

# Технологии программирования (второй семестр)

Обзор курса

## Технологии программирования

- Программное обеспечение := компьютерные программы и соответствующая документация.
- Программирование := процесс создания программного обеспечения.
- Программное обеспечение := {любительское, промышленное:= {«коробочное» ПО, сделанное на заказ}}.
- Промышленное ПО := заказчик + команда (фирма разработчик) + бюджет (деньги) + сроки.
- Современное ПО СЛОЖНО:
  - Сложность реального мира. Взаимоотношения между заказчиком, пользователями и разработчиками.
  - Трудность управления процессами разработки. Разработка – коллективный процесс. Программирование – процесс творческий.
  - Гибкость ПО. В отличие от других областей возможно все написать «с нуля», не всегда есть готовые «кирпичи-компоненты».
  - Проблемы описания поведения больших дискретных систем. (Задачи, которые ставятся для решения программному обеспечению сложны).
- **Необходимы технологии по созданию ПО := совокупность процессов, ведущих к созданию или развитию ПО.**
- **Технологий разработано много, но все они содержат следующие базовые процессы:**

## Базовые процессы разработки ПО (существуют в любой технологии)

- **Разработка спецификации ПО:** = определяет все функции и действия, которые будет выполнять разрабатываемое ПО.
- **Проектирование** : = на основе спецификации разработка архитектуры системы.
- **Реализация** := создание ПО (кодирование, написание документации и т.д.).
- **Аттестация:** = верификация и аттестация. Разработанное ПО должно быть аттестовано на соответствие заказчику.
- **Эволюция ПО:** = дальнейшая модификация ПО в соответствии с требованиями заказчика.

Технологии отличаются друг от друга последовательностью и порядком применения и проведения базовых процессов. Для систематизации технологий выделяются модели. Модель : = абстрактное представление процесса

***PS:Знать наизусть даже на два.***

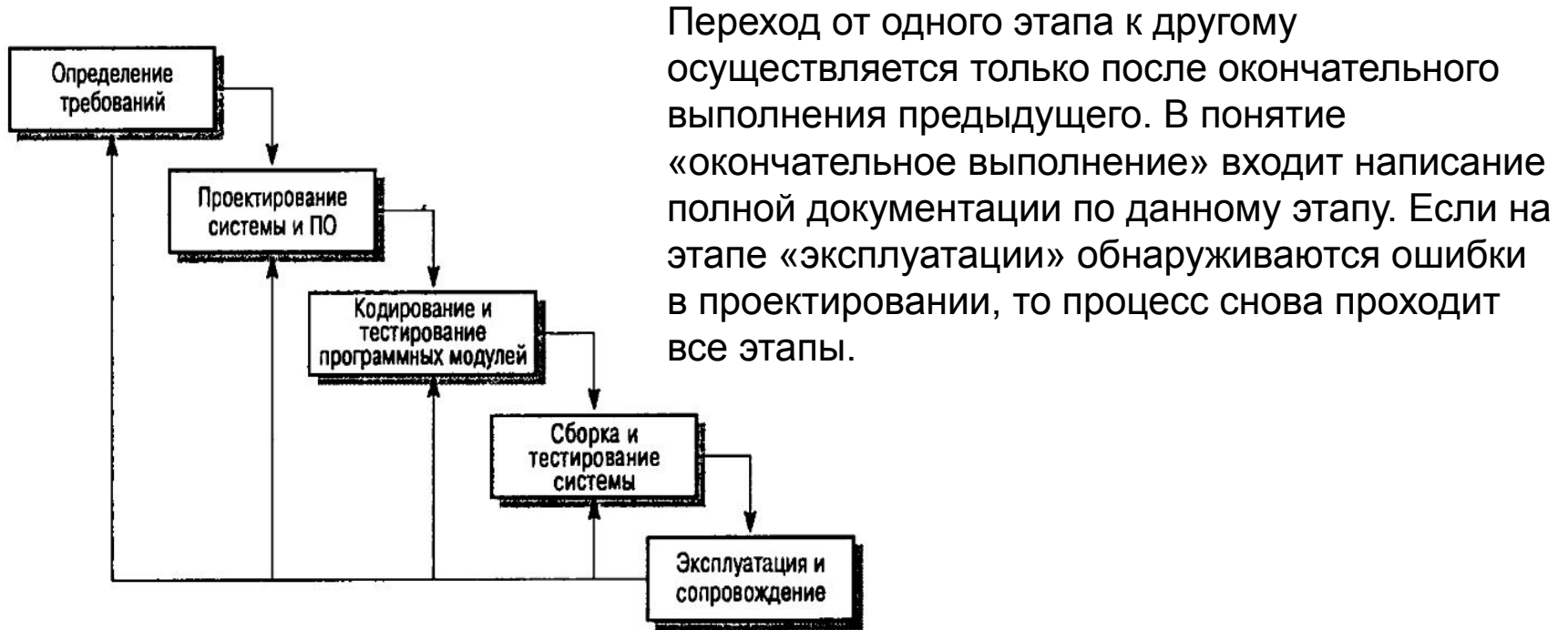
## Модели процесса создания ПО (жизненного цикла)

- Каскадная (водопадная модель)
- Эволюционная модель: = {подход пробных разработок, прототипирование}
- Разработка на основе ранее созданных компонент
- Спиральная модель

Жизненный цикл : = это совокупность процессов, протекающих в период от момента принятия решения о создании ПО до его полного выхода из строя. Жизненный цикл шире чем модели создания ПО.

Модели показывают возможные подходы к разработке ПО, на практике могут применяться любые «комбинации» этих подходов. Например, какой-то этап в спиральной модели может быть осуществлен с применением каскадной модели.

## Каскадная модель



- Не гибкое разделение процесса создания на этапы
- Определяющее значение имеют решения принятые на ранних этапах. Если возникают ошибки на ранних этапах, то приходится повторять все с начала.
- Классический подход требует параллельной документации каждого процесса.
- + Хорошо отражает и структурирует процесс создания ПО
- + Используется при проектировании систем, где системные требования четко определены заранее.

## Эволюционная модель разработки

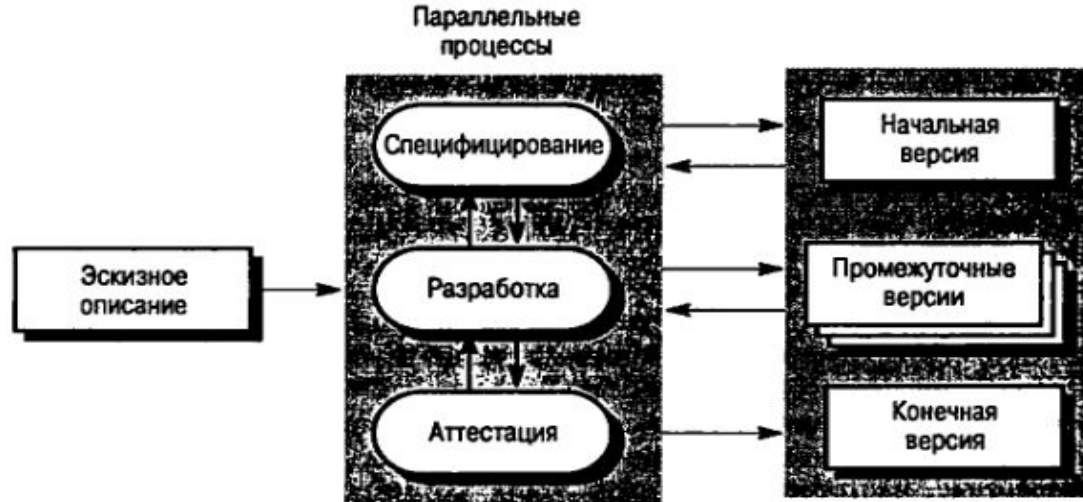
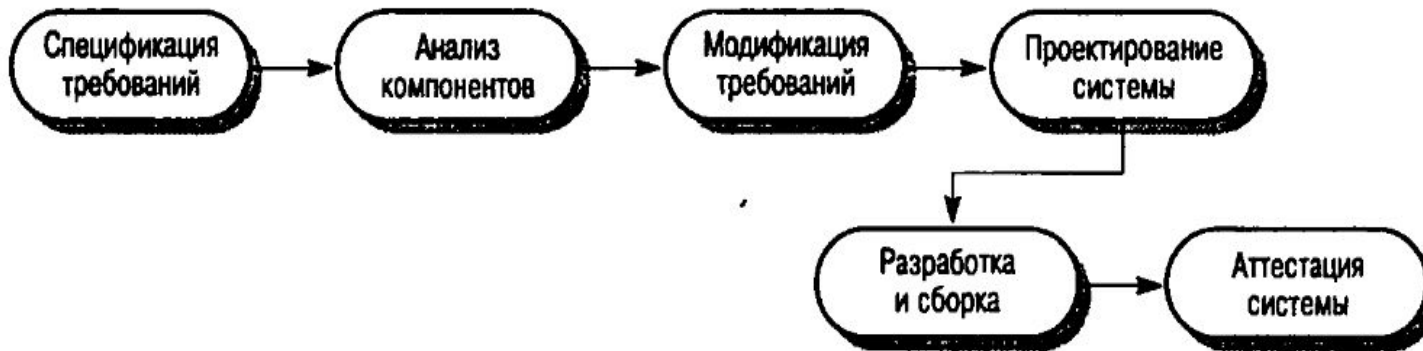


Рис. 3.2. Эволюционная модель разработки

Идея – разрабатывается первоначальная версия ПО, которая передается на испытание пользователям, затем она дорабатывается с учетом мнение пользователей, получается промежуточная версия и т.д. Пока окончательная версия не будет удовлетворять пользователя. Процессы специфицирование, разработки, аттестации выполняются параллельно при постоянном взаимном обмене информацией.

- Многие этапы не документированы.
- Система получается плохо структурированной (иногда).
- Часто требуются специальные средства и технологии разработки.
- + Более эффективен, чем каскадная модель если требования заказчика могут меняться в процессе разработки.
- + Спецификация разрабатывается постепенно, по мере того как заказчик осознает и формулирует задачи, которое должно решать ПО.

## Разработка на основе ранее созданных компонент



*Рис. 3.5. Разработка ПО с повторным использованием ранее созданных компонентов*

!!! После анализа функциональности компонентов возможна модификация требований. Так как не все требования заказчика в полном объеме могут быть реализованы существующими компонентами.

+ Процесс построения системы заключается в компоновке готовых компонентов.

+ Уменьшается срок создания ПО.

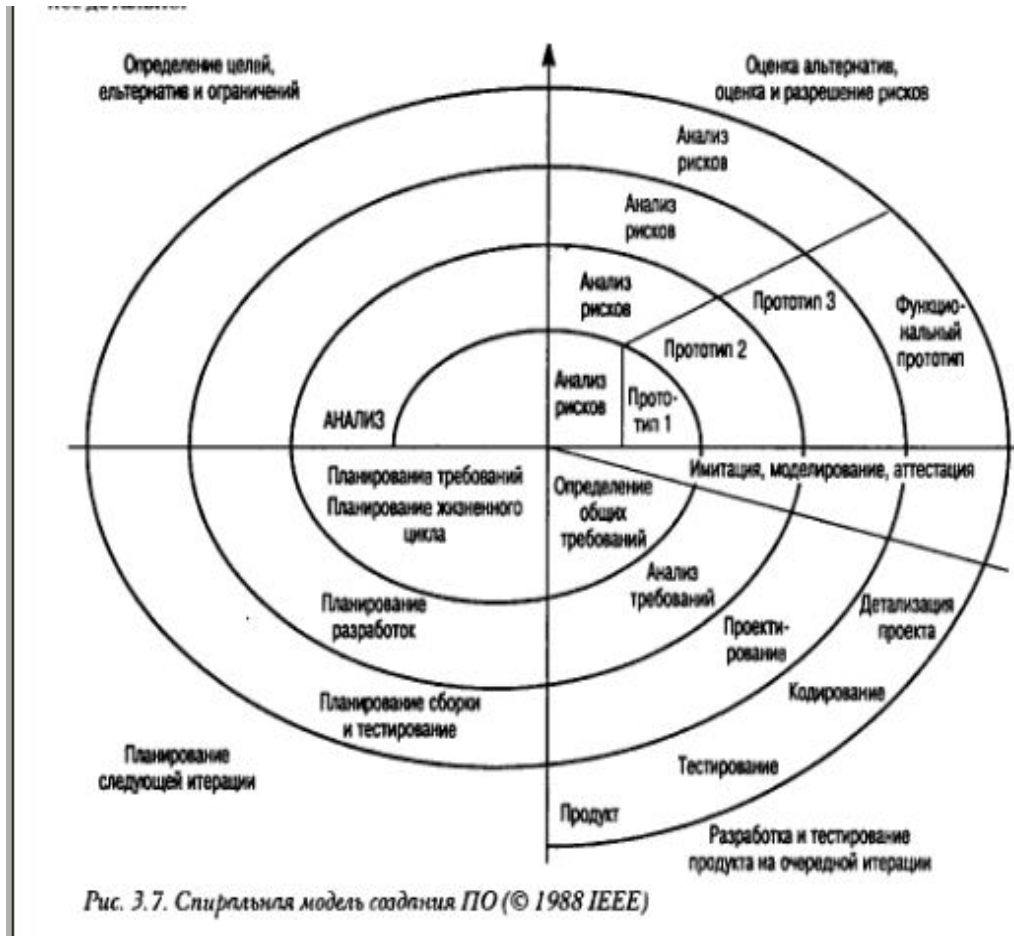
+ Уменьшается стоимость ПО из-за использования готовых частей.

+ Требуется тестировать только сборку, не требуется тестировать готовые компоненты.

-На данный момент «рынок» компонентов окончательно не сформировался. Не понятно, где брать и искать эти готовые компоненты.

-Не вся функциональность может быть реализована на основании готовых компонентов.

# Спиральная модель



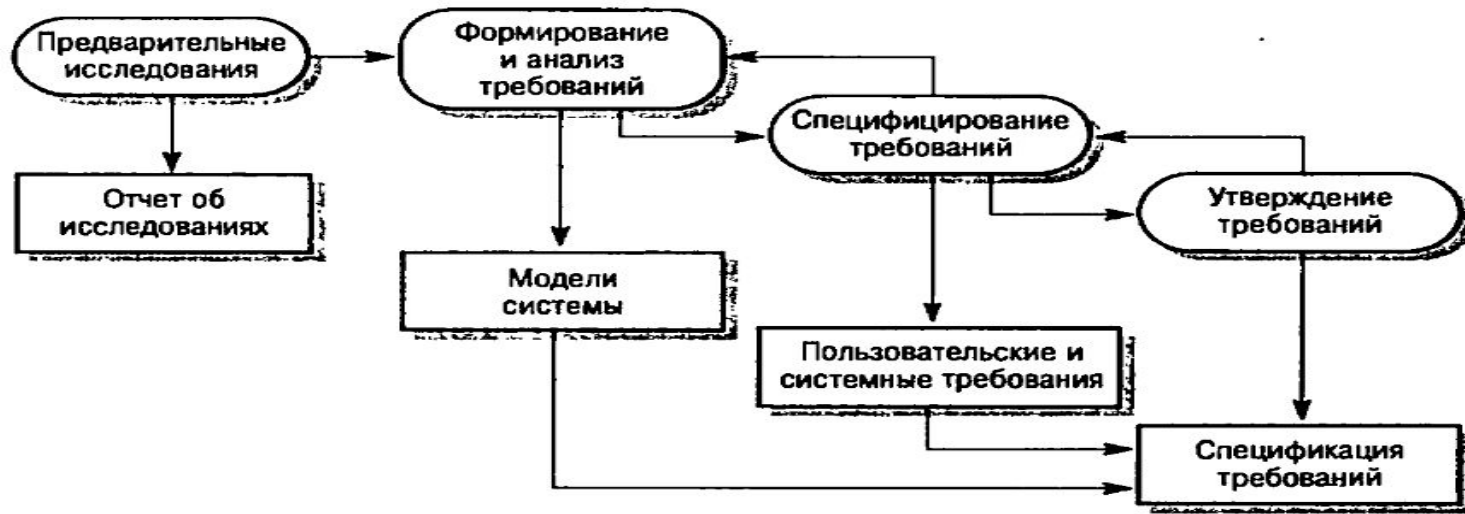
Каждый виток спирали:

1. Определение целей
2. Оценка и разрешение рисков
3. Разработка и тестирование
4. Планирование.

Нет четких этапов как в других моделях. Эта модель может включать в себя любые другие модели.



## Разработка спецификации ПО



- **Предварительное исследование** - оценивается степень удовлетворенности пользователей существующем ПО, бюджетные ограничения на разработку нового ПО, его экономическая эффективность.
- **Формирование и анализ требований** – изучение существующих аналогичных систем, обсуждение будущего ПО с заказчиками и пользователями, анализ задач и т.д. Может включать разработку нескольких моделей системы и ее прототипов, что помогает сформировать функциональные требования к системе.
- **Специфицирование требований** – перевод всей информации в «бумажный» документ.
- **Утверждение требований** – утверждение сформированных требований заказчиком.

## Требования к программному обеспечению.

- **Пользовательские требования** – описание на естественном языке ( плюс диаграммы) функций, выполняемых системой, и ограничений, накладываемых на систему.
- **Системные требования** – детализированное описание системных функций и ограничений. Основа для заключения контракта между покупателями и заказчиком.
- **Проектная системная спецификация** – дополняет и детализирует системные требования. Основа для более детализированного проектирования системы. (Обычно составляется для разработчиков системы)

**Таблица 5.1. Пользовательские и системные требования**

---

### **Пользовательские требования**

---

1. ПО должно предоставить средство доступа к внешним файлам, созданным в других программах.
- 

### **Спецификация системных требований**

---

- 1.1. Пользователь должен иметь возможность определять тип внешних файлов.
  - 1.2. Для каждого типа внешнего файла должно иметься соответствующее средство, применимое к этому типу файлов.
  - 1.3. Внешний файл каждого типа должен быть представлен соответствующей пиктограммой на дисплее пользователя.
  - 1.4. Пользователю должна быть предоставлена возможность самому определять пиктограмму для каждого типа внешних файлов.
  - 1.5. При выборе пользователем пиктограммы, представляющей внешний файл, к этому файлу должно быть применено средство, ассоциированное с внешними файлами данного типа.
- 

Пример пользовательских и системных требований.

## Требования к программному обеспечению

- **Функциональные требования** – перечень сервисов, которые должна выполнять система, должно быть указано, как система реагирует на входные данные, как ведет в определенных ситуациях и т.д.
- **Нефункциональные требования** – описывает характеристики системы и ее окружения, а не поведение системы. Перечень ограничений, накладываемых на действия, выполняемые системой.
  - **Требования к продукту** – эксплуатационные свойства программного продукта (требования к производительности системы, объемы необходимой памяти, надежности, удобства эксплуатации и т.д.)
  - **Организационные требования** – отображают политику и организационные процедуры заказчика и разработчика ПО (стандарты разработки ПО, сроки изготовления, требования к документации и т.д.)
  - **Внешние требования** – учитываются факторы, внешние по отношению к ПО и его разработке, например, взаимодействие системы с другими системами, этические правила и т.д.
- **Требования к предметной области** – характеризует ту предметную область, где будет эксплуатироваться система. Делятся на функциональные и нефункциональные.

## Способы записи системных требований

Спецификации системных требований часто пишутся на естественных языках. Но применение естественных языков подразумевает, что те, кто пишет спецификацию и те, кто ее читают, одни и те же слова и выражения понимают одинаково. Поэтому существуют следующие методы записи системных требований:

Структурированный естественный язык	Сокращенная форма естественного языка, предназначенная для написания спецификаций. Стандартные формы и шаблоны.
Языки описания программ	Использование специальных структурированных языков. Например, псевдокода или PDL (program description language) и т.д.
Графические нотации	Графический язык, используемый для описания функциональных требований диаграммы и блок-схемы. Например, стандарты IDEF0,3 и UML.
Математические спецификации.	Система нотаций, основанная на математических концепциях. Формализованная однозначная, лишенная двусмысленности запись системных требований. Но она понятна далеко не всем заказчикам.

## Структурированный естественный язык.

### Пример спецификации системного требования:

#### **Врезка 5.8. Спецификация системного требования, использующая стандартную форму**

##### **Эклипс/APM/Средства/DE/RD/3.5.1**

**Функция.** Добавление структурных элементов в схему.

**Описание.** Добавление структурных элементов в существующую схему системной архитектуры. Пользователь выбирает тип структурного элемента и его местоположение. После вставки в схему структурный элемент становится выделенным (текущим структурным элементом). Пользователь определяет местоположение элемента путем перемещения курсора по области схемы.

**Входные данные.** Тип элемента, позиция элемента, идентификатор схемы.

**Источники входных данных.** Тип элемента и позиция элемента задаются пользователем, идентификатор схемы получен из базы данных проекта.

**Выходные данные.** Идентификатор схемы.

**Пункт назначения.** База данных проекта. Идентификатор схемы помещается в базу данных проекта по завершении выполнения данной функции.

**Для выполнения функции требуется схема, определенная входным идентификатором схемы.**

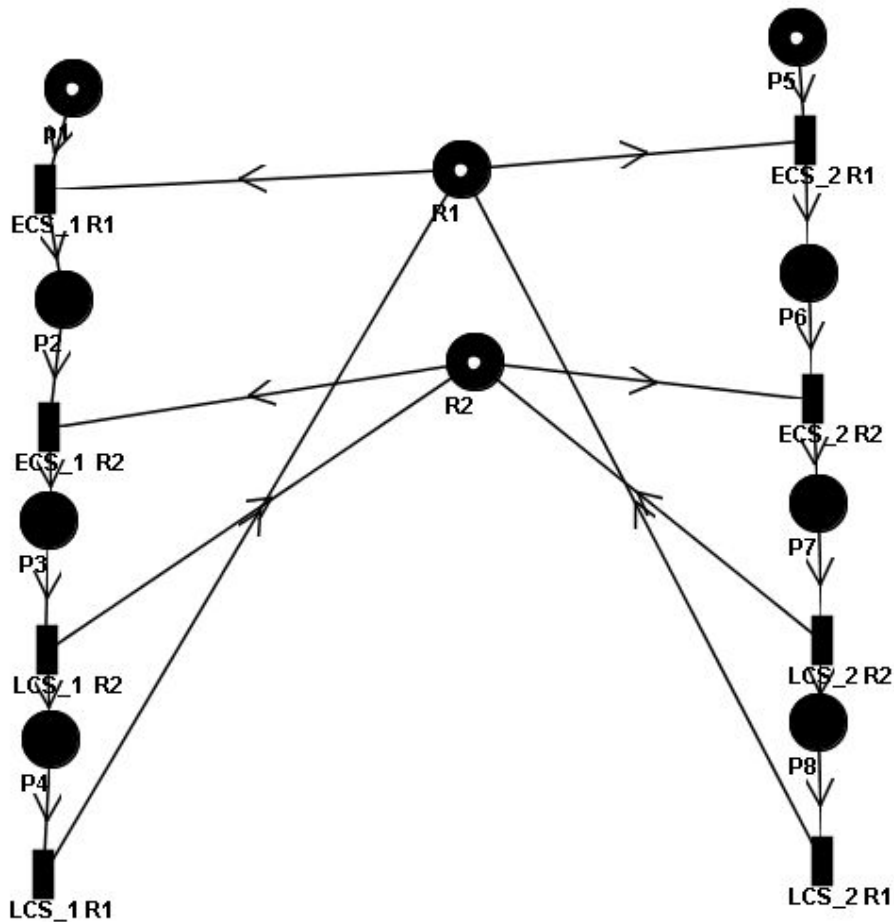
**Предусловие.** Схема открыта и отображается на экране пользователя.

**Постусловие.** Схема, за исключением вставки нового структурного элемента, не изменяется.

**Побочные эффекты.** Нет.

**Спецификация:** Эклипс/APM/Средства/DE/RD/3.5.1

# Математическая спецификация. Формальные спецификации для параллельных систем. Сети Петри.



**Потоки использующие два разделяемых ресурса.**

```

CriticalSection R1,R2;
void ThreadExecute1()
{ EnterCriticalSection &R1;
  EnterCriticalSection &R2;
  Выполнить действия с R1 и R2
  LeaveCriticalSection &R2
  LeaveCriticalSection &R1}
Void ThreadExecute2()
{EnterCriticalSection
 &R1;EnterCriticalSection &R2;
  Выполнить действия с R1 и
 R2LeaveCriticalSection
 &R2LeaveCriticalSection &R1}

Int Main() {
  Запустить поток 1
  Запустить поток 2
}
    
```

Рис.1 Сеть Петри, специфицирующая данный код

## Математическая спецификация. Продолжение.

- Сети Петри могут представляться с помощью графов, как показано на предыдущем примере, а могут иметь формальную запись, например, в виде протоколов флаговых и пусковых функций.
- Для формально записанной спецификации можно проводить анализ соответствующими методами. Спецификация записанная формальным языком может быть проанализирована (проверена, исследована) автоматически. Этим и интересна формальная спецификация.
- Таблица флаговых и пусковых функций для сети Петри рис.1

$$\Psi_t(\text{ECS\_1 R1}) = R1 * P1 * \neg P2$$

$$\Phi_{t+1}(\text{ECS\_1 R1}): R1=0; P1=0; P2=1$$

$$\Psi_t(\text{ECS\_1 R2}) = R2 * P2 * \neg P3$$

$$\Phi_{t+1}(\text{ECS\_1 R2}): R1=0; P1=0; P2=1$$

(аналогично можно построить и для всех остальных переходов)

$\Psi$  – пусковая функция, задает правило срабатывания перехода

$\Phi$  - флаговая функция, задает состояние флагов после срабатывания перехода

**Иногда флаговые и пусковые функции называют одним термином: асинхронный протокол.**

# Модели систем

- Распространенная **методика документирования системных требований** является **построение ряда моделей системы**. Модели используют графическое представления, показывающие как решение задачи, для которой создается система, так и разрабатываемой системы. Модели являются связующим звеном между процессом анализа и проектированием системы.
- Модели показывают систему в различных аспектах:
  - Внешнее представление, когда моделируется окружение или рабочая среда системы.
  - Описание поведения системы, когда моделируется ее поведение
  - Описание структуры системы, когда моделируется архитектура системы или структуры данных, обрабатываемые системой.
- Типы системных моделей, которые могут создаваться в процессе анализа систем
  - Модель обработки данных. Последовательность обработки данных в системе.
  - Композиционные модели (диаграммы «сущность-связь»). Как одни сущности связаны с другими.
  - Архитектурная модель. Основные подсистемы из которых строится система.
  - Классификационная модель. (диаграмма классов, показывает какие объекты имеют общие характеристики).
  - Модель «стимул-ответ» . Диаграммы изменения состояний показывают, как система реагирует на внутренние и внешние события.



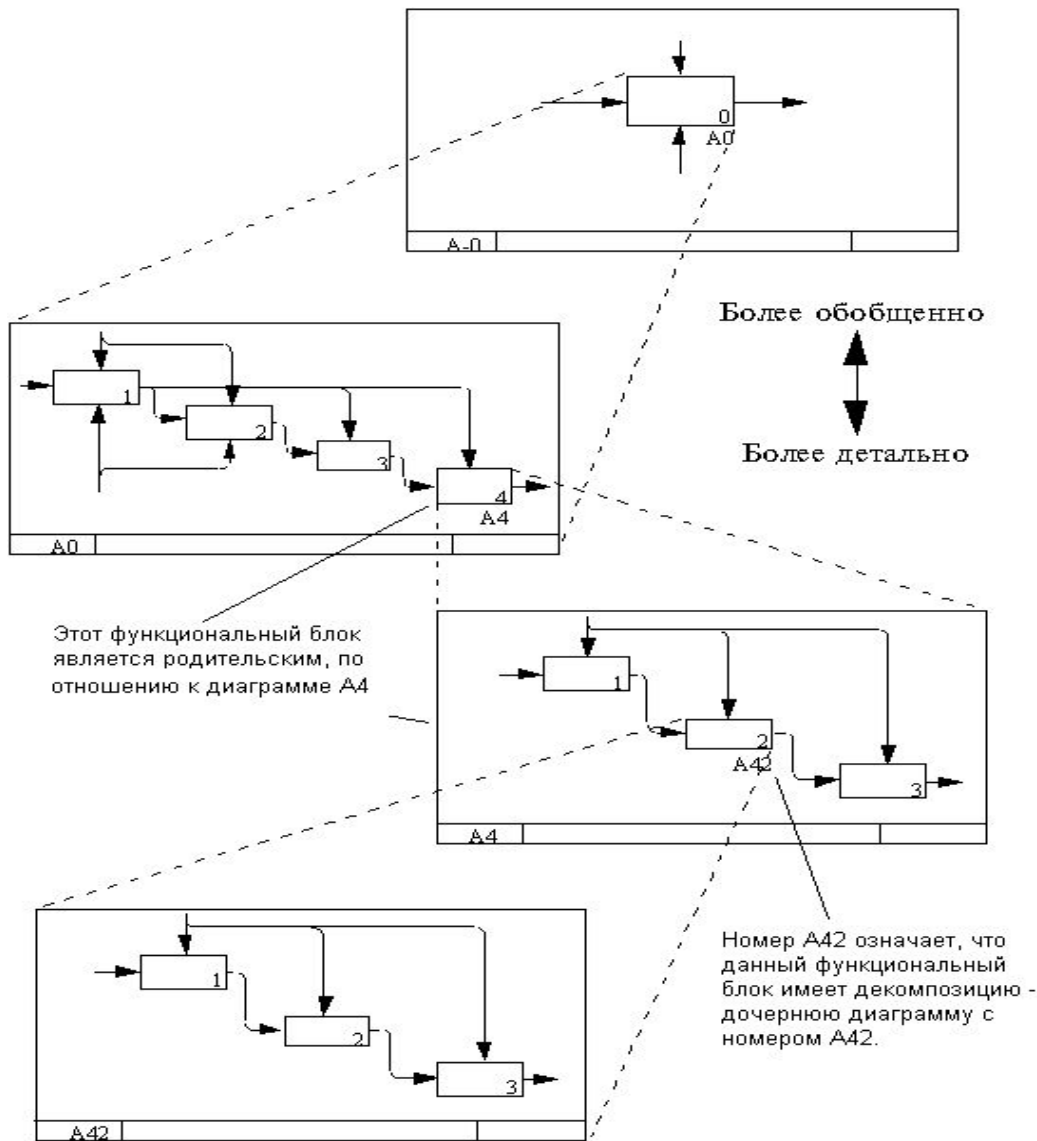
## Процесс анализа (неформальное введение).

- Процесс анализа - необходим на этапе разработки системных требований, чтобы понять какие задачи должно решать создаваемое ПО и правильно построить модель системы.
- **Строить модель и документировать** можно **как угодно**, в любых понятных терминах и изображениях. Но чтобы вас **понимали другие** лучше использовать **готовые языки (методы) построения моделей**. Почти всегда язык определяет также и технологию моделирования.
- Структурный анализ - проводится с целью исследования статических характеристик системы путем выделения в ней подсистем и элементов различного уровня и определения отношений и связей между ними.
  - Описание функциональной структуры системы (метод SADT)
  - Последовательность выполняемых действий (метод IDEF3)
  - Передача информации между функциональными процессами (метод DFD)
  - Отношения между данными (метод ER. Модель «сущность-связь»).
- Объектно-ориентированный анализ (ООА)– методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области. (UML).

## SADT – Structured Analysis and Design Technique.

- IDEF0 (Icam Definition)- методология функционального моделирования. С помощью наглядного графического языка IDEF0, изучаемая система предстает перед разработчиками и аналитиками в виде набора взаимосвязанных функций (функциональных блоков - в терминах IDEF0). Как правило, моделирование средствами IDEF0 является первым этапом изучения любой системы.
- **Основные понятия IDEF0:**
  - **Activity Box – функциональный блок.** Каждая из четырех сторон функционального блока имеет своё определенное значение (роль), при этом:
    - Верхняя сторона имеет значение “Управление” (Control);
    - Левая сторона имеет значение “Вход” (Input);
    - Правая сторона имеет значение “Выход” (Output);
    - Нижняя сторона имеет значение “Механизм” (Mechanism).
  - **Arrow – интерфейсная дуга.** Отображает элемент системы, который обрабатывается функциональным блоком или оказывает иное влияние на функцию, отображенную данным функциональным блоком.
  - **Декомпозиция** - разбиение процесса на составляющие функции. Позволяет представлять модель системы в виде иерархической структуры отдельных диаграмм, что делает ее менее перегруженной и легко усваиваемой. Модель IDEF0 всегда начинается с представления системы как единого целого – одного функционального блока с интерфейсными дугами, простирающимися за пределы рассматриваемой области.
  - **Глоссарий** Для каждого из элементов IDEF0: диаграмм, функциональных блоков, интерфейсных дуг существующий стандарт подразумевает создание и поддержание набора соответствующих определений, ключевых слов, повествовательных изложений и т.д., которые характеризуют объект, отображенный данным элементом.

# Пример декомпозиции структурных диаграмм. На примере IDEF0.

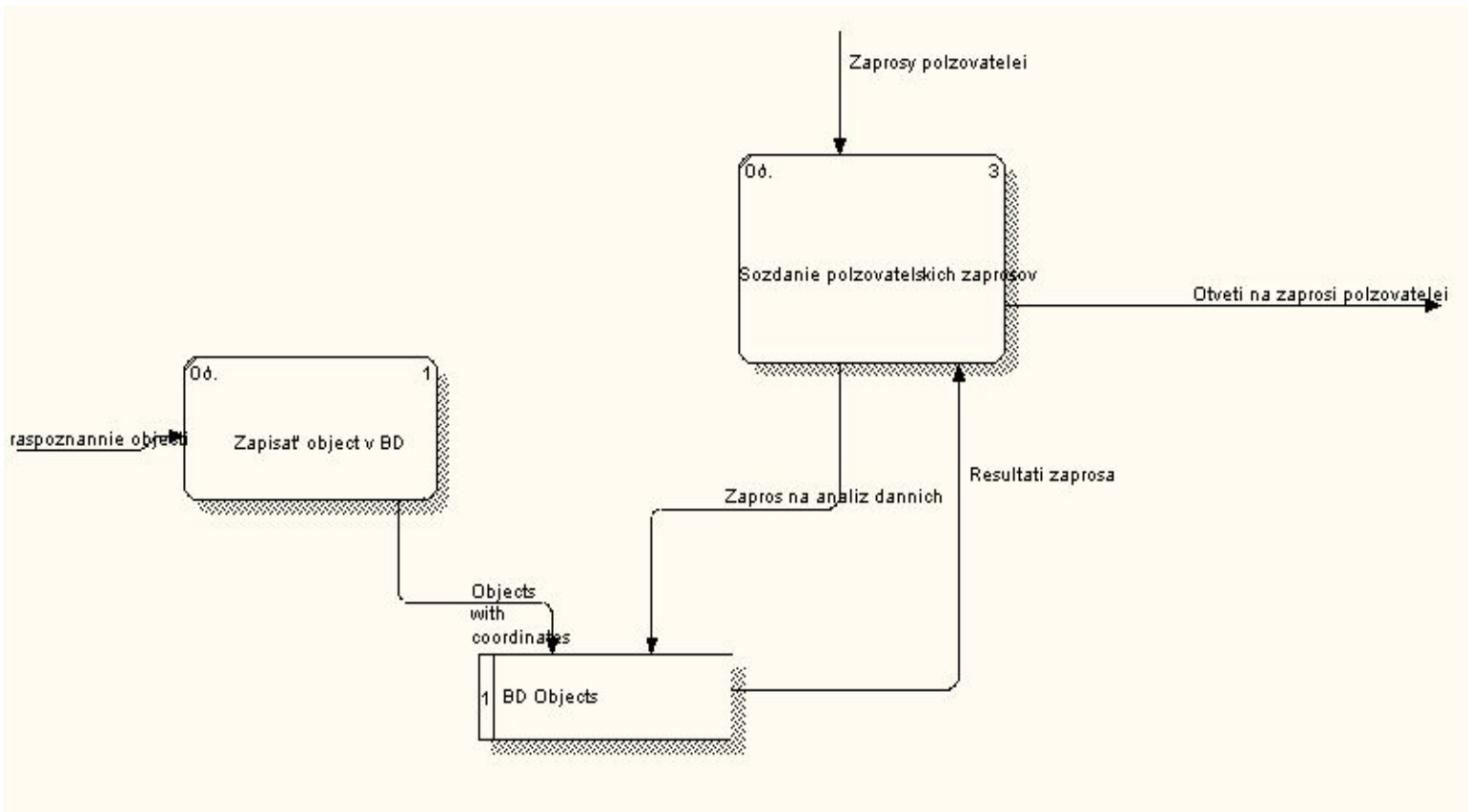


# IDEF3

- IDEF3 является стандартом документирования технологических процессов, происходящих на предприятии, и предоставляет инструментарий для наглядного исследования и моделирования их сценариев. **Сценарием** (Scenario) называется описание последовательности изменений свойств объекта, в рамках рассматриваемого процесса.
- Два типа диаграмм:
  - **PFDD** (Process Flow Description Diagrams) Диаграмма Описания Последовательности Этапов Процесса.
    - **Функциональный блок** (UOB – Unit of Behavior)
    - **Перекрестки (Junction)**. Используются для отображения логики взаимодействия стрелок (потоков) при слиянии и разветвлении или для отображения множества событий, которые могут или должны быть завершены перед началом следующей работы
      - Перекрестки слияния (Fan-in junction)
      - Перекрестки разветвления (Fan-out junction)
      - 5 типов перекрестков ( AND (синхр. и асинхр.), OR (синхр. и асинхр.), XOR)
  - **OSTN** (Object State Transition Network). Диаграмма Состояния Объекта и его Трансформаций в процессе. Состояния объекта отображаются окружностями, а их изменения направленными линиями. Каждая линия имеет ссылку на соответствующий функциональный блок UOB, в результате которого произошло отображаемое ей изменение состояния объекта

# Data Flow Diagram (DFD)

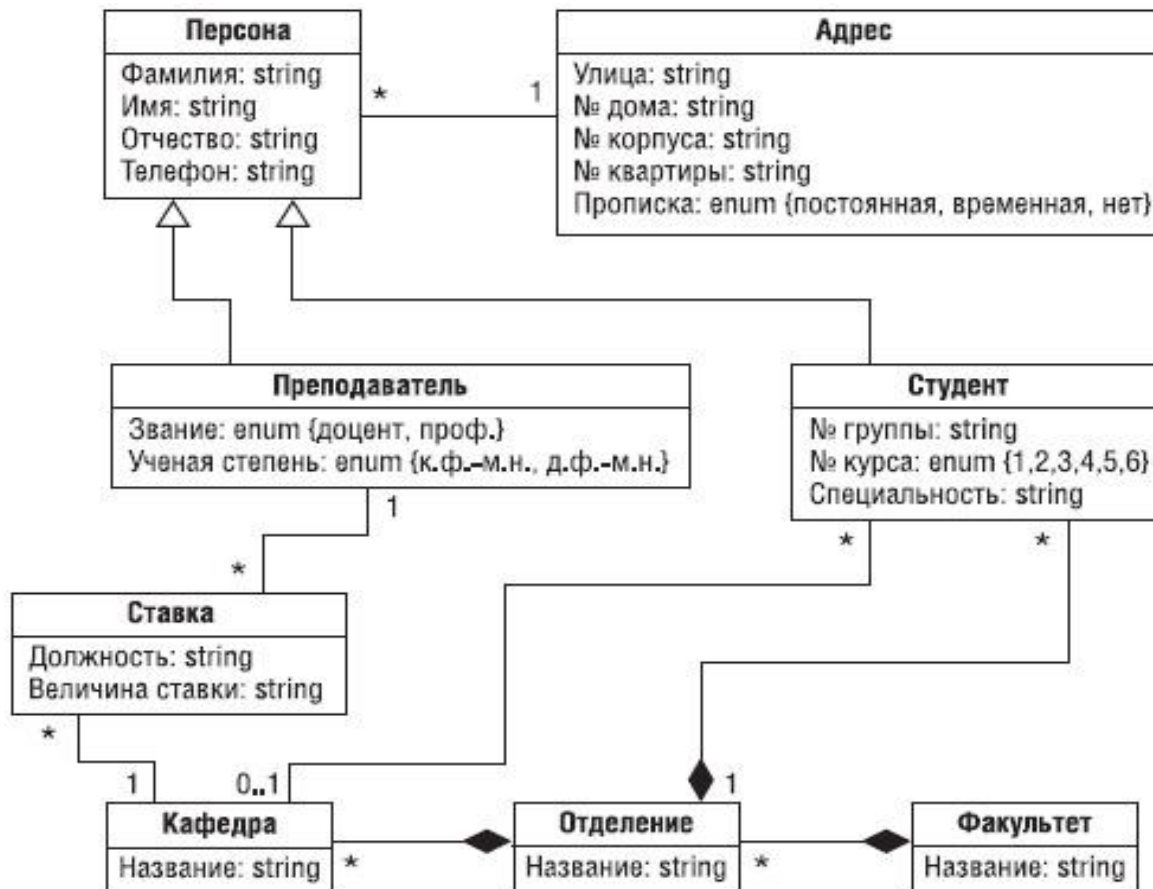
- Диаграммы потоков данных представляют собой иерархию функциональных процессов, связанных потоками данных. Цель такого представления - продемонстрировать, как каждый процесс преобразует свои входные данные в выходные, а также выявить отношения между этими процессами. Модель системы определяется как иерархия потоков данных, описывающих процесс преобразования информации от входа в систему до выдачи пользователю.



## ER – диаграммы. IDEF1X.

- Метод ER (Entity Relationship) «сущность-связь» основан на выделении в предметной области «сущностей» и отражении взаимосвязи между этими сущностями.
  - **Сущность** (entity) - это "предмет" рассматриваемой предметной области, который может быть идентифицирован некоторым способом, отличающим его от других "предметов". Конкретные человек, компания или событие являются примерами сущности.
  - **Связь** (relationship) - это некоторое отношение между двумя и более сущностями, отражающее то, как они участвуют в общей деятельности, взаимодействуют друг с другом, совместно используются некоторой другой сущностью и т. д.
- Стандарт IDEF1 – моделирование предметной области с помощью ER-диаграмм.
- IDEF1X – метод проектирования реляционных баз данных.(ERWin), основанный на ER-диаграммах.
- CASE средства (например, ERWin) позволяют спроектировать реляционную структуру базы данных в виде диаграмм. При этом поддерживается два уровня представлений: логический и физический. На основании диаграмм генерируется скрипт (SQL/DDDL). Существует Forward Engineering – генерация скрипта на основании модели, Reverse Engineering – обратная генерация диаграммы на основании базы данных.

## Пример диаграммы IDEF1X. Логическая модель.



# Введение в UML

- **UML** – Unified Modeling Language – это стандартный инструмент для разработки «чертежей» программного обеспечения. Используется для визуализации, спецификации, конструирования и документирования программных систем. Особо эффективен при объектно-ориентированном подходе к разработке ПО.
- **Иерархия диаграмм UML :**
- **Структурные (structural) модели (статические модели):**
  - диаграммы классов (class diagrams) - для моделирования статической структуры классов системы и связей между ними; набор классов, интерфейсов, связи между классами.
  - диаграммы компонентов (component diagrams) - для моделирования иерархии компонентов (подсистем) системы; структура компонентов.
  - диаграммы размещения (deployment diagrams) - для моделирования физической архитектуры системы.
- **Срезы текущего состояния системы:**
  - диаграммы объектов(object) – набор объектов и их связи. Статические копии состояний экземпляров системы.
- **Модели поведения (behavioral):**
  - диаграммы вариантов использования (use case diagrams) - для моделирования функциональных требований к системе (в виде сценариев взаимодействия пользователей с системой);
  - диаграммы взаимодействия (interaction diagrams):
    - диаграммы последовательности (sequence diagrams) - для моделирования процесса обмена сообщениями между объектами;
  - диаграммы состояний (statechart diagrams) - для моделирования поведения объектов системы при переходе из одного состояния в другое;
    - диаграммы деятельности (activity diagrams) - для моделирования поведения системы в рамках различных вариантов использования, или потоков управления.



# Проектирование программного обеспечения



Рис. 3.9. Обобщенная схема процесса проектирования

- **Архитектурное проектирование** — определяются и документируются подсистемы и взаимосвязи между ними.
- **Обобщенная спецификация** — для каждой подсистемы разрабатывается обобщенная спецификация на ее сервисы и ограничения.
- **Проектирование интерфейсов** — для каждой подсистемы определяется и документируется интерфейс.
- **Компонентное проектирование** — распределение системных функций по различным компонентам и их интерфейсам.
- **Проектирование структур данных** — разрабатываются структуры данных, необходимые для реализации программной системы.
- **Проектирование алгоритмов** — детально разрабатываются алгоритмы для реализации системных сервисов.

# Архитектурное проектирование

Этапы общие, для всех процессов архитектурного проектирования:

1. Структурирование системы – программная система структурируется в виде совокупности относительно независимых подсистем. Определяется взаимодействие между подсистемами.
  1. Модель хранилища (репозитария)
  2. Модель клиент-сервер
  3. Модель абстрактной машины.
2. Моделирование управления – разрабатывается базовая модель управления взаимоотношениями между частями.
  1. Централизованное управление (Одна из подсистем полностью отвечает за управление)
    - Модель «вызов-возврат».
    - Модель диспетчера.
  2. Управление, основанное на событиях.
    - Модели передачи сообщений.
    - Модели, управляемые прерываниями.
3. Модульная декомпозиция – каждая определенная на первом этапе подсистема разбивается на отдельные модули. Определяются типы модулей и типы их взаимодействий.
  1. Объектно-ориентированная модель.
  2. Поток данных
  3. Функциональная (алгоритмическая) декомпозиция.

## Подсистема и модуль.

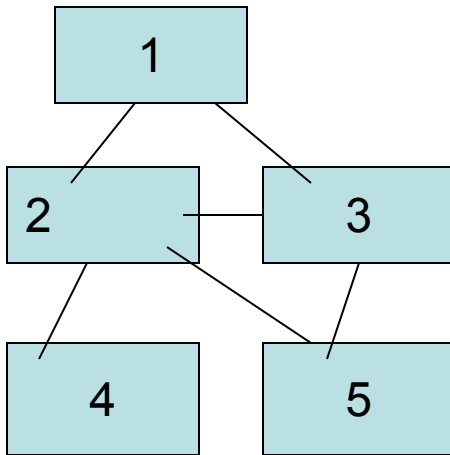
- Подсистема - это система, операции (методы) которой не зависят от сервисов, предоставляемых другими подсистемами. Подсистемы состоят из модулей и имеют определенные интерфейсы, с помощью которых взаимодействуют с другими системами.
- Модуль – компонент системы, который предоставляет сервисы другим модулям, может использовать сервисы других модулей. Не является как правило независимой системой. В терминах языков программирования – модуль это отдельный файл, содержащий программный код. В каждом языке свои правила написания модулей. В С++ модуль - \*.h, \*.cpp файлы, в Delphi - \*.pas и т.д.
- **Свойства модуля**
  - Размер модуля. Рекомендуемый (100-1000 операторов)
  - Прочность модуля – мера его внутренних связей. Чем выше прочность, тем больше связей он может спрятать. (Логическая, временная, процедурная, коммутативная, информационная, функциональная). Рекомендуемые: Информационная – действия над одной структурой данных, функциональная – реализуют какую либо одну функциональность.
  - Сцепление модуля – мера зависимости по данным. (по содержимому, по общей области, по внешним ссылкам, по управлению, по образцу, параметрическое – единственное рекомендованное).
  - Рутинность модуля – его независимость от предыстории обращений к нему.

## Методы проектирования программного обеспечения.

- Структурный подход - алгоритмическая (функциональная) декомпозиция. **ОТВЕТ НА ВОПРОС: «КАК РАБОТАЕТ СИСТЕМА?»**
  - Восходящая разработка («снизу вверх») (Не рекомендуется к использованию, но почему-то почти все студенты пишут свои программы именно так....).
    - Классический подход
    - Архитектурный поход
  - Нисходящая разработка («сверху вниз»)
    - Классический подход.
    - Конструктивный подход
- Потоки данных (Data Flow). Программная система рассматривается как преобразователь входных данных в выходные.
  - Граф-диаграммы
  - Диаграммы Варнье-Орра
  - Функциональные диаграммы
  - ПЕРТ - диаграммы
- Объектно-ориентированные методы проектирования (ООД). Программа рассматривается как совокупность объектов, взаимодействующих между собой. **Объектная декомпозиция. ОТВЕТ НА ВОПРОС: «КТО РАБОТАЕТ В СИСТЕМЕ?»**
- Использование паттернов. Паттерны – готовые образцы решения конкретных типовых задач.

## Структурный подход. Древоподобная структура модулей.

- Принцип «разделяй и властвуй». Декомпозиция по функциям (алгоритмам).
- Обычно модульную структуру программы представляют в виде древоподобной структуры. В узлах такого дерева размещаются программные модули, а направленные дуги (стрелки) показывают статическую подчиненность модулей, т.е. каждая дуга показывает, что в тексте модуля, из которого она исходит, имеется ссылка на модуль, в который она входит.



Дерево представляет из себя граф, где вершинами являются модули, а ребрами – статическую подчиненность, если в модуле есть вызов другого модуля. Граф линейно упорядочен по уровням. Нижний уровень (листья дерева) соответствуют самым нижним модулям. !!!Деревья могут быть со сросшимися ветвями.

1. Под модулем в этой структуре может пониматься и подпрограмма и конкретная процедура.
2. В UML древоподобная структура модулей представляется в виде диаграммы артефактов.

## Восходящее и нисходящее программирование.

- Классический подход. В начале разрабатывается дерево модулей.
- Восходящая разработка - программируются модули, начиная с самого нижнего модуля, верхний модуль программируется последним.
- Нисходящая разработка – программируются модули, начиная с самого верхнего модуля. Нижние модули программируются последними.
- Конструктивный подход (нисходящая разработка) - разработка программы представляет собой модификацию нисходящей разработки, при которой модульная древовидная структура программы формируется в процессе программирования модуля.
- Архитектурный подход (восходящая разработка) -Архитектурный подход к разработке программы представляет собой модификацию восходящей разработки, при которой модульная структура программы формируется в процессе программирования модуля. Но при этом ставится существенно другая цель разработки: повышение уровня используемого языка программирования, а не разработка конкретной программы. Это означает, что для заданной предметной области выделяются типичные функции, каждая из которых может использоваться при решении разных задач в этой области, и специфицируются, а затем и программируются отдельные программные модули, выполняющие эти функции.

# Объектно-ориентированное программирование (ООП)

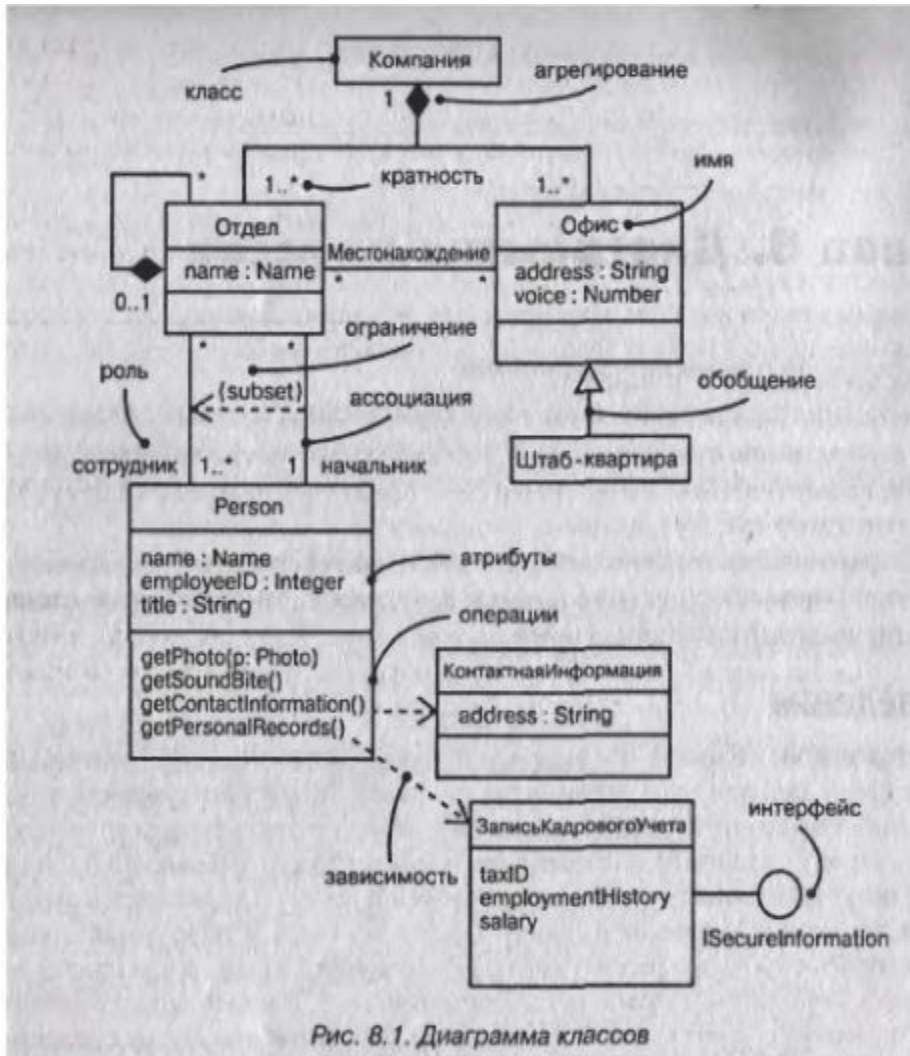
- **ООП (ООР)** – это методология программирования, основанная на представлении программы в виде **совокупности объектов**, каждый из которых является экземпляром определенного **класса**, а классы образуют **иерархию** объектов.
- **Концептуальная база объектно-ориентированного стиля программирования:**
  - Абстракция (абстрагирование) – выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его границы с точки зрения наблюдателя.
  - Инкапсуляция – процесс разделения устройства и поведения объекта. Служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации. Пользователь видит только интерфейсную часть объекта и не вникает в его внутреннюю реализацию.
  - Модульность – состояния системы, разложенной на внутренне связанные (связано то, что внутри модуля), но слабо связанные между собой модули. Позволяет хранить абстракции отдельно.
  - Иерархия – это упорядочивание абстракций, расположение их по уровням.
  - Типизация – способ защититься от использования объектов одного класса, вместо другого, или по крайней мере, управлять такой подменой. (Сильная и слабая типизация. Статическое и динамическое связывание. Полиморфизм – один и тот же код выполняется по разному в зависимости от того, объект какого класса используется при вызове данного кода).
  - Параллелизм – отличает активные объекты (содержат отдельный поток управления) от пассивных.
  - Сохраняемость – способность объекта существовать во времени и (или) пространстве.

## Классы и объекты

- **Объект** – осязаемая реальность, проявляющая четко выделяемое поведение. Объект обладает состоянием, поведением и идентичностью. Состояние – перечень всех свойств объекта и текущими значениями каждого из свойств. Поведение – как объект действует и реагирует.
- **Класс** – это некое множество объектов, имеющих общую структуру и общее поведение.
- **Отношения между классами:**
  - Ассоциация – смысловая связь между классами. (Пример товары и продажи)
  - Наследование – возможность порождать один класс от другого с сохранением всех свойств и методов класса – предка. Отношение «is a» «общего и частного».  
Одиночное наследование и множественное наследование (Проблемы: конфликт имен между суперклассами ( $A\{int\ i\}$ ,  $B\{char\ i\}$ ,  $C:A, B\{i-??\}$ ), повторное наследование ( $B : A, C : A, D : B, C ??$ )).
  - Агрегация – физическое включение. Отношение «целое/часть» (part of). Часто ассоциация превращается в простую агрегацию.
  - Использование – один класс пользуется услугами другого. (Есть ссылка в вызовах функций, непосредственное использование в теле функций и т.д.).
  - Инстанцирование – параметризованные классы.
  - Метакласс – это класс, экземпляры которого есть классы.



## Диаграмма классов. Нотация UML.



- Связь наследования (обобщения) От потомка к родителю)
- Связь агрегации (содержит, есть ссылка в атрибутах одного класса на другой).
- Связь зависимости (Один класс использует другой класс в своих процедурах).
- Современные CASE-средства (Rational Rose или STAR UML(бесплатная программа) и т.д.) поддерживают процесс генерации исходного кода по диаграмме классов (генерируется структура класса, иерархия, шаблоны функций). Разделяется Forward Engineering - процесс генерации кода на основании диаграммы, и процесс Reverse Engineering – процесс построения диаграмм на основании кода программы.

## Объектно-ориентированные языки. Сравнительная таблица.

		SmallTalk	C++
Абстракции	<ul style="list-style-type: none"> <li>•Переменные экземпляра</li> <li>•Методы экземпляра</li> <li>•Переменные класса</li> <li>•Методы класса</li> </ul>	<ul style="list-style-type: none"> <li>•Да</li> <li>•Да</li> <li>•Да</li> <li>•Да</li> </ul>	<ul style="list-style-type: none"> <li>•Да</li> <li>•Да</li> <li>•Да</li> <li>•Да</li> </ul>
Инкапсуляция	<ul style="list-style-type: none"> <li>•Переменных</li> <li>•Методов</li> </ul>	<ul style="list-style-type: none"> <li>•<b>Закрытые</b></li> <li>•<b>Открытые</b></li> </ul>	<ul style="list-style-type: none"> <li>•<b>Закр., отк., защищ.</b></li> <li>•<b>Закр., отк., защищ.</b></li> </ul>
Модульность	Разновидности модулей	<b>Нет</b>	<b>Файл</b>
Иерархии	<ul style="list-style-type: none"> <li>•Наследование</li> <li>•Шаблоны</li> <li>•Метаклассы</li> </ul>	<ul style="list-style-type: none"> <li>•<b>Один.</b></li> <li>•<b>Нет</b></li> <li>•<b>Да</b></li> </ul>	<ul style="list-style-type: none"> <li>•<b>Множ.</b></li> <li>•<b>Да</b></li> <li>•<b>Нет</b></li> </ul>
Типизация	<ul style="list-style-type: none"> <li>•Сильная типизация</li> <li>•Полиморфизм</li> </ul>	<ul style="list-style-type: none"> <li>•<b>Нет</b></li> <li>•<b>Да</b></li> </ul>	<ul style="list-style-type: none"> <li>•<b>Да</b></li> <li>•<b>Да</b></li> </ul>
Параллельность	Многозадачность	Непрямая	Непрямая
Сохраняемость	Долгоживущие объекты	Нет	Нет

## Достоинства объектно-ориентированного подхода

- Объектно-ориентированные системы более открыты и легче поддаются внесению изменений, поскольку их конструкция **базируется на устойчивых формах**. Возможность развиваться постепенно и не приводит к полной ее переработке даже в случае существенных изменений исходных требований.
- Объектная декомпозиция дает возможность создавать программные системы меньшего размера путем использования общих механизмов (готовых классов). Повышает уровень унификации разработки и пригодность **для повторного использования кода ПО**. Ведет к сборочному созданию ПО.
- Объектная декомпозиция **уменьшает риск** создания сложных систем ПО, так как она предполагает эволюционный путь развития на базе небольших подсистем. Интеграция растягивается на все время разработки, а не превращается в единовременное событие.
- Объектная модель вполне естественна, поскольку ориентирована на **человеческое восприятие мира**, а не на компьютерную реализацию.
- Объектная модель позволяет в полной мере использовать возможности объектно-ориентированных языков.

## Недостатки объектно-ориентированного подхода.

- Психологический фактор: объектная декомпозиция существенно отличается от функциональной. **Нужно учиться думать объектами. ООП – это определенное мировоззрение.**
- **Не дает мгновенной отдачи.** Результаты появляются только после повторного использования накопленных компонентов.
- Многие объекты **реального мира** все таки **сложно представить** в виде системных **объектов**.
- Если при изменении системы требуется изменить интерфейс какого-либо класса, то нужно оценивать эффект от такого изменения с учетом всех пользователей классов.
- При использовании сервисов объекты должны явно ссылаться на имена других объектов и знать их интерфейс.

## Объектно-ориентированное программирование в C++. Что нужно знать.

- Классы. Управление доступом. Указатель `this`.
- Дружественные функции.
- Наследование: одиночное, множественное. Последовательность вызова конструкторов и деструкторов.
- Виртуальные функции. Переопределение функций.
- Полный путь к функциям и данным класса. (`::`)
- Особенности множественного наследования. Разрешение конфликтов. Виртуальные базовые классы.
- Управление доступом при организации наследования.
- Шаблоны функций. Параметризованные классы.

ЧИТАЙТЕ СТРАУСТРУПА «ЯЗЫК ПРОГРАММИРОВАНИЯ C++».

Типы пользовательского интерфейса. Виды взаимодействия пользователя и программы.

- Про пользовательский интерфейс смотри в книге **«Инженерия программного обеспечения»**.

## Аттестация и верификация программного обеспечения.

- Верификация ПО - отвечает на вопрос правильно ли создана система?
- Аттестация ПО – на вопрос правильно ли работает система?
- Аттестация ПО – больше относится к аттестации ПО заказчиком. Когда заказчик проверяет устраивает его система или нет. (Приемо-сдаточные испытания, Опытно-промышленная эксплуатация и т.д.)
- **Хороший тест** выявляет **дефекты программы**, а не доказывает правильность ее работы.

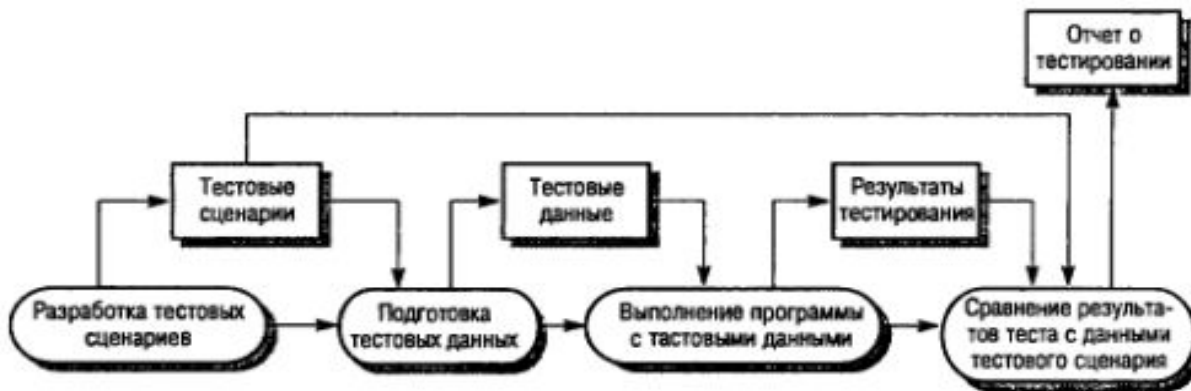


Рис. 20.2. Процесс тестирования дефектов

Схема процесса тестирования.

## Классификация подходов к аттестации и верификации:

### **1. Инспектирование (статические методы)**

1. Инспектирование программ
2. Автоматический статический анализ программ

### **2. Тестирование (динамические методы)**

#### **1. Тестирование компонентов**

1. Метод «черного ящика». Функциональное тестирование.
2. Метод «белого ящика». Структурное тестирование. Тестирование ветвей.

#### **2. Тестирование сборки**

1. Нисходящее тестирование
2. Восходящее тестирование
3. Тестирование интерфейсов
4. Тестирование с «нагрузкой». Оценка «производительности» и «надежности».



## Тестирование

- **Тест**  $T = (INP, OUT)$ . Inp – входные данные (сценарий), OUT – правильные выходные данные (сценарий).
- Процесс тестирования: P – программа.  
 $P(T.INP) = P.OUT$ . Если  $P.OUT = T.OUT$ , тест пройден, иначе УРА!!!  
**ОБНАРУЖИЛИ ДЕФФЕКТ.**
- Как правильно выбрать входные данные для теста?. Выход: **постройте классы эквивалентности.** (Пример был на лекции для процедуры поиска элемента в последовательности)
- Не забывайте, что нужно проверять не только значения лежащие в классе эквивалентности, но и данные **НА ГРАНИЦЕ КЛАССОВ ЭКВИВАЛЕНТНОСТИ.** (метод границ)
- **Функциональное тестирование (черный ящик):**  $T = F$ (Спецификация)
- **Структурное тестирование (белый ящик):**  $T = F$  (Спецификация, код). (Идеальный вариант – протестировать выполнение каждого оператора, выполнить все ветки программы).

# Особенности тестирования объектно-ориентированных систем

- Особенности ООС:
  - Объекты, нечто большее, чем отдельные подпрограммы и функции
  - Объекты, интегрированные в подсистемы обычно слабо связаны между собой и поэтому сложно определить «самый верхний уровень».
  - При анализе повторно используемых компонентов код компонентов может быть недоступным для испытателя.
- Уровни тестирования объектно-ориентированных систем:
  - Тестирование отдельных методов(операций), ассоциированных с объектом.
    - Методы черного и белого ящика. Точно также, как и не для ОО систем.
  - Тестирование отдельных классов
    - Тестирование всех методов, ассоциированных с объектом
    - Проверка атрибутов, ассоциированных с объектом
    - Проверка всевозможных состояний объектов
  - Тестирование кластеров объектов
    - Тестирование сценариев и вариантов использования

## «Заповеди» тестирования.

- **Заповедь 1.** Считайте тестирование ключевой задачей разработки ПС, поручайте его самым квалифицированным и одаренным программистам; нежелательно тестировать свою собственную программу.
- **Заповедь 2.** Хорош тот тест, для которого высока вероятность обнаружить ошибку, а не тот, который демонстрирует правильную работу программы.
- **Заповедь 3.** Готовьте тесты как для правильных, так и для неправильных данных.
- **Заповедь 4.** Избегайте невоспроизводимых тестов, документируйте их пропуск через компьютер; детально изучайте результаты каждого теста.
- **Заповедь 5.** Каждый модуль подключайте к программе только один раз; никогда не изменяйте программу, чтобы облегчить ее тестирование.
- **Заповедь 6.** Пропускайте заново все тесты, связанные с проверкой работы какой-либо программы ПС или ее взаимодействия с другими программами, если в нее были внесены изменения (например, в результате устранения ошибки).

## Менеджмент качества (Теория качества)

- Сертификат ISO 9001 (сертификат ИСО 9001)– это сертификат соответствия системы менеджмента качества (СМК) международному стандарту ISO 9001:2001 (Сертификат ГОСТ Р ИСО 9001-2001).
- Сертифицируется не программный продукт, а система управления производства.
- Получение сертификата во-первых позволяет «привести в порядок» процессы, протекающие на предприятии. Во-вторых, является некой неформальной гарантией качества, особенно для иностранных организаций.
- Примеры реальной СМК смотрите в материалах к курсу.
- Как и любой стандарт в ИСО 9001 большое внимание уделяется оформлению документов (требования к оформлению).
- Сертифицировать можно совершенно любой процесс. Особенности ИТ-процесса заключается в том, что это интеллектуальный труд, и практически не бывает повторяющихся одинаковых проектов.
- Сертификат выдается специализированным сертификационным органом, через определенное время проводятся проверки, подтверждающие соответствие работы организации стандарту.

## CASE – средства

- Классификация CASE-средств: взять описание из «Инженерии программного обеспечения».
- Представлять, что делает CASE-средство «BP Win», «ER Win», «Star UML».
- Кроме средств проектирования, на которые обращено внимание в этом курсе, есть много программ по поддержке процесса разработки. Например, в своей работе я всегда пользовался MVSS ( Microsoft Visual Source Safe) для организации корпоративной разработки. Мы использовали MVSS даже для хранения файлов конфигурации 1С7.7. Сейчас, на платформе 1С8.1 в интегрированную среду разработки вставлен механизм «хранилища конфигурации», который помогает процессу корпоративной разработки.
- Также есть CASE-средства поддерживающие процесс тестирования. Опять пример из 1С: программа «1С Тест центр», которая позволяет проверять работу конфигурации в различных режимах, нагрузках и создавать собственных сценарии тестирования.

## Некоторые технологии разработки ПО

- Тяжелые технологии разработки:
  - RUP – Rational Unified Process  
<http://www.interface.ru/fset.asp?Url=/rational/rupmethoditehnol.htm>
- «Живые» технологии разработки:
  - XP – Extreme Programming
  - SCRUM - <http://www.citforum.ru/SE/project/scrum/>
  - DSDM – Dynamic System Development method
  - и т.д.
- Любой процесс разработки, который приводит к успешному выполнению проекта или группы проектов, с возможностью повторного использования, можно назвать технологией.
- Технология накладывает определенные ограничения на организацию проведения непосредственно работ по созданию ПО.
- По поводу новых технологий не плохая обзорная статья:  
<http://www.cyberguru.ru/programming/programming-theory/coding-methodology-new.html>

## Как бы не был организован процесс разработки на выходе должно получиться **Качественное ПО.**

- **Требование функциональности ( с точки зрения пользователя)**
- **Удобство в эксплуатации** – ПО должно быть удобным в эксплуатации и не требовать чрезмерных усилий пользователя, на которых оно рассчитано. Должна обладать соответствующим пользовательским интерфейсом и справочной документацией.
- **Надежность** — определяется рядом характеристик, таких как безотказность, защищенность, безопасность. Надежность значит, что возможные сбои в программе не приведут к физическому или экономическому ущербу.
- **Эффективность** — работа ПО не должна приводить к расточительному использованию таких системных ресурсов, как память и время работы процессора. Описывается следующими характеристиками: скорость выполнения, объем требуемой памяти, используемое процессорное время.
- **Удобство сопровождения** — возможность усовершенствования (модернизации) ПО в ответ на измененные требования заказчика или пользователей.

## Список литературы

- Иан Соммервилл «Инженерия программного обеспечения».6-е издание.2002 г.
- Гради Буч «Объектно-ориентированный анализ и проектирование с примерами приложений на С++»
- Бьерн Страуструп «Язык программирования С++»
- Дж.Питерсон «Теория сетей Петри и моделирование систем»
- [www.idefinfo.ru](http://www.idefinfo.ru) – все о стандартах IDEFX. Описание стандартов, комментарий и примеры их использования. (IDEF0,IDEF3,DFD,IDEF1X)
- Г.Буч,Д.Рамбо,И.Якобсон «Язык UML. Руководство пользователя»
- [www.omg.org](http://www.omg.org) текущая спецификация UML 2.0
- [www.citforum.ru](http://www.citforum.ru) – ответы на прочие вопросы.