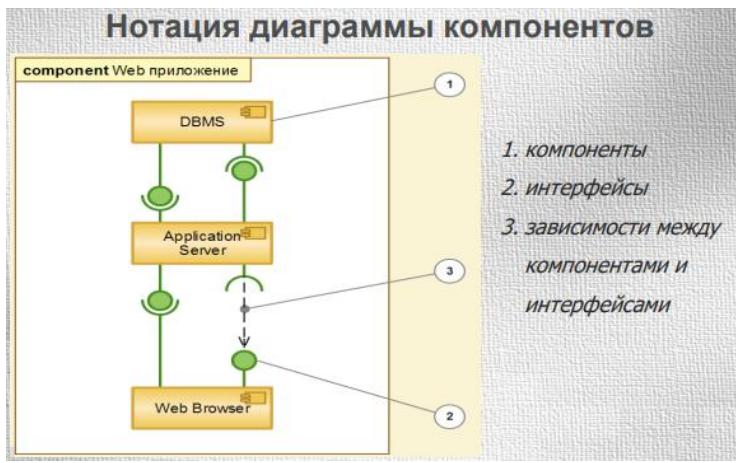


Диаграмма размещения.

Каждая система имеет требования к безопасности, производительности, доступности и другим возможностям. Диаграмма развертывания помогает удовлетворить эти требования. В большинстве случаев диаграмма развертывания единственная, отображающая сетевые аспекты вашего решения. Диаграмма развертывания должна ясно и понятно показывать, как запросы проходят через систему.

Диаграммы размещения применяются только при моделировании многокомпонентных приложений.

Проектирование элементов системы (детальное проектирование).

Детальное проектирование — это техническая деятельность, которая следует за выбором архитектуры. Основной целью этой деятельности является как можно более полная подготовка проекта к его реализации. Другими словами, программисты должны иметь возможность реализовать детальный проект, сконцентрировавшись только на проблемах, связанных с созданием программного кода.

Этапы детального проектирования

- **Проектирование подсистем**

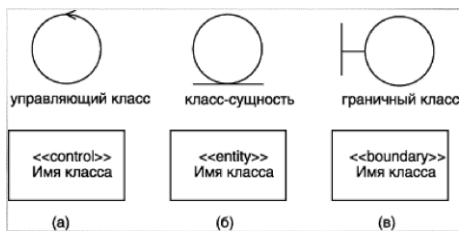
Подсистемы можно независимо заказывать, настраивать или доставлять; разрабатывать, пока интерфейсы остаются без изменений; развертывать в наборе распределенных вычислительных узлов; изменять без ущерба для других частей системы. Таким образом, подсистемы идеальны для моделирования компонентов - сменных блоков, предназначенных для сборки во время разработки на базе компонентов. Подсистема предоставляет поведение через один или несколько интерфейсов, которые она реализует.

После уточнения реализаций вариантов использования осуществляется проектирование подсистем. Оно включает в себя следующие действия: 1. Распределение поведения подсистемы по ее элементам документирование взаимодействия элементов подсистемы в виде коопераций («реализаций интерфейса»); 2. построение одной или более диаграмм взаимодействия для каждой операции интерфейса подсистемы, 3. Документирование элементов подсистемы (в виде диаграмм классов), 4. Описание зависимостей между подсистемами.

- **Проектирование классов**

Стереотипы классов – это механизм, позволяющий разделять классы на категории. В языке UML определены три основных стереотипа классов:

- Границный класс (Boundary). Границные классы - классы, которые расположены на границе системы и всей окружающей среды. Это экранные формы, отчеты, интерфейсы с аппаратурой и интерфейсы с другими системами. Каждому взаимодействию между действующим лицом и вариантом использования должен соответствовать, по крайней мере, один граничный класс. Именно такой класс позволяет действующему лицу взаимодействовать с системой.
- Класс-сущность (Entity). Классы-сущности содержат хранимую информацию. Класс-сущность содержит информацию, которая должна храниться постоянно и не уничтожается с уничтожением объектов данного класса или прекращением работы моделируемой системы, связанные с выключением системы или завершением программы. Обычно для каждого класса-сущности создают таблицу в базе данных.
- Управляющий класс (Control). Управляющие классы отвечают за координацию действий других классов. Обычно у каждого варианта использования имеется один управляющий класс, контролирующий последовательность событий этого варианта использования. В дальнейшем они могут объединяться, разделяться или исключаться.



• Проектирование базы данных

Детальное проектирование:

- Создание логической модели данных: На этом этапе переходят от высокоуровневой модели к более детальной, учитывая ограничения и особенности конкретной базы данных (например, использование определенных СУБД).
- Нормализация данных: Позволяет устраниТЬ избыточность данных и связать их таким образом, чтобы избежать аномалий при обновлении, вставке или удалении данных.
- Физическое проектирование: Разработка структуры базы данных, выбор конкретных типов данных, создание таблиц, определение индексов и ключей.

Процесс детального проектирования базы данных требует тщательного планирования и учета различных аспектов, таких как производительность, безопасность данных, расширяемость и эффективность запросов. Кроме того, важно учитывать нормализацию данных для избежания избыточности и аномалий.

Использование инструментов моделирования баз данных, таких как ER-диаграммы (сущность-связь), UML-диаграммы и специализированные программы для проектирования баз данных, может значительно упростить этот процесс, помогая визуализировать и анализировать структуру данных.

• Проектирование интерфейса

Проектирование интерфейса включает в себя создание пользовательского опыта, который удовлетворяет потребности пользователей и обеспечивает удобство использования системы. Детальное проектирование интерфейса включает несколько ключевых этапов:

1. Исследование пользовательских потребностей: Проведение анализа пользовательских потребностей и понимание их целей и задач. Это может включать опросы, интервью, наблюдение за пользователями и анализ сценариев использования.
2. Создание концепции интерфейса: Разработка общей концепции пользовательского интерфейса, включая компоненты, функции, навигацию и структуру. Это может включать создание эскизов, макетов или прототипов интерфейса.
3. Интерактивное проектирование: Создание прототипов с использованием специализированных инструментов для интерактивного проектирования, которые позволяют пользователям протестировать и взаимодействовать с интерфейсом. Это помогает выявить проблемы удобства использования и внести необходимые изменения.
4. Дизайн пользовательского интерфейса: Определение внешнего вида интерфейса, включая цветовую схему, шрифты, стили, элементы управления и макет. Важно обеспечить консистентность и удобство восприятия информации.
5. Тестирование и итерации: Проведение тестирования интерфейса на предмет удобства использования и эффективности. Получение обратной связи от пользователей и внесение необходимых изменений для улучшения интерфейса.
6. Документация дизайна: Создание документации, описывающей элементы интерфейса, их функции и поведение. Это помогает обеспечить консистентность при разработке и поддержке интерфейса.

При детальном проектировании интерфейса важно учитывать контекст использования, потребности и предпочтения пользователей, а также современные тенденции в дизайне интерфейсов.

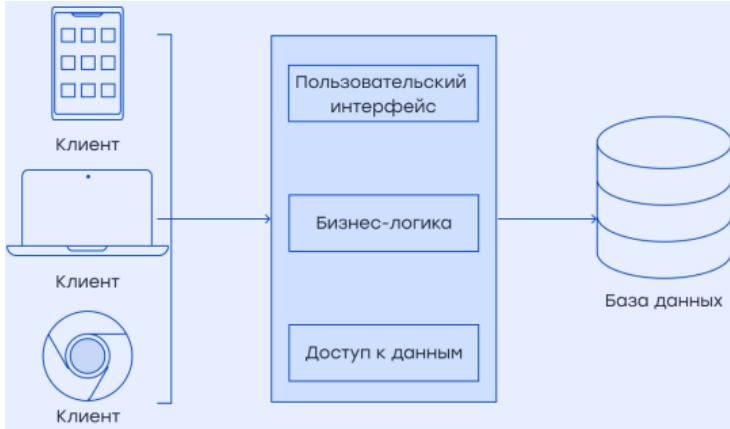
8.3 Микросервисная архитектура

- Микросервисы против Монолитов. Проблемы монолитной архитектуры
- Что такое микросервисы
- Различия между монолитной архитектурой и микросервисами
- Архитектура микросервисов
- Преимущества и недостатки микросервисов

Микросервисы - это архитектурный стиль, который структурирует приложение как набор небольших автономных сервисов, смоделированных на основе микросервисной архитектуры, или микросервисов. В микросервисной архитектуре каждая служба является автономной и реализует единую бизнес-функцию. Разработчики используют эту технологию для масштабирования своих приложений в соответствии со своими потребностями.

Микросервисы против Монолитов. Проблемы монолитной архитектуры

И микросервисы, и монолитные сервисы – это архитектурные паттерны, которые используются при разработке программных приложений для обслуживания бизнес-требований. У каждого из них есть свои преимущества, недостатки и проблемы. Монолитные архитектуры могут быть громоздкими в работе, когда нужно добавить новые функции, внести изменения или даже удалить некоторые ненужные функции.



Монолитные архитектуры по-прежнему являются хорошим вариантом для приведенных ниже случаев: малогабаритные приложения; нет необходимости в обновлении технологии; требуется меньше времени для вывода на рынок; команды знакомы с Монолитными подходами. С другой стороны, архитектура микросервисов не очень хорошо подходит для небольших приложений, требующих единого технологического стека, платформы деплоя, доменной зоны и т. д. Но для крупномасштабных приложений пока нет ничего лучше этой архитектуры.

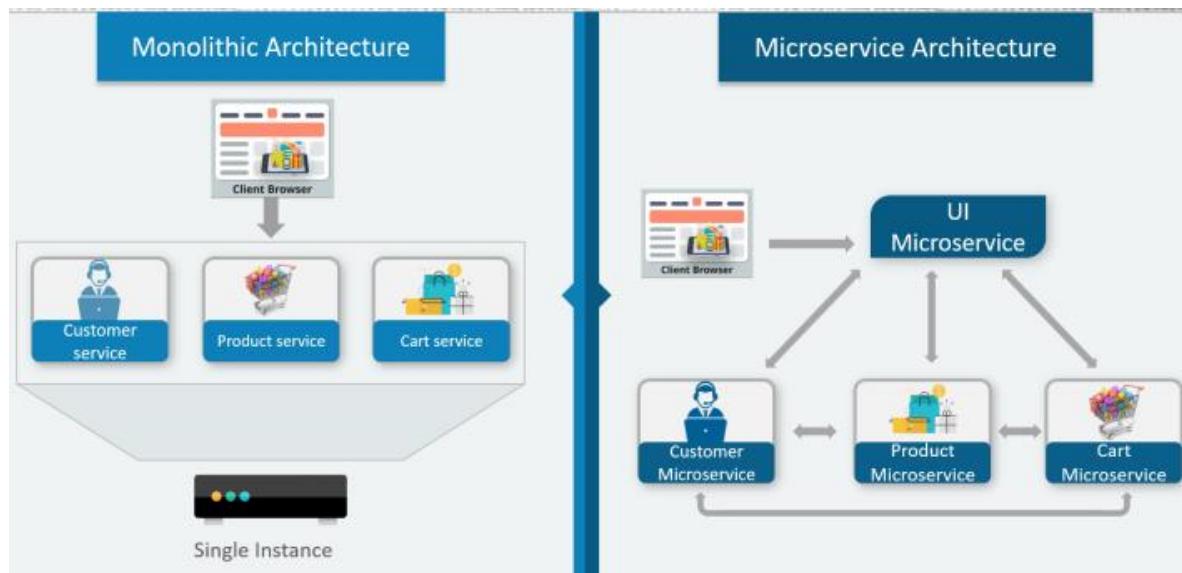
Проблемы монолитной архитектуры:

- Негибкий – монолитные приложения не могут быть построены с использованием различных технологий
- Ненадежный – даже если одна особенность системы не работает, то вся система не работает
- Немасштабируемый – приложения не могут быть легко масштабированы, так как каждый раз, когда приложение должно быть обновлено, вся система должна быть перестроена.
- Блоки непрерывного развития – многие функции приложений не могут быть созданы и развернуты одновременно.
- Медленное развитие – разработка в монолитных приложениях занимает много времени, так как каждая функция должна быть построена одна за другой.
- Не подходит для сложных применений – особенности сложных приложений имеют тесно связанные зависимости.

Что такое микросервисы

Микросервисы — это архитектурный стиль, который структурирует приложение как набор небольших автономных служб, смоделированных вокруг бизнес-доменов. В данной архитектуре каждая служба является автономной и реализует единую бизнес - возможность. Каждая команда agile создает индивидуальный компонент сервиса с доступными фреймворками и выбранным стеком технологий. Каждый компонент сервиса формирует надежную архитектуру микросервиса для обеспечения лучшей масштабируемости. Кроме того, проблемы с каждым компонентом сервиса могут решаться индивидуально командой agile без какого-либо или минимального воздействия на все приложение.

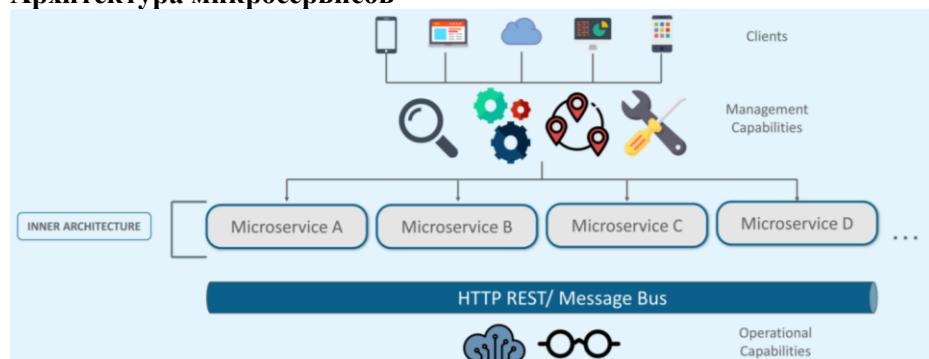
Различия между монолитной архитектурой и микросервисами



Основное различие, которое можно наблюдать на приведенной выше диаграмме, заключается в том, что все функции изначально находились под одним экземпляром, совместно использующим одну базу данных. Но затем, с микрослужбами, каждой функции была выделена отдельная микрослужба, обрабатывающая свои собственные данные и выполняющая различные функции.

Характеристика	Микросервисы	Монолит
Скорость	Сравнительно медленнее в понятиях разработки, поскольку мелкие сервисы разрабатываются отдельно.	Быстрее, когда речь идет о разработке одного приложения, содержащего почти все необходимое с самого начала.
Независимость	Каждый сервис имеет собственное хранилище данных и отвечает за конкретный домен, и может разрабатываться, изменяться или деплоиться независимо.	Благодаря тесному взаимодействию все приложение деплоится при любом изменении, поскольку вся бизнес-логика находится в одном месте.
Гибкость	Сервисы имеют меньший размер, что делает их легко адаптируемыми к новым технологическим изменениям.	Используется одна и та же технология, которую нельзя изменить, поскольку все приложение было разработано с ее использованием.
Масштабирование	Легко масштабировать сервисы без ущерба для других.	Приходится масштабировать все приложение, даже если нужно масштабировать только одну фичу внутри.
Надежность	Нет необходимости останавливать приложение, если требуется внедрить изменения в сервис.	Приходится заново деплоить все приложение, если что-то требует изменений.
Тестируемость	Сервисы, специфичные для конкретного домена, легко тестировать.	Сложно протестировать всю систему сразу.
Удобство обслуживания	Отдельные сервисы могут быть обновлены и изменены в соответствии с рыночными условиями.	Трудно поддерживать при увеличении размера приложения, поскольку изменения в одном месте могут иметь регрессивные последствия для других.
Команды	Можно задействовать несколько небольших команд с различными наборами технологических скиллов.	Требуется одна большая команда с почти одинаковыми навыками и пониманием технологий.
Управление	Управление различными небольшими командами, каждая из которых работает над определенной областью бизнеса, позволяет легко анализировать, кто за что отвечает.	Управление одной большой командой, работающей над полным доменом, требует внутреннего разделения.
Коммуникация	Коммуникация медленнее, но легче.	Есть сложность общения с руководством по поводу возникающих проблем.
Инфраструктура	Экспоненциальная стоимость, поскольку разные команды с разными навыками работают с несколькими репозиториями, средами и пайплайнами.	Экономическая эффективность, поскольку команды работают над одним и тем же репозиторием, и все команды обладают практически одинаковыми навыками в области технологий. Таким образом, нет необходимости в нескольких различных средах разработки и деплоя.

Архитектура микросервисов



Клиенты – разные пользователи с разных устройств отправляют запросы.

Поставщики удостоверений (Identity Providers) – проверяет подлинность удостоверений пользователей или клиентов и выдает токены безопасности.

API Gateway – Обрабатывает запросы клиентов.

Статическое содержимое (Static Content) – содержит все содержимое системы.

Управление (Management) – балансирует службы на узлах и выявляет сбои.

Обнаружение служб (Service Discovery) – руководство по поиску маршрута связи между микросервисами.

Сети доставки контента (Content Delivery Networks) – распределенная сеть прокси-серверов и их центров обработки данных.

Удаленное обслуживание (Remote Service) – позволяет получить удаленный доступ к информации, которая находится в сети ИТ-устройств.

Разные клиенты с разных устройств пытаются использовать разные сервисы, такие как поиск, сборка, настройка и другие возможности управления. Все службы разделены в зависимости от их доменов и функциональных возможностей и далее распределяются по отдельным микрослужбам. Эти микрослужбы имеют свой собственный балансировщик нагрузки и среду выполнения для выполнения своих функций и в то же время захватывают данные в своих собственных базах данных.

Все микрослужбы взаимодействуют друг с другом через сервер без отслеживания состояния, который является либо REST, либо Message Bus. Микросервисы знают свой путь связи с помощью Service Discovery и выполняют операционные возможности, такие как автоматизация, мониторинг. Затем все функциональные возможности, выполняемые микрослужбами, передаются клиентам через шлюз API. Все внутренние точки подключаются из шлюза API. Таким образом, любой, кто подключается к шлюзу API, автоматически подключается ко всей системе.

Преимущества и недостатки микросервисов

Преимущества:

1. Независимая разработка — все микросервисы могут быть легко разработаны на основе их индивидуальной функциональности.

2. Независимое развертывание — на основе их услуг они могут быть индивидуально развернуты в любом приложении.

3. Изоляция неисправностей — даже если одна служба приложения не работает, система все равно продолжает функционировать.

4. Смешанный стек технологий — различные языки и технологии могут использоваться для создания различных сервисов одного и того же приложения.

5. Гранулярное масштабирование — отдельные компоненты могут масштабироваться в соответствии с потребностями, нет необходимости масштабировать все компоненты вместе.

6. Изменение в сервисе не повлияет на всю систему, что обеспечивает высокую надежность.

7. Тестирование каждой службы является простым и менее трудоемким, поскольку службы специфичны для своего хранилища данных.

Недостатки:

1. Инфраструктура требует больших затрат, поскольку различные технологии имеют различные методологии тестирования, среды для каждого сервиса и деплоя.

2. Дебаг может стать громоздким из-за различных логов каждого сервиса и выявления ошибок в реальном сервисе.

3. Полиглот микросервисов имеет и побочные эффекты, когда компании необходимо поддерживать ресурсы для каждой технологии, используемой для разработки сервисов.

8.4 Принципы объектно-ориентированного проектирования

- Принцип единственной обязанности
- Принцип открытости/закрытости
- Принцип подстановки Барбары Лисков
- Принцип разделения интерфейсов
- Принцип инверсии зависимостей

Роберт Мартин составил список, состоящий всего из пяти правил хорошего проектирования, которые известны, как принципы SOLID.

1. SRP: Single Responsibility Principle (принцип единственной обязанности)

2. OCP: Open/Closed Principle (принцип открытости/закрытости)

3. LSP: Liskov Substitution Principle (принцип замещения Лисков)

4. ISP: Interface Segregation Principle (принцип изоляции интерфейса)

5. DIP: Dependency Inversion Principle (принцип обращения зависимости)

Принцип единственной обязанности

Если класс отвечает за несколько операций сразу, вероятность возникновения багов возрастает – внося изменения, касающиеся одной из операций вы, сами того не подозревая, можете затронуть и другие. Назначение:

Принцип служит для разделения типов поведения, благодаря которому ошибки, вызванные модификациями в одном поведении, не распространялись на прочие, не связанные с ним типы.

Принцип открытости/закрытости

Классы должны быть открыты для расширения, но закрыты для модификации. Когда меняете текущее поведение класса, эти изменения сказываются на всех системах, работающих с данным классом. Если хотите, чтобы класс выполнял больше операций, то идеальный вариант – не заменять старые на новые, а добавлять новые к уже существующим. Назначение: Принцип служит для того, чтобы делать поведение класса более разнообразным, не вмешиваясь в текущие операции, которые он выполняет. Благодаря этому избегаете ошибок в тех фрагментах кода, где задействован этот класс.

Принцип подстановки Барбары Лисков

Должна быть возможность вместо базового типа подставить любой его подтип. Если тип Т имеет подтип П, то любые объекты типа Т, присутствующие в программе, могут заменяться объектами типа П без негативных последствий для функциональности программы. В случаях, когда класс-потомок не способен выполнять те же действия, что и класс-родитель, возникает риск появления ошибок. Класс-потомок должен производить такие же операции, как и класс-родитель. Это называется наследственностью. Назначение: Принцип служит для того, чтобы обеспечить постоянство: класс-родитель и класс-потомок могут использоваться одинаковым образом без нарушения работы программы.

Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом. Если сформулировать принцип подстановки Барбары Лисков упрощенно: подклассы не должны противоречить надклассам. Речь идет о том, как организовать код, чтобы, если потребуется дополнить или изменить какую-либо часть, не нужно было переписывать все с самого начала.

Принцип разделения интерфейсов

Клиенты не должны вынужденно зависеть от методов, которыми не пользуются. Интерфейсы принадлежат клиентам, а не иерархиям. Когда классу приходится производить действия, не несущие никакой реальной пользы, это выливается в пустую трату ресурса, а в случае, если класс выполнять эти действия не способен, ведёт к возникновению багов. Класс должен производить только те операции, которые необходимы для осуществления его функций. Все другие действия следует либо удалить совсем, либо переместить, если есть вероятность, что они понадобятся другому классу в будущем. Назначение: Принцип служит для того, чтобы раздробить единый набор действий на ряд наборов поменьше – таким образом, каждый класс делает то, что от него действительно требуется, и ничего больше.

Принцип инверсии зависимостей

Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те, и другие должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций. Модули (или классы) верхнего уровня = классы, которые выполняют операцию при помощи инструмента. Модули (или классы) нижнего уровня = инструменты, которые нужны для выполнения операций. Абстракции – представляют интерфейс, соединяющий два класса. Детали = специфические характеристики работы инструмента.

Согласно данному принципу, класс не должен соединяться с инструментом, который применяется для выполнения операции. Вместо этого он должен быть соединён с интерфейсом, который поможет установить связь между инструментом и классом. Кроме того, принцип гласит, что ни интерфейс, ни класс, не обязаны вникать в специфику работы инструмента. Напротив, это инструмент должен подходить под требования интерфейса. Назначение: Этот принцип служит для того, чтобы устранить зависимость классов верхнего уровня от классов нижнего уровня за счёт введения интерфейсов.