1. Стратегии разработки ПО и модели жизненного цикла. Каскадная модель, спиральная модель и инкрементная стратегии разработки ПО. Перечень, последовательность и содержимое этапов разработки по методологиям. Привести примеры итераций, задач, макетов и моделей.

Стратегии разработки ПО:

* + Однократный проход – водопадная модель

(все исходные требования есть, 1 цикл конструирования, промежуточное ПО не используется)

* + Инкрементная стратегия – инкрементная модель, RAD

(все исходные требования есть, разбиение функций на версии, N циклов, последовательная реализация версий)

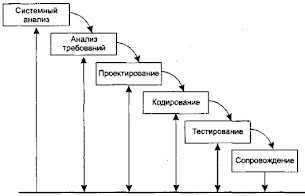
– наращивание ПО.

* + Эволюционная стратегия – спиральная модель, компонентно-ориентированная (не все исходные требования определены, разработка версий – N циклов, промежуточное ПО, уточнение требований)

–последовательная разработка с уточнением требований.

В настоящее время наиболее широко используются три базовые стратегии разработки программного обеспечения: **каскадная; инкрементная; спиральная**. Выбор той или иной стратегии определяется характеристиками: проекта; требований к продукту; команды разработчиков; команды пользователей.

**Каскадная** (водопадная) модель; Особенность - переход на следующий этап после завершения предыдущего.



Этапы ЖЦ каскадной модели:

* Системный анализ
  + требования к системе в целом;
  + роли элементов в компьютерной системе, их взаимодействие;
  + распределение требований по элементам системы;
  + интерфейс ПО с внешней средой (аппаратное обеспечение, люди, БД);
  + оценка объема проектных работ, трудозатрат, рисков;
  + рабочие задачи и план-график работ.
* Анализ требований

По каждому элементу ПО:

* + детализация требований;
  + функции, характеристики, интерфейс;
  + завершение планирование проекта.

=> спецификации анализа

* Проектирование
  + архитектура ПО;
  + модульная структуры;
  + алгоритмы;
  + структуры данных;
  + входной /выходной интерфейс (формы данных);
  + оценка качество ПО.

=> множество проектных представлений

* Кодирование

=> составить текст (код) программы

* Тестирование
  + выявление ошибок в функциях, логике, форме реализации ПО (через исполнение ПО)
* Сопровождение
  + внесение изменений в эксплуатируемое ПО для:
    - удаления ошибок;
    - адаптации к изменениям внешней среды для ПО;
    - усовершенствования ПО по требованиям заказчика.

## Особенности классического ЖЦ:

Особенность - переход на следующий этап после завершения предыдущего.

Плюсы:

* Дает план и временной график по всем этапам;
* Упорядочивает ход конструирования. Минусы:
* Реальный проекты часто отклоняются от стандартной последовательности;
* Цикл основан на точных исходных требованиях, что нереально;
* Результаты доступны заказчику только в конце.

## Инкрементная модель

* Объединяет последовательности и итерации
* Содержит этапы классического жизненного цикла
* Все исходные требования известны
* Разбиение функций системы на версии
* По каждой версии ее конструирование и запуск
* Каждый инкремент – это работающий продукт





Этапы спиральной модели

* Сбор исходных требований
* Начальный анализ риска (неопределенные требования)
* Начальный макет
* Оценка и уточнение требований заказчиком (новые предложения)
* Уточнение требований
* Анализ риска с учетом реакции заказчика
* Макет, моделирование или промежуточное ПО
* Сконструированная система Особенности спиральной модели
* Плюсы:
  + Реально (по эволюции) отражает разработку ПО;
  + – Явный учет риска на каждом витке;
  + – Включение системного подхода в итерационный подход
  + – Моделирование для снижения риска и повышения качества/эффективности ПО.

• Минусы:

– Повышенные требования к заказчику;

– Трудности контроля и управления временем разработки. .

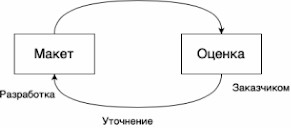
1. Стратегии разработки ПО и модели жизненного цикла.

Макетирование и модель быстрой разработки RAD. Перечень, последовательность и содержимое этапов разработки по методологиям. Примеры итераций, задач, макетов и моделей.

**Макетирование (прототипирование)**

Макетирование – многократный повтор, если

* заказчик не может сформулировать подробные требования по вводу/обработке/ выводу данных;
* разработчик не может определить возможности ПО под ОС, эффективность алгоритма, форму диалога.



**Этапы макетирования**

1. Сбор и уточнение требований (исходных) – (разработчик + заказчик)
2. Быстрое проектирование – (разработчик) – наглядная демонстрация; –

неэффективный алгоритм; – не та ОС, язык программирования и т.д.

1. Построение макета – (разработчик)
2. Оценка макета – (заказчик)
3. Если все в порядке, то к пункту 6, иначе уточнение требований и к пункту 2.
4. Разработка ПО.

**Особенности макетирования**

* + Позволяет снять неопределенность в требованиях заказчика.
  + Формы моделей: – бумажный макет (рис. человеко-машинный диалога); – работающий макет (реализация части функций ПО); – существующее ПО (которое должно быть улучшено).
  + Достоинства: – Возможность определения полных требований к ПО.
  + Недостатки: – Заказчик/разработчик могут принять макет за продукт.

**Модель быстрой разработки приложений RAD (Rapid Application Development)**

* Особенности:
  + все исходные требования известны;
  + проектная область ограничена;
  + экстремально короткий цикл (высокоскоростная адаптация линейной последовательности модели);
  + компонентно-ориентированная разработка.
* Этапы:
  + Разбиение исходных требований на главные функции;
  + Каждая функция реализуется своей группой за 60-90 дней;
  + Интеграция главных функций.



**Этапы RAD**

* Бизнес моделирование
  + Бизнес-функции
  + Информационный поток между бизнес-функциями
  + Информация, руководящая бизнес-процессами
  + Генерируется информация (какая, кто)
  + Где применяется информация?
  + Кто обрабатывает информацию?
* Моделирование данных
  + Информационный поток (набор объектов данных)
  + Определяются свойства, атрибуты, отношения объектов
* Моделирование обработки
  + Определяются преобразования объектов (алгоритмы)
  + Описание обработки для добавления/изменения/удаления/поиска данных
* Генерация приложения
  + Язык программирования 4 и 5 -х поколений (среда)
  + Применение/создание повторно используемых компонентов
  + Утилиты автоматизации
* Тестирование и объединение
  + Новые компоненты тестируются
  + Повторные нет (уменьшается время тестирования)

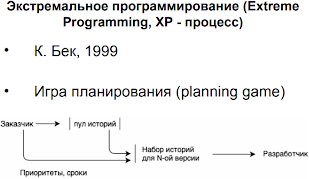
**Ограничения RAD**

* Для больших проектов необходимы большие человеческие ресурсы (много групп);
* RAD применима, если возможна декомпозиция на отдельные модули и производительность не является критической величиной;
* RAD не применима в условиях высоких технических рисков (новых технологиях)

1. Тяжеловесные и облегченные процессы разработки ПО. Экстремальное программирование (XP-процесс). Перечень, последовательность и содержимое этапов разработки по методологиям. Примеры итераций, задач, макетов и моделей.

**Процессы разработки**

* + Прогнозирующие (тяжеловесные) – predictive / heavyweight
  + прогноз работ
  + порядок разработки
  + документирование
  + *Применимы при фиксированных требованиях и многих разработчиках разной квалификации*
  + Адаптивные (подвижные, облегченные) - agile / lightweight
  + меньше документов
  + ориентированы на человека (разработчиков)
  + адаптируют изменения требований
  + *Применимы при частых изменениях требований,*
  + *малой группе высококвалифицированных разработчиков*
  + *и грамотных заказчиках, участвующих в разработке*



**Шаги XP - процесса**

1. Игра планирования (planning game)
2. Частая смена версий (small releases)
   * Запуск в производство новых версий каждые 2 недели
3. Метафора (Metaphor)

– Простое, глобальное, общедоступное представление проекта.

1. Простое проектирование
   * Максимальная простота проектирования;
   * Минимум документации проектирования;
   * При остановке изменений требований – «нафталиновая документация» на 5-10

страниц.

1. Тестирование (Testing)

* «Тестируй, потом кодируй»
* Планирование тестирования (параллельно с анализом требований);
* Тесты модулей непрерывно;
* Тесты функций.

1. Реорганизация (Refactoring)

– Постоянное проектирование с сохранением функциональности (удаление

дублирований, добавление гибкости, упрощение)

1. Парное программирование (Pair programming)

* 2 программиста на 1 ПК;
* Затраты повышаются на 15%;
* Время уменьшается на 40-45%;
* Повышается качество, снижаются затраты на сопровождение.

1. Коллективное владение кодом (Collective ownership)

– Каждый разработчик может изменять любой код в любой момент.

1. Непрерывная интеграция (Continuous integration)

* По каждой задаче (N раз в день);
* Непрерывное регрессионное тестирование (повтор тестов).

1. 40-часовая неделя (40-hour week);
2. Локальный заказчик (On-site customer)

* В группе;
* Может отвечать на вопросы.

1. Стандарты кодирования (Coding standards)

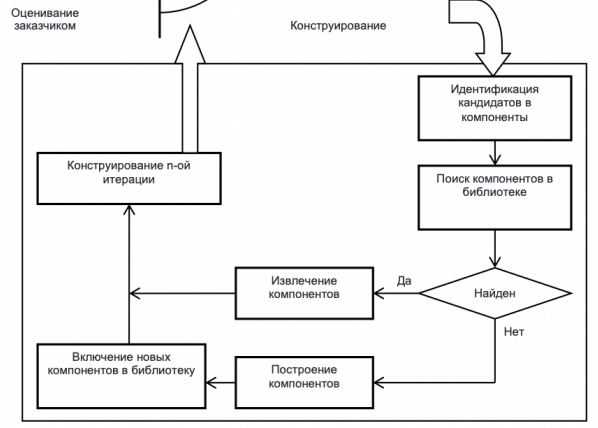
– Правила для одинакового кода во всех частях ПО.

**Особенности XP - процесса**

* + Управление XP проектом
    - через метрики (3-4 штуки), видимые всем.
    - основная метрика - «скорость проекта»
    - количество историй, реализуемых в итерации
  + Плюсы:
    - Ориентирована на малые/средние группы (до 10 человек) при неопределенности/ частом изменении требований;
    - Уменьшение стоимости внесения изменений.
  + Минусы:
    - Обладает социальным воздействием;
    - Все методы необходимо исполнять совместно.



1. Стратегии разработки ПО и модели жизненного цикла. Agile (Scrum) процесс и компонентно-ориентированная модель. Перечень, последовательность и содержимое этапов разработки по методологиям. Примеры итераций, задач, макетов и моделей.

**Компонентно-ориентированная модель**

Особенности:

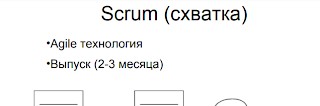
– Повторное использование компонентов;

– Развитие эволюционной модели.

**Основополагающие принципы Agile-манифеста**

1. Наивысшим приоритетом для нас является удовлетворение потребностей заказчика, благодаря регулярной и ранней поставке ценного программного обеспечения.
2. Изменение требований приветствуется, даже на поздних стадиях разработки. Agile- процессы позволяют использовать изменения для обеспечения заказчику конкурентного преимущества.
3. Работающий продукт следует выпускать как можно чаще, с периодичностью от пары недель до пары месяцев.
4. На протяжении всего проекта разработчики и представители бизнеса должны ежедневно работать вместе.
5. Над проектом должны работать мотивированные профессионалы. Чтобы работа была сделана, создайте условия, обеспечьте поддержку и полностью доверьтесь им.
6. Непосредственное общение является наиболее практичным и эффективным способом обмена информацией как с самой командой, так и внутри команды.
7. Работающий продукт — основной показатель прогресса.
8. Инвесторы, разработчики и пользователи должны иметь возможность поддерживать постоянный ритм бесконечно. Agile помогает наладить такой устойчивый процесс разработки.
9. Постоянное внимание к техническому совершенству и качеству проектирования повышает гибкость проекта.
10. Простота искусства минимизации лишней работы — крайне необходима.
11. Самые лучшие требования, архитектурные и технические решения рождаются у самоорганизующихся команд.
12. Команда должна систематически анализировать возможные способы улучшения эффективности и соответственно корректировать стиль своей работы.

**Приоритеты Agile**

* + Более важно:
    - Личности и их взаимодействия, ЧЕМ процессы и инструменты.
    - Работающее ПО, ЧЕМ полная документация.
    - Сотрудничество с заказчиками, ЧЕМ контрольные обязательства.
    - Реакция на изменения, ЧЕМ следование плану.



**Роли в Scrum**

* + Основные роли (свиньи, целиком в проекте):
    - Скрам-мастер – руководит процессом;
    - Владелец проекта;
    - Команда разработчиков 7+/-2 человека, разные специализации (кодировщик,

тестировщик, архитектор и т.п.)

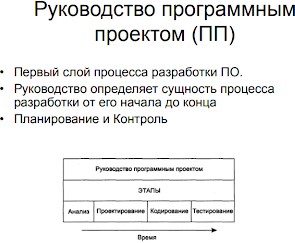
* + Дополнительные роли (куры):
    - Пользователи;
    - Клиенты;
    - Управленцы;
    - Эксперты.

**Термины Scrum**

* + Скрам – рассматриваемая методология управления проектами.
  + Выпуск (release) – временной отрезок, в ходе которого над продуктом ведется работа. Фиксирован по времени. Имеет список задач, подлежащих решению в этот выпуск. Часто завершается выпуском новой версии продукта. Состоит из спринтов.
  + Спринт (итерация) – временной отрезок, в ходе которого над продуктом ведется работа. Фиксирован по времени. Имеет список задач, подлежащих решению в этот спринт.
  + Пожелание заказчика (user story) – функциональность, которую заказчик хочет реализовать в проекте. • Резерв продукта (Product Backlog) – список пожеланий заказчика, которые необходимо реализовать в проекте, отсортирован по важности.
  + Резерв выпуска(Release Backlog) – список пожеланий заказчика, которые необходимо реализовать в текущем выпуске.
  + Резерв спринта(Sprint Backlog) – список пожеланий заказчика, которые необходимо реализовать в текущем спринте.
  + Очки истории (Story Points) – абстрактные очки сложности пожеланий заказчика.
  + График сгорания задач – график, показывающий оптимальный темп решения задач и реальный.

**Этапы Scrum**

1. Планирование
   * команда и владелец выбирают задачи,
   * команда обсуждает технические вопросы
2. Ежедневная встреча (15 минут, говорят свиньи)
   * результаты/планы/проблемы
   * скрам над скрамом приводит N команд
3. Демонстрация инкремента
   * Подводят итоги
   * время ограничено
4. Руководство программным проектом. Планирование проекта. Структурная декомпозиция работ. Ресурсы и риски. Распределение задач и ресурсов. Вехи. Привести определения и примеры.



**Руководство ПП – начало проекта**

Определить в начале проекта:

* + цели и проблемная область;
  + альтернативные решения;
  + выявить технические и управленческие ограничения. Выполнить:
  + Стратегическое планирование
  + Бизнес-моделирование

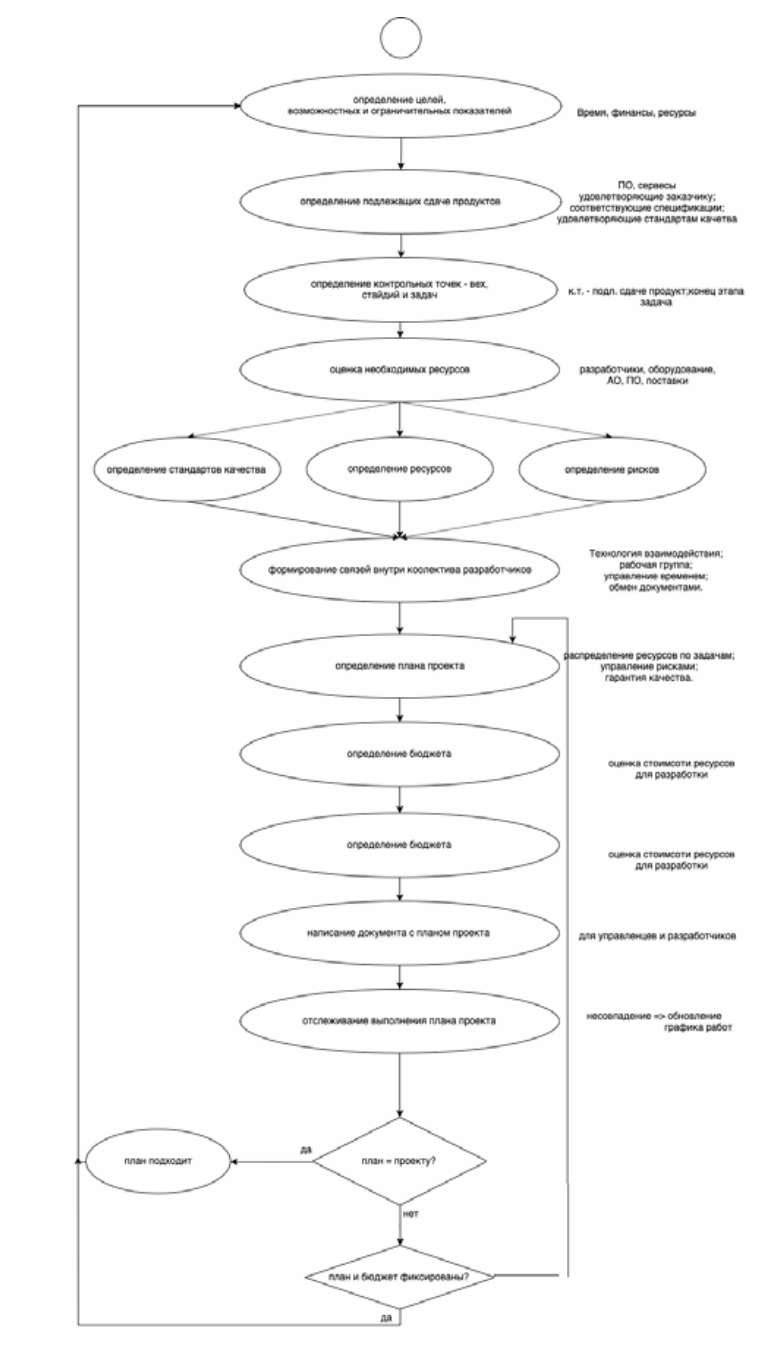
**Действия по руководству ПП**

Определить перед технической работой:

* + требования к АО и ПО;
  + объем работ;
  + возможные риски (и способы управления ими)
  + задачи;
  + вехи; проектные показатели; промежуточные результаты;
  + стоимость; сроки (внешние ограничения)
  + план работ (время и ресурсы по задачам)
  + организация проекта (ограничения коллектива разработчиков)

**Количественные оценки ПП**

* + Мера - количественная характеристика свойства объекта; измеряются прямо.
  + Метрики - вычисляются по формулам на основе других свойств. Оценка перед/в процессе планирования:
    - людские ресурсы – (трудо) затраты (чел-месяц)
    - длительность (день/месяц)
    - стоимость (тысяч Р/$)
  + основана на прошлом опыте (по аналогии)
  + постепенное уточнение; начальное отличие в 4 раза



**Задачи планирования**

* + Составить набор проектных задач (операций)
  + Определить связи между задачами;
  + Определить сложность любой задачи;
  + Определить людские и другие ресурсы
  + Составить сетевой график задач и его временную разметку.

**Распределение работ (WBS)**

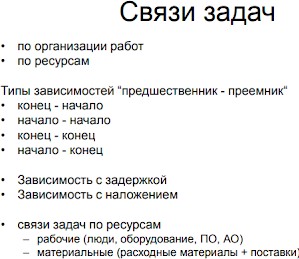
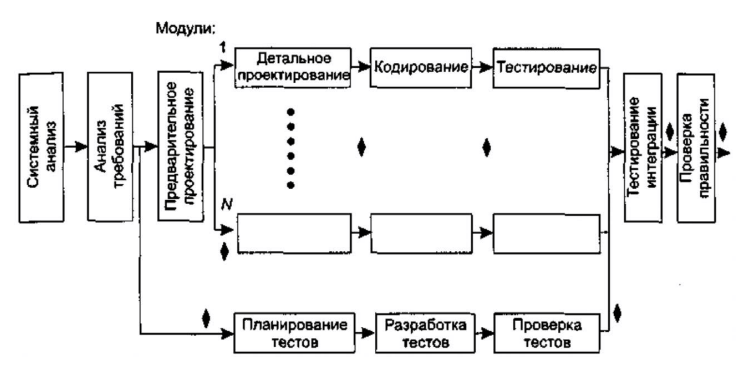
* + вехи через регулярные интервалы (вдоль всего процесса разработки ПО);
  + вехи на документацию;
  + параллельность выполнения задач и межзадачные зависимости.
  + Рекуррентные задачи, иерархия задач (связи по ресурсам и ограничениям);
  + критический путь (=> выполнение проекта в срок)

**Планирование задач проекта**

* + Структурная декомпозиция работ зависит от модели жизненного цикла
* список задач - операция с определением
* имеет продолжительность (1день - 2 недели)
* нужны ресурсы (которые ограничены:сложность перепланирования))
  + Инструментальные средства (WBS)
  + Связи задач
  + Распределение ресурсов по задачам (неполное и избыточное распределение ресурсов)

**Планирование проектных задач**

* + WBS - Work Breakdown Structure (структура распределения работ).
  + составляется с помощью утилиты планирования проекта



**Ресурсы**

* + определить (название, тип, единицы),
  + календарь ресурса - рабочие дни/часы
  + назначение ресурсов задачам
  + график ресурса – график (диаграмма занятости ресурса по дням)





1. Руководство программным проектом. Планирование проекта. Анализ и управление рисками. Типы рисков. Календари ресурсов. Привести определения и примеры.

Планирование проекта (см. предыдущий пункт)

**Анализ рисков**

* + исследование области неопределенности;
  + влияние на проект;
  + поиск скрытых технических проблем;
  + возможность отставания;

В результате принимается решение - выполнять проект или нет.

**Управление рисками**

RE = P(UO) x L(UO),

* RE — показатель риска;
* P(UO) — вероятность неудовлетворительного результата;
* L(UO) — потеря при неудовлетворительном результате.

Неудовлетворительный результат:

* + превышение бюджета,
  + срыв сроков,
  + низкая надежность,
  + неправильное функционирование и т. д

**Действия по управлению рисками**

* + Идентификация риска — выявление элементов риска в проекте.
  + Анализ риска — оценка вероятности и величины потери по каждому элементу риска.
  + Ранжирование риска — упорядочение элементов риска по степени их влияния.
  + Планирование управления риском — подготовка к работе с каждым элементом риска.
  + Разрешение риска — устранение или смягчение элементов риска.
  + Наблюдение риска — отслеживание динамики элементов риска, выполнение корректирующих действий

**Идентификация риска**

Источниками проектного риска являются:

* + выбор бюджета, плана, человеческих ресурсов;
  + формирование требований к программному продукту;
  + сложность, размер и структура программного проекта;
  + методика взаимодействия с заказчиком. К источникам технического риска относят:
  + трудности проектирования, реализации, формирования интерфейса, тестирования и сопровождения;
  + неточность спецификаций;
  + техническая неопределенность или отсталость принятого решения. Источники коммерческого риска включают:
  + создание продукта, не требующегося на рынке;
  + создание продукта, опережающего требования рынка (отстающего от них);
  + потерю финансирования.

**Проверочный список (пример)**

1. Дефицит персонала.
2. Нереальные расписание и бюджет.
3. Разработка неправильных функций и характеристик.
4. Разработка неправильного пользовательского интерфейса.
5. Слишком дорогое обрамление.
6. Интенсивный поток изменения требований.
7. Дефицит поставляемых компонентов.
8. Недостатки в задачах, разрабатываемых смежниками.
9. Дефицит производительности при работе в реальном времени.
10. Деформирование научных возможностей.

**Анализ риска**

* + На основе экспертных оценок или статистики

**Ранжирование риска**

* + По приоритету (по влиянию)
  + Принцип Парето (80% риска от 20% элементов риска)
  + Выявление 20% элементов риска – самые существенные

**Планирование управления рисками**

1. ЭЛЕМЕНТЫ РИСКА - [Ri, Pi, Li, REi]
2. эталонные уровни риска в проекте (выход за уровень приводит к останову проекта)
3. зависимости между [Ri, Pi, Li, REi] и эталонным уровнем.
4. набор эталонных точек, образующих сферу останова. В сфере останова предсказываются области неопределенности.
5. Для каждого элемента риска - план управления «зачем, что, когда, кто, где, как и сколько».
6. План управления каждым элементом риска интегрируется в общий план программного проекта

**Разрешение риска**

* + Уход от риска
    - Перепланирование проекта
    - Изменение требований
  + Ограничение риска
    - Затрагивает часть системы
  + Смягчение риска
    - Реализация риска
    - Наблюдение риска
    - Уход, ограничение, контроль
    - План на случай неожиданности Убийца проекта
    - Устраняется на ранней стадии

**Наблюдение риска**

* + цикличность процесса слежения за риском;
  + вызов корректирующих воздействий.

**Отслеживание 10 верхних элементов риска:**

1. выделение и ранжирование наиболее существенных элементов риска в проекте.
2. планирование регулярных просмотров (проверок) процесса разработки (в группе более 20 человек - ежемесячно, в остальных проектах — чаще).
   * проверка изменений в 10 верхних элементах риска (от 7 до 12),
   * приоритет в предыдущем просмотре,
   * частота попадания элемента в список верхних элементов.
   * Если элемент в списке опустился, он нуждается в наблюдении, но не требует управляющего воздействия. Если элемент поднялся в списке, или только появился в нем, то требует повышенного внимания.

4. исследование проблем в разрешении элементов риска.

**Типы рисков:**

*Временные риски.*

Это вероятность того, что на выполнение задач в проекте уйдёт больше времени, чем запланировано. Помните о сроках, потому что время — это ресурсы. Если команда тратит много времени на задачи, растёт и фонд оплаты труда. Кроме того, стейкхолдеры проекта могут разочароваться из-за задержек.

*Бюджетные риски.*

Из-за плохого планирования стоимость проекта может оказаться больше, чем заложено в бюджете. Обычно бюджет закладывают перед запуском проекта, тогда же планируют траты по статьям. Если команда не уложится в план, потребуются дополнительные средства, и если их не будет, проект остановится.

*Риски изменения объёмов работы.*

Они могут появиться, если исполнители не поняли требований заказчика или он сам внёс в проект изменения. Это может привести к пересмотру бюджета, сроков и списка задач.

*Внешние риски.*

Это потенциальные события, которые находятся за пределами компании и которые компания не может контролировать. Например, на проект могут повлиять новые законы. *Единая точка отказа*.

Так называют единственное событие, которое может остановить всю работу над проектом. Ни один член команды не сможет дальше выполнять свои задачи, пока проблема не решится. Например, для интернет-магазина единой точкой отказа может стать отключение электричества в офисе. Если доступ к инструментам, таким как CRM, был только из офиса, вся команда не сможет выполнять задачи.

В результате команда не выполнит ни одной задачи. Зная об этой точке отказа, можно принять меры: создать резервные копии сервисов и информации в облаке.

*Зависимости*. Это связи между двумя задачами в проекте: когда начало одной задачи зависит от завершения другой. Зависимости часто становятся риском для проекта.

1. Оценка бюджета проекта: экспертная, по аналогии, алгоритмическая, на основе графика выполнения. Точность оценки. Оценка затрат, длительности и стоимости разработки. Метрический базис. Привести определения и примеры.

* Экспертная оценка - оценка на основе консультации с экспертами, точность средняя
  + различные эксперты
  + обсуждение
* Оценка по аналогии - оценка на основе аналогии с похожими проектами,

точность низкая

* На основе графика выполнения
  + план на основе исходных данных
  + стоимость ресурсов (зависит от графика ресурсов; может изменяться)
  + фиксированные затраты
  + нормы и ставки
  + накладывающиеся расходы
  + средняя точность
  + Отслеживание выполнения проекта:
    - план на основе исходных данных. ⇔ фактический план
    - бюджет на основе исходных данных. ⇔ фактический бюджет
  + Причины несоответствия:
    - появление/разбиение задач ( => корректировка плана)
    - недоступность ресурсов
    - система бухучета (не все затраты могут быть учтены) (бухучет и управление проектом не синхронны)
* Алгоритмическая оценка - оценка по определенному алгоритму для которого подбираются настройки, точность высокая
  + Размерно-ориентированные метрики
  + Функционально-ориентированные метрики
  + Модель конструктивной стоимости СОСОМО Желательно использовать ≥ 2 штук для проверки адекватности

1. Оценка затрат, длительности и стоимости разработки. Оценка размера проекта на основе LOC, FP или OP. Привести определения и примеры.

## LOC (Lines of code) и FP

Цель этой деятельности — сформировать предварительные оценки, которые позволят:

* предъявить заказчику корректные требования по стоимости и затратам на разработку программного продукта;
* составить план программного проекта.

При выполнении оценки возможны два варианта использования LOC- и FP-

данных:

* в качестве оценочных переменных, определяющих размер каждого элемента продукта;
* в качестве метрик, собранных за прошлые проекты и входящих в метрический базис фирмы.

Обсудим шаги процесса оценки.

1. *Шаг 1.* Область назначения проектируемого продукта разбивается на ряд функций, каждую из которых можно оценить индивидуально:

*f1, f2,…,fn.*

1. *Шаг 2.* Для каждой функции *fi*, планировщик формирует лучшую LOCлучш*i* (FРлучш*i*), худшую LOCхудш*i*(FРхудш*i*) и вероятную оценку LOC вероятнi (FР вероятнi). Используются опытные данные (из метрического базиса) или интуиция. Диапазон значения оценок соответствует степени предусмотренной неопределенности.
2. *Шаг 3.* Для каждой функции/ в соответствии с -распределением вычисляется ожидаемое значение LOC- (или FP-) оценки:

LOC*ожi* =(LOCлучш*i* + LOCхудш*i* +4x LOCвероятн*i* )/ 6.

1. *Шаг 4.* Определяется значение LOC- или FP-производительности разработки функции.

Используется один из трех подходов:

* 1. для всех функций принимается одна и та же метрика средней производительности ПРОИЗВср, взятая из метрического базиса;
  2. для *i*-й функции на основе метрики средней производительности вычисляется настраиваемая величина производительности:

ПРОИЗВ*i* =ПРОИЗВсрх(LOCср /LOCож*i*),

где LOCcp — средняя LOC-оценка, взятая из метрического базиса

(соответствует средней производительности);

* 1. для *i*-й функции настраиваемая величина производительности вычисляется по аналогу, взятому из метрического базиса:

ПРОИЗВ*i* =ПРОИЗВан*i*х(LOCан*i* /LOCож*i*).

Первый подход обеспечивает минимальную точность (при максимальной простоте вычислений), а третий подход — максимальную точность (при максимальной сложности вычислений).

1. *Шаг* 5. Вычисляется общая оценка затрат на проект: для первого подхода

для второго и третьего подходов



1. *Шаг 6.* Вычисляется общая оценка стоимости проекта: для первого и второго подходов



где УД\_СТОИМОСТЬср — метрика средней стоимости одной строки, взятая из метрического базиса.

для третьего подхода



где УД\_СТОИМОСТЬан*i* — метрика стоимости одной строки аналога, взятая из метрического базиса. Пример применения данного процесса оценки приведем ниже.

**OP** - модель композиции используется на ранней стадии конструирования ПО, когда:

* рассматривается макетирование пользовательских интерфейсов;
* обсуждается взаимодействие ПО и компьютерной системы;
* оценивается производительность; определяется степень зрелости технологии.

Модель композиции приложения ориентирована на применение объектных указателей. Объектный указатель - средство косвенного измерения ПО, для его расчета определяется количество экранов (как элементов пользовательского интерфейса), отчётов и компонентов, требуемых для построения приложения. каждый объектный экземпляр (экран, отчет) относят к одному из трех уровней сложности. Здесь места подстановки измеренных и вычисленных значений отмечены прямоугольниками (прямоугольник играет роль метки- заполнителя). В свою очередь, сложность является функцией от параметров клиентских и серверных таблиц данных), которые требуются для генерации экрана и отчета, а также от количества представлений и секций, входящих в экран или отчет

Для учета реальных условий разработки вычисляется процент повторного использования программных компонентов %REUSE и определяется количество новых объектных указателей NOP:

NOP = (Объектные указатели) х [(100 - %REUSE) /100].

Для оценки затрат, основанной на величине NOP, надо знать скорость разработки продукта PROD. Эту скорость определяют по табл. 2.18, учитывающей уровень опытности разработчиков и зрелость среды разработки.

Проектные затраты оцениваются по формуле

ЗАТРАТЫ = NOP /PROD [чел.-мес],

где PROD — производительность разработки, выраженная в терминах объектных указателей.

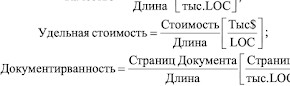
1. Оценка затрат, длительности и стоимости разработки. Размерно- ориентированные и функционально-ориентированные метрики, их соотношение. Привести определения и примеры.

**Размерно-ориентированные метрики** прямо измеряют программный продукт и процесс его разработки. Основываются размерно-ориентированные метрики на LOC-оценках (Lines Of Code). LOC-оценка — это количество строк в программном продукте.

Исходные данные для расчета этих метрик сводятся в таблицу

На основе таблицы вычисляются размерно-ориентированные метрики производительности и качества (для каждого проекта):





.

*Достоинства размерно-ориентированных метрик:*

1. широко распространены;
2. просты и легко вычисляются.

*Недостатки размерно-ориентированных метрик:*

1. зависимы от языка программирования;
2. требуют исходных данных, которые трудно получить на начальной стадии проекта;
3. не приспособлены к непроцедурным языкам программирования.

**Функционально-ориентированные метрики** косвенно измеряют программный продукт и процесс его разработки. Вместо подсчета LOC-оценки при этом рассматривается не размер, а функциональность или полезность продукта.

Используется 5 информационных характеристик.

* 1. *Количество внешних вводов.* Подсчитываются все вводы пользователя, по которым поступают разные прикладные данные. Вводы должны быть отделены от запросов, которые подсчитываются отдельно.
  2. *Количество внешних выводов.* Подсчитываются все выводы, по которым к пользователю поступают результаты, вычисленные программным приложением. В этом контексте выводы означают отчеты, экраны, распечатки, сообщения об ошибках. Индивидуальные единицы данных внутри отчета отдельно не подсчитываются.
  3. *Количество внешних запросов.* Под запросом понимается диалоговый ввод, который приводит к немедленному программному ответу в форме диалогового вывода. При этом диалоговый ввод в приложении не сохраняется, а диалоговый вывод не требует выполнения вычислений. Подсчитываются все запросы — каждый учитывается отдельно.
  4. *Количество внутренних логических файлов.* Подсчитываются все логические файлы (то есть логические группы данных, которые могут быть частью базы данных или отдельным файлом).
  5. *Количество внешних интерфейсных файлов.* Подсчитываются все логические файлы из других приложений, на которые ссылается данное приложение.

Вводы, выводы и запросы относят к категории *транзакция.* Транзакция — это элементарный процесс, различаемый пользователем и перемещающий данные между внешней средой и программным приложением. В своей работе транзакции используют внутренние и внешние файлы. Приняты следующие определения.

*Внешний ввод* — элементарный процесс, перемещающий данные из внешней среды в приложение. Данные могут поступать с экрана ввода или из другого приложения. Данные могут использоваться для обновления внутренних логических файлов. Данные могут содержать как управляющую, так и деловую информацию. Управляющие данные не должны модифицировать внутренний логический файл.

*Внешний вывод —* элементарный процесс, перемещающий данные, вычисленные в приложении, во внешнюю среду. Кроме того, в этом процессе могут обновляться внутренние логические файлы. Данные создают отчеты или выходные

файлы, посылаемые другим приложениям. Отчеты и файлы создаются на основе внутренних логических файлов и внешних интерфейсных файлов. Дополнительно этот процесс может использовать вводимые данные, их образуют критерии поиска и параметры, не поддерживаемые внутренними логическими файлами. Вводимые данные поступают извне, но носят временный характер и не сохраняются во внутреннем логическом файле.

*Внешний запрос —* элементарный процесс, работающий как с вводимыми, так и с выводимыми данными. Его результат — данные, возвращаемые из внутренних логических файлов и внешних интерфейсных файлов. Входная часть процесса не модифицирует внутренние логические файлы, а выходная часть не несет данных, вычисляемых приложением (в этом и состоит отличие запроса от вывода).

*Внутренний логический файл* — распознаваемая пользователем группа логически связанных данных, которая размещена внутри приложения и обслуживается через внешние вводы.

*Внешний интерфейсный файл —* распознаваемая пользователем группа логически связанных данных, которая размещена внутри другого приложения и поддерживается им. Внешний файл данного приложения является внутренним логическим файлом в другом приложении.

Каждой из выявленных характеристик ставится в соответствие сложность. Для этого характеристике назначается низкий, средний или высокий ранг, а затем формируется числовая оценка ранга.

Для транзакций ранжирование основано на количестве ссылок на файлы и количестве типов элементов данных. Для файлов ранжирование основано на количестве типов элементов-записей и типов элементов данных, входящих в файл.

Тип элемента-записи — подгруппа элементов данных, распознаваемая пользователем в пределах файла.

Тип элемента данных — уникальное не рекурсивное (неповторяемое) поле, распознаваемое пользователем.

В конечном счете все сводится в таблицу

После сбора всей необходимой информации приступают к расчету метрики —

*количества функциональных указателей FP* (Function Points).

FP = Общее количество х (0,65+ 0,01 x),

где *Fi* — коэффициенты регулировки сложности.

Каждый коэффициент может принимать следующие значения: 0 — нет влияния, 1 — случайное, 2 — небольшое, 3 — среднее, 4 — важное, 5 — основное.

После вычисления FP на его основе формируются метрики производительности, качества и т. д.:



Область применения метода функциональных указателей — коммерческие информационные системы.

*Достоинства функционально-ориентированных метрик:*

1. Не зависят от языка программирования.
2. Легко вычисляются на любой стадии проекта.

*Недостаток функционально-ориентированных метрик:* результаты основаны на субъективных данных, используются не прямые, а косвенные измерения. FP- оценки легко пересчитать в LOC-оценки. Как показано в табл. 2.13, результаты пересчета зависят от языка программирования, используемого для реализации ПО.

1. Вычисление затрат, длительности и стоимости разработки на основе моделей СОСОМО. Модели композиции приложения, раннего этапа проектирования и этапа пост-архитектуры. Привести определения и примеры.

СОСОМО (Constructive Cost Model) - конструктивная модель стоимости, в данной модели для вывода формул использовался статистический подход — учитывались реальные результаты огромного количества проектов. Автор ввел в ее состав три разные по сложности статистические подмодели.

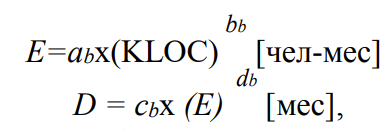
Иерархию подмоделей Боэма (версии 1981 года) образуют:

* *базисная СОСОМО* — статическая модель, вычисляет затраты разработки и ее стоимость как функцию размера программы;
* *промежуточная СОСОМО* — дополнительно учитывает атрибуты стоимости, включающие основные оценки продукта, аппаратуры, персонала и проектной среды;
* *усовершенствованная СОСОМО* — объединяет все характеристики промежуточной модели, дополнительно учитывает влияние всех атрибутов стоимости на каждый этап процесса разработки ПО (анализ, проектирование, кодирование, тестирование и т. д.).

Подмодели СОСОМО могут применяться к трем типам программных проектов. По терминологии Боэма, их образуют:

* *распространенный тип —* небольшие программные проекты, над которыми работает небольшая группа разработчиков с хорошим стажем работы, устанавливаются мягкие требования к проекту;
* *полунезависимый тип —* средний по размеру проект, выполняется группой разработчиков с разным опытом, устанавливаются как мягкие, так и жесткие требования к проекту;
* *встроенный тип —* программный проект разрабатывается в условиях жестких аппаратных, программных и вычислительных ограничений.

Уравнения базовой подмодели имеют вид



где *Е* — затраты в человеко-месяцах, *D —* время разработки, KLOC — количество строк в программном продукте.

Коэффициенты *аb, bb, сb, db* берутся из табл.



В 1995 году Боэм ввел более совершенную модель СОСОМО II, ориентированную на применение в программной инженерии XXI века [21].

В состав СОСОМО II входят:

* Модель композиции приложения;

используется на ранней стадии конструирования ПО, когда:

* + q рассматривается макетирование пользовательских интерфейсов;
  + q обсуждается взаимодействие ПО и компьютерной системы;
  + q оценивается производительность;
  + q определяется степень зрелости технологии.

Проектные затраты оцениваются по формуле ЗАТРАТЫ = NOP /PROD [чел.-мес],где PROD — производительность разработки, выраженная в терминах объектных указателей

* Модель раннего этапа проектирования;

используется в период, когда стабилизируются требования и определяется базисная программная архитектура.

Основное уравнение этой модели имеет следующий вид:

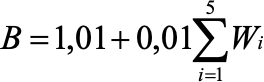
ЗАТРАТЫ = А х РАЗМЕРв х Ме + ЗАТРАТЫаuto[чел.-мес],

где:

* + масштабный коэффициент *А* = 2,5;
  + показатель *В* отражает нелинейную зависимость затрат от размера проекта

(размер системы РАЗМЕР выражается в тысячах LOC);

* + множитель поправки *Мe* зависит от 7 формирователей затрат, характеризующих продукт, процесс и персонал;
  + слагаемое 3ATPATЫauto отражает затраты на автоматически генерируемый программный код.

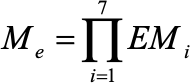
Значение показателя степени *В* изменяется в диапазоне 1,01... 1,26, зависит от пяти масштабных факторов *Wi* и вычисляется по формуле

Общая характеристика масштабных факторов приведена в табл. 2.19, позволяет определить оценки этих факторов. Оценки принимают 6 значений: от очень низкой

(5) до сверхвысокой (0).



Для каждого формирователя затрат определяется оценка (по 6-балльной шкале), где 1 соответствует очень низкому значению, а 6 — сверхвысокому значению. На основе оценки для каждого формирователя по таблице Боэма определяется множитель затрат *EMi* Перемножение всех множителей затрат формирует множитель поправки:



Слагаемое 3ATPATbIauto используется, если некоторый процент программного кода генерируется автоматически. Поскольку производительность такой работы значительно выше, чем при ручной разработке кода, требуемые затраты вычисляются отдельно, по следующей формуле:

ЗАТРАТЫаuto = (КALOC x (AT /100)) / ATPROD,

где:

* KALOC — количество строк автоматически генерируемого кода (в тысячах строк);
* AT — процент автоматически генерируемого кода (от всего кода системы);
* ATPROD — производительность автоматической генерации кода.

Сомножитель AT в этой формуле позволяет учесть затраты на организацию взаимодействия автоматически генерируемого кода с оставшейся частью системы.

Далее затраты на автоматическую генерацию добавляются к затратам, вычисленным для кода, разработанного вручную.

* Модель этапа пост-архитектуры.

Модель этапа постархитектуры используется в период, когда уже сформирована архитектура и выполняется дальнейшая разработка программного продукта.

Основное уравнение постархитектурной модели является развитием уравнения предыдущей модели и имеет следующий вид:

ЗАТРАТЫ = *А* х *К~req* х РАЗМЕР*B* х *Мр* +3ATPATЫauto [чел.-мес],

где

* коэффициент *К~req* учитывает возможные изменения в требованиях;
* показатель *В* отражает нелинейную зависимость затрат от размера проекта (размер выражается в KLOC), вычисляется так же, как и в предыдущей модели;
* в размере проекта различают две составляющие — новый код и повторно используемый код;
* множитель поправки *Мр* зависит от 17 факторов затрат, характеризующих продукт, аппаратуру, персонал и проект.

Изменчивость требований приводит к повторной работе, требуемой для учета

предлагаемых изменений, оценка их влияния выполняется по формуле

*К~req* =l + (BRAK/100),

где BRAK — процент кода, отброшенного (модифицированного) из-за изменения требований.

Размер проекта и продукта определяют по выражению

РАЗМЕР = PA3MEPnew + PA3MEPreuse [KLOC],

где

* PA3MEPnew — размер нового (создаваемого) программного кода;
* PA3MEPreuse — размер повторно используемого программного кода.

Формула для расчета размера повторно используемого кода записывается следующим образом:

PA3MEPreuse =KASLOC x ((100 - *AT)*/ 100) x (*AA* + *SU +* 0,4 *DM +* 0,3 *CM +* 0,3

*IM*) /100,

где

* KASLOC — количество строк повторно используемого кода, который должен быть модифицирован (в тысячах строк);
* AT — процент автоматически генерируемого кода;
* DM — процент модифицируемых проектных моделей;
* CM — процент модифицируемого программного кода;
* IM — процент затрат на интеграцию, требуемых для подключения повторно используемого ПО;
* SU — фактор, основанный на стоимости понимания добавляемого ПО; изменяется от 50 (для сложного неструктурированного кода) до 10 (для хорошо написанного объектно-ориентированного кода);
* АА — фактор, который отражает стоимость решения о том, может ли ПО быть повторно используемым; зависит от размера требуемого тестирования и оценивания (величина изменяется от 0 до 8).

Правила выбора этих параметров приведены в руководстве по СОСОМО II.

Для определения множителя поправки *Мр* основного уравнения используют 17 факторов затрат, которые могут быть разбиты на 4 категории. Перечислим факторы затрат, сгруппировав их по категориям.

*Факторы продукта:*

1. требуемая надежность ПО — RELY;
2. размер базы данных — DATA;
3. сложность продукта — CPLX;
4. требуемая повторная используемость — RUSE;
5. документирование требований жизненного цикла — DOCU.

*Факторы платформы (виртуальной машины):*

1. ограничения времени выполнения — TIME;
2. ограничения оперативной памяти — STOR;
3. изменчивость платформы — PVOL.

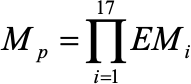
*Факторы персонала:*

1. возможности аналитика — АСАР;
2. возможности программиста — РСАР;
3. опыт работы с приложением — АЕХР;
4. опыт работы с платформой — РЕХР;
5. опыт работы с языком и утилитами — LTEX;
6. непрерывность персонала — PCON.

*Факторы проекта:*

1. использование программных утилит — TOOL;
2. мультисетевая разработка — SITE;
3. требуемый график разработки — SCED.

Для каждого фактора определяется оценка (по 6-балльной шкале). На основе оценки

для каждого фактора по таблице Боэма определяется множитель затрат *ЕМi.* Перемножение всех множителей затрат дает множитель поправки пост- архитектурной модели:

Значение *Мр* отражает реальные условия выполнения программного проекта и позволяет троекратно увеличить (уменьшить) начальную оценку затрат.

От оценки затрат легко перейти к стоимости проекта. Переход выполняют по формуле:

СТОИМОСТЬ = ЗАТРАТЫ х РАБ\_КОЭФ,

где среднее значение рабочего коэффициента составляет $15000 за человеко-месяц. После определения затрат и стоимости можно оценить длительность разработки. Модель СОСОМО IIсодержит уравнение для оценки календарного времени TDEV, требуемого для выполнения проекта. Для моделей всех уровней справедливо: Длительность (TDEV) = [3,0 х (ЗАТРАТЫ)(0,33+0,2(*B*-1,01))] х

SCEDPercentage/100 [мес],

где

*В* — ранее рассчитанный показатель степени;

SCEDPercentage — процент увеличения (уменьшения) номинального графика.

Для описания моделей СОСОМО II требуется информация о размере программного продукта. Возможно использование LOC-оценок, объектных указателей, функциональных указателей.

1. Оценка затрат, длительности и стоимости разработки. Анализ чувствительности программного проекта с применением модели СОСОМО. Цель и методы выполнения. Привести определения и примеры.

СОСОМО II — авторитетная и многоплановая модель, позволяющая решать самые разнообразные задачи управления программным проектом.

Рассмотрим возможности этой модели в задачах анализа чувствительности —

чувствительности программного проекта к изменению условий разработки.

Будем считать, что корпорация «СверхМобильныеСвязи» заказала разработку ПО для встроенной космической системы обработки сообщений. Ожидаемый размер ПО — 10 KLOC, используется серийный микропроцессор. Примем, что масштабные факторы имеют номинальные значения (показатель степени *В =* 1,16) и что автоматическая генерация кода не предусматривается. К проведению разработки привлекаются главный аналитик и главный программист высокой квалификации, поэтому средняя зарплата в команде составит $ 6000 в месяц. Команда имеет годовой опыт работы с этой проблемной областью и полгода работает с нужной аппаратной платформой.

В терминах СОСОМО II проблемную область (область применения продукта) классифицируют как «операции с приборами» со следующим описанием: встроенная система для высокоскоростного мультиприоритетного обслуживания удаленных линий связи, обеспечивающая возможности диагностики.

Оценку пост-архитектурных факторов затрат для проекта сведем в табл. 2.27.

Из таблицы следует, что увеличение затрат в 1,3 раза из-за очень высокой

сложности продукта уравновешивается их уменьшением вследствие высокой квалификации аналитика и программиста, а также активного использования программных утилит.



Рассчитаем затраты и стоимость проекта:

ЗАТРАТЫ = AхРАЗМЕР*B*хМр=2,5(10)1,16х1,088=36x1,088= 39[чел.-мес],

СТОИМОСТЬ = ЗАТРАТЫ х $6000 = $234 000.

Таковы стартовые условия программного проекта. А теперь обсудим несколько сценариев возможного развития событий.

Сценарий понижения зарплаты

Положим, что заказчик решил сэкономить на зарплате разработчиков. Рычаг — понижение квалификации аналитика и программиста. Соответственно, зарплата сотрудников снижается до $5000. Оценки их возможностей становятся номинальными, а соответствующие множители затрат принимают единичные значения:

*EM*ACAP*=EM*PCAP*=1.*

Следствием такого решения является возрастание множителя поправки Мр= 1,507, а также затрат и стоимости:

ЗАТРАТЫ = З6х 1,507 = 54 [чел.-мес],

СТОИМОСТЬ = ЗАТРАТЫ х$5000 = $270 000,

Проигрыш\_ в\_стоимости = $36 000.

Сценарий наращивания памяти

Положим, что разработчик предложил нарастить память — купить за $1000 чип ОЗУ емкостью 96 Кбайт (вместо 64 Кбайт). Это меняет ограничение памяти (используется не 70%, а 47%), после чего фактор STORснижается до номинального:

*EM* =1-> *М* =1,026,

STOR *р*

ЗАТРАТЫ = 36x1,026 = 37 [чел.-мес],

СТОИМОСТЬ = ЗАТРАТЫ х $6000 = $222000,

Выигрыш\_ в\_стоимости = $ 12 000.

Сценарий использования нового микропроцессора

Положим, что заказчик предложил использовать новый, более дешевый МП (дешевле на $1000). К чему это приведет? Опыт работы с его языком и утилитами понижается от номинального до очень низкого и *EM*LTEX =1,22, а разработанные для него утилиты (компиляторы, ассемблеры и отладчики) примитивны и ненадежны (в результате фактор TOOL понижается от высокого до очень низкого и *EM*ТООL= 1,24):

*Мр =* (1,088 / 0,86) х 1,22 x 1,24 = 1,914, ЗАТРАТЫ = 36х1,914 = 69[чел.-мес], СТОИМОСТЬ = ЗАТРАТЫ х $6000 = $414000,

Проигрыш\_в\_стоимости = $180000.

Сценарий уменьшения средств на завершение проекта

Положим, что к разработке принят сценарий с наращиванием памяти: ЗАТРАТЫ = 36 х 1,026 = 37 [чел.-мес],

СТОИМОСТЬ = ЗАТРАТЫ х $6000 = $222000.

Кроме того, предположим, что завершился этап анализа требований, на который было израсходовано $22 000 (10% от бюджета). После этого на завершение проекта осталось $200 000.

Допустим, что в этот момент «коварный» заказчик сообщает об отсутствии у него достаточных денежных средств и о предоставлении на завершение разработки только $170 000 (15%-ное уменьшение оплаты).

Для решения этой проблемы надо установить возможные изменения факторов затрат, позволяющие уменьшить оценку затрат на 15%.

**Первое решение:** уменьшение размера продукта (за счет исключения некоторых функций). Нам надо определить размер минимизированного продукта. Будем исходить из того, что затраты должны уменьшиться с 37 до 31,45 чел.-мес. Решим уравнение:

2,5 (НовыйРазмер)1,16= 31,45 [чел.-мес].

Очевидно, что

(НовыйРазмер)1,16 = 12,58,

(НовыйРазмер)1,16 = 12,581/1,16 = 8,872 [KLOC].

Другие решения:

* q уменьшить требуемую надежность с номинальной до низкой. Это сокращает стоимость проекта на 12% (*EM*RELY изменяется с 1 до 0,88). Такое решение приведет к увеличению затрат и трудностей при применении и сопровождении;
* q повысить требования к квалификации аналитиков и программистов (с высоких до очень высоких). При этом стоимость проекта уменьшается на 15-

19%. Благодаря программисту стоимость может уменьшиться на (1 - 0,74/0,87) х 100% = 15%. Благодаря аналитику стоимость может понизиться на (1 - 0,67/0,83) х 100% = 19%. Основная трудность — поиск специалистов такого класса (готовых работать за те же деньги);

* q повысить требования к опыту работы с приложением (с номинальных до очень высоких) или требования к опыту работы с платформой (с низких до высоких). Повышение опыта работы с приложением сокращает стоимость проекта на (1- 0,81) х 100% = 19%; повышение опыта работы с платформой сокращает стоимость проекта на (1 - 0,88/1,12) х 100% = 21,4%. Основная трудность — поиск экспертов (специалистов такого класса);
* q повысить уровень мультисетевой разработки с низкого до высокого. При этом стоимость проекта уменьшается на (1 - 0,92/1,1) х 100% = 16,4%;
* q ослабить требования к режиму работы в реальном времени. Предположим, что 70%-ное ограничение по времени выполнения связано с желанием заказчика обеспечить обработку одного сообщения за 2 мс. Если же заказчик согласится на увеличение среднего времени обработки с 2 до 3 мс, то ограничение по времени станет равно (2 мс/3 мс) х 70% = 47%, в результате чего фактор TIME уменьшится с высокого до номинального, что приведет к экономии затрат на (1- 1/1,11) х 100%= 10%;
* q учет других факторов затрат не имеет смысла. Некоторые факторы (размер базы данных, ограничения оперативной памяти, требуемый график разработки) уже имеют минимальные значения, для других трудно ожидать быстрого улучшения (использование программных утилит, опыт работы с языком и утилитами), третьи имеют оптимальные значения (требуемая повторная используемость, документирование требований жизненного цикла). На некоторые разработчик почти не может повлиять (сложность продукта, изменчивость платформы). Наконец, житейские неожиданности едва ли позволят улучшить принятое значение фактора «непрерывность персонала».

Какое же решение следует выбрать? Наиболее целесообразное решение — исключение отдельных функций продукта. Вторым (по предпочтительности) решением является повышение уровня мультисетевой разработки (все равно это придется сделать в ближайшее время). В качестве третьего решения можно рассматривать ослабление требований к режиму работы в реальном времени. Принятие же других решений зависит от наличия необходимых специалистов или средств разработки. Впрочем, окончательное решение должно выбираться в процессе переговоров с заказчиком, когда учитываются все соображения.

Выводы.

1. Факторы затрат оказывают существенное влияние на выходные параметры программного проекта.
2. Модель СОСОМО II предлагает широкий спектр факторов затрат, учитывающих большинство реальных ситуаций в «жизни» программного проекта.
3. Модель СОСОМО II обеспечивает перевод качественного обоснования решения менеджера на количественные рельсы, тем самым повышая объективность принимаемого решения.
4. Проектирование программных систем. Декомпозиция подсистем на модули. Модуль и модульность. Виды связности модуля. Виды сцепление модулей. Привести определения и примеры.

Известны два типа моделей модульной декомпозиции:

* q модель потока данных;
* q модель объектов.

В основе модели потока данных лежит разбиение по функциям.

Модель объектов основана на слабо сцепленных сущностях, имеющих собственные наборы данных, состояния и наборы операций.

Очевидно, что выбор типа декомпозиции должен определяться сложностью разбиваемой подсистемы.

**Модульность**

Модуль — фрагмент программного текста, являющийся строительным блоком для физической структуры системы. Как правило, модуль состоит из интерфейсной части и части-реализации.

Модульность — свойство системы, которая может подвергаться декомпозиции на ряд внутренне связанных и слабо зависящих друг от друга модулей.

По определению Г. Майерса, модульность — свойство ПО, обеспечивающее интеллектуальную возможность создания сколь угодно сложной программы. Проиллюстрируем эту точку зрения.

Пусть *С(х) —* функция сложности решения проблемы *х, Т(х) —* функция затрат времени на решение проблемы *х.* Для двух проблем *р1* и *р2* из соотношения *С(р1)*

*> С(р2)* следует, что

*T(pl)>T(p2).*

Этот вывод интуитивно ясен: решение сложной проблемы требует большего времени.

Далее. Из практики решения проблем человеком следует:

*С(р1+ р2)>С(р1) + С(р2).*

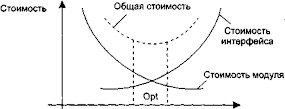
Отсюда с учетом соотношения (4.1) запишем:

*T(pl+p2)>T(pl) + T(p2).*

Соотношение — это обоснование модульности. Оно приводит к заключению

«разделяй и властвуй» — сложную проблему легче решить, разделив ее на управляемые части. Результат, выраженный неравенством, имеет важное значение для модульности и ПО. Фактически, это аргумент в пользу модульности.

Однако здесь отражена лишь часть реальности, ведь здесь не учитываются затраты на межмодульный интерфейс. Как показано на рисунке, с увеличением количества модулей (и уменьшением их размера) эти затраты также растут.





Таким образом, существует оптимальное количество модулей Opt, которое приводит к минимальной стоимости разработки. Увы, у нас нет необходимого опыта для гарантированного предсказания Opt. Впрочем, разработчики знают, что оптимальный модуль должен удовлетворять двум критериям:

* q снаружи он проще, чем внутри;
* q его проще использовать, чем построить.

**Связность модуля (Cohesion)** — это мера зависимости его частей. Связность

— внутренняя характеристика модуля. Чем выше связность модуля, тем лучше результат проектирования, то есть тем «черней» его ящик (капсула, защитная оболочка модуля), тем меньше «ручек управления» на нем находится и тем проще эти «ручки».

Для измерения связности используют понятие силы связности (СС). Существует

7 типов связности:

1. *Связность по совпадению* (СС=0). В модуле отсутствуют явно выраженные внутренние связи.
2. *Логическая связность* (СС=1). Части модуля объединены по принципу функционального подобия. Например, модуль состоит из разных подпрограмм обработки ошибок. При использовании такого модуля клиент выбирает только одну из подпрограмм.

Недостатки:

q сложное сопряжение;

q большая вероятность внесения ошибок при изменении сопряжения ради одной из функций.

1. *Временная связность* (СС=3). Части модуля не связаны, но необходимы в один и тот же период работы системы.

Недостаток: сильная взаимная связь с другими модулями, отсюда — сильная чувствительность внесению изменений.

1. *Процедурная связность* (СС=5). Части модуля связаны порядком выполняемых ими действий, реализующих некоторый сценарий поведения.
2. *Коммуникативная связность* (СС=7). Части модуля связаны по данным

(работают с одной и той же структурой данных).

1. *Информационная (последовательная) связность* (СС=9). Выходные данные одной части используются как входные данные в другой части модуля.
2. *Функциональная связность* (СС=10). Части модуля вместе реализуют одну функцию.

Отметим, что типы связности 1,2,3 — результат неправильного планирования архитектуры, а тип связности 4 — результат небрежного планирования архитектуры приложения.

Общая характеристика типов связности представлена в табл



**Сцепление (Coupling)** — мера взаимозависимости модулей поданным.

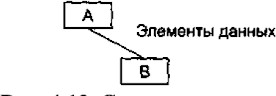
Сцепление — внешняя характеристика модуля, которую желательно уменьшать.

Количественно сцепление измеряется степенью сцепления (СЦ). Выделяют 6

типов сцепления.

1. *Сцепление по данным* (СЦ=1). Модуль А вызывает модуль В.

Все входные и выходные параметры вызываемого модуля — простые элементы данных (рис. 4.13).

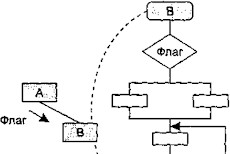


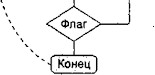
**Рис. 4.13.** Сцепление поданным

1. *Сцепление по образцу* (СЦ=3). В качестве параметров используются структуры данных (рис. 4.14).



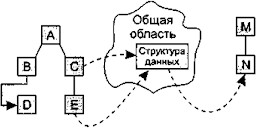
**Рис. 4.14.** Сцепление по образцу

1. *Сцепление по управлению* (СЦ=4). Модуль А явно управляет функционированием модуля В (с помощью флагов или переключателей), посылая ему управляющие данные (рис. 4.15).



**Рис. 4.15.** Сцепление по управлению

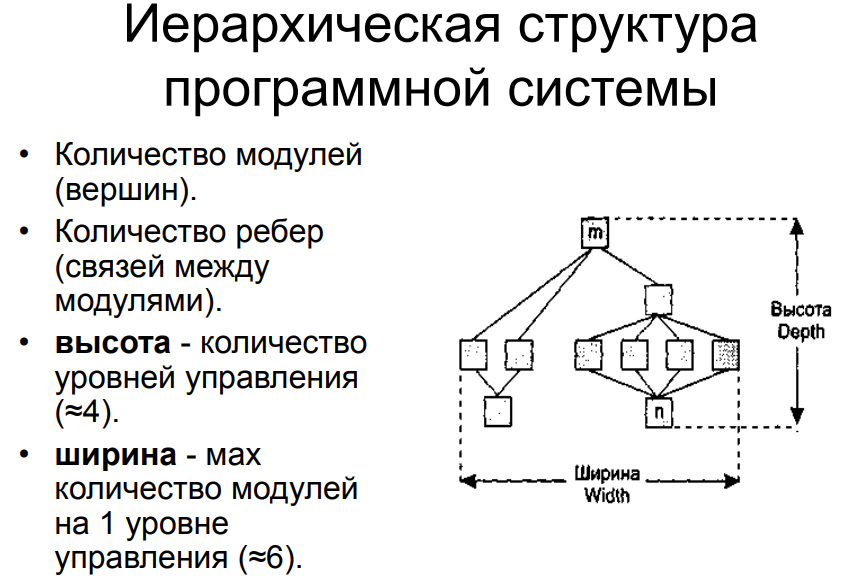
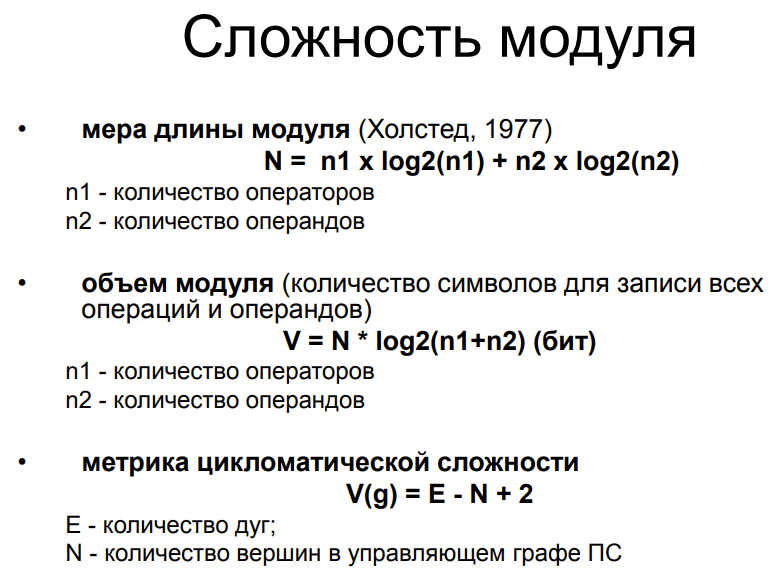
1. *Сцепление по внешним ссылкам* (СЦ=5). Модули А и В ссылаются на один и тот же глобальный элемент данных.
2. *Сцепление по общей области* (СЦ=7). Модули разделяют одну и ту же глобальную структуру данных (рис. 4.16).
3. *Сцепление по содержанию* (СЦ=9). Один модуль прямо ссылается на содержание другого модуля (не через его точку входа). Например, коды их команд перемежаются друг с другом (рис. 4.16).

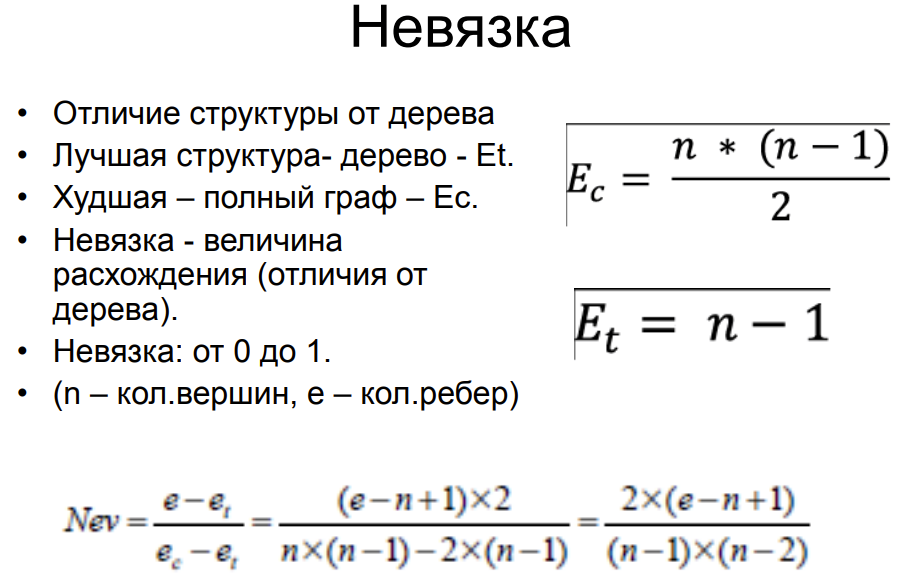
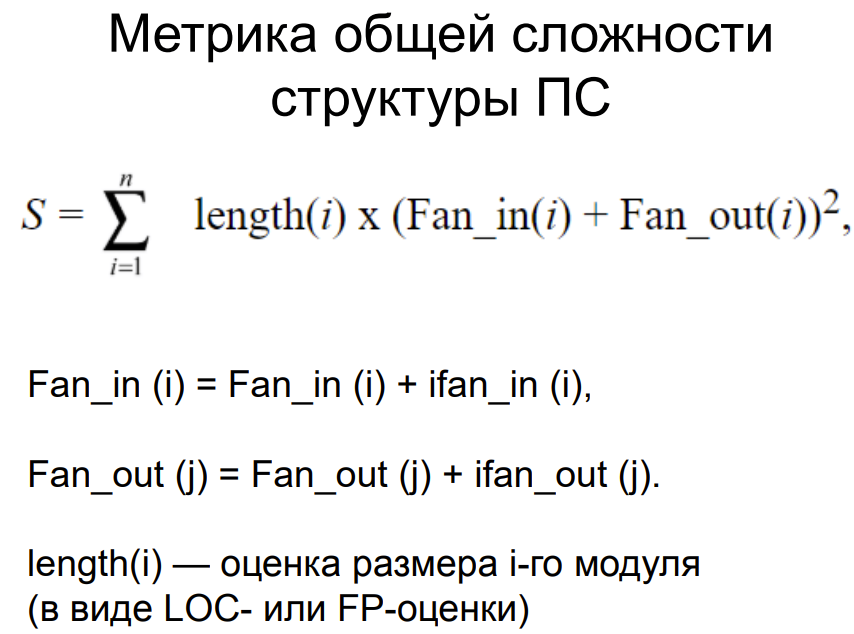


**Рис. 4.16.** Сцепление по общей области и содержанию

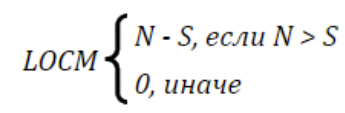
1. Оценка качества программных систем. Функциональные метрики программных систем. Сложность модуля. Сложность программной системы. Иерархическая структура. Метрики высоты, ширины, невязки. Привести определения и примеры.

Функциональны метрики см. 9.





1. Оценка качества программных систем. Метрики объектно- ориентированных систем: Чидамбера и Кемерера, Лоренца и Кидда, Абреу. Их назначение и методы вычислений. Отличие от функциональных метрик. Привести определения и примеры.

Метрики Чидамбера и Кемерера • Взвешенные методы на класс - WMC (Weighted Methods Per Class) – считать только методы класса, без унаследованных. – считать все методы, в т.ч. числе унаследованные. – промежуточный. Например, методы класса и его родителя. – WMC - относительная мера сложности классов. • Высота дерева наследования - DIT(Depth of Inheritance Tree) – Для отдельного класса - это длина максимального пути от данного класса до корневого класса в иерархии классов. – У нижнего уровня много методов => трудно предсказать поведение класса. Возрастает сложность проекта, многие методы могут выполняться многократно. Метрики Чидамбера и Кемерера • Количество детей - NOC (Number of children) – NOC = количество непосредственных наследников класса. – ↑NOC = ↓абстракции родительского класса (-> часть детей может быть использована неправильно). – ↑количества тестов для проверки ребенка. – ↑многократного использования. • DIT и NOC определяют форму и размер структуры классов. • Следует строить сбалансированные по высоте и ширине структуры наследования: обычно не выше, чем 7 ± 2 уровня, и не шире, чем 7 ± 2 ветви. Метрики Чидамбера и Кемерера • Сцепление между классами объектов - СВО (Coupling between object classes) – СВО = количество сцеплений класса (вызов метода или свойства другого класса) – ↑СВО = ↓повторное использование класса. – Высокое значение СВО усложняет модификацию и тестирование. Минимизация межобъектных сцеплений улучшает модульность и содействует инкапсуляции проекта • Отклик для класса - RFC (Response For a Class) – RFC = количество методов в множестве отклика. – Множество отклика - это множество методов класса и вызовов методов других классов из данного класса. – С ростом RFC увеличивается сложность класса. – Определяет временя тестирования. Метрики Чидамбера и Кемерера • Недостаток связности в методах - LСOM (Lack of Cohesion in Methods) – Определяет, насколько методы не связаны друг с другом через свойства (переменные). Если все методы обращаются к одинаковым свойствам, то LCOM = 0. – N - количество пар методов без общих экземплярных переменных; – S - количество пар методов с общими экземплярными переменными. – Если LCOM имеет высокое значение, то методы слабо связаны друг с другом через свойства. Это увеличивает сложность, в связи с чем возрастает вероятность ошибок при проектировании. 

Метрики Лоренца и Кидда • Коллекция метрик Лоренца и Кидда — результат практического, промышленного подхода к оценке ОО-проектов. • Размер класса CS (Class Size) – общее количество операций (вместе с приватными и наследуемыми экземплярными операциями), которые инкапсулируются внутри класса; – количество свойств (вместе с приватными и наследуемыми экземплярными свойствами), которые инкапсулируются классом. – Большие значения CS указывают, что класс имеет слишком много обязанностей. Они уменьшают возможность повторного использования класса, усложняют его реализацию и тестирование. – Рекомендуемое значение CS <= 20 методов. Метрики Лоренца и Кидда • Количество операций, переопределяемых подклассом, NOO (Number of Operations Overrided by a Subclass) – Большие значения NOO обычно указывают на проблемы проектирования, так же нарушается абстракция суперкласса, ослабляется иерархия классов, усложняет тестирование и модификацию программного обеспечения. – Рекомендуемое значение NOO <= 3 методов. • Количество операций, добавленных подклассом - NOA (Number of Operations Added by a Subclass) – С ростом NOA подкласс удаляется от абстракции суперкласса. – Обычно при увеличении высоты иерархии классов (увеличении DIT) должно уменьшаться значение NOA на нижних уровнях иерархии. – Для рекомендуемых значений CS = 20 и DIT = 6 рекомендуемое значение NOA <= 4 методов (для класса-листа). Метрики Лоренца и Кидда • Индекс специализации SI - (Specialization Index) SI = (NOO x уровень) /Mобщ, – где уровень — номер уровня в иерархии, на котором находится подкласс, Мобщ — общее количество методов класса. – Чем выше значение SI, тем больше вероятность того, что в иерархии классов есть классы, нарушающие абстракцию суперкласса. – Рекомендуемое значение SI <= 0,15. Метрики Лоренца и Кидда • Средний размер операции OSAVG (Average Operation Size) – количество строк программы. – Альтернативный вариант — «количество сообщений, посланных операцией». – Рост значения означает, что обязанности размещены в классе не очень удачно. – Рекомендуемое значение OSAVG <= 9. • Среднее количество параметров на операцию NPavg (Number of Parameters average) – Чем больше параметров у операции, тем сложнее сотрудничество между объектами. Поэтому значение NPavg должно быть как можно меньшим. – Рекомендуемое значение NPavg = 0,7. Метрики Лоренца и Кидда – Сложность операции ОС (Operation Complexity) • LOC- или FP-оценок, метрики цикломатической сложности, метрики Холстеда. • желательно уменьшать ОС. • Рекомендуемое значение ОС <= 65

Параметр Вес  
Вызовы функций API 5,0  
Присваивания 0,5   
Арифметические операции 2,0   
Сообщения с параметрами 3,0   
Вложенные выражения 0,5   
Параметры 0,3   
Простые вызовы 7,0   
Временные переменные 0,5   
Сообщения без параметров 1,0

Метрики Лоренца и Кидда – Количество описаний сценариев NSS (Number of Scenario Scripts) • Рекомендуется — не менее одного сценария на публичный протокол системы, отражающий основные функциональные требования к подсистеме. – Количество ключевых классов NKC (Number of Key Classes) • Ключевой класс прямо связан с проблемной областью. • 20-40% от общего количества классов. Оставшиеся классы реализуют общую инфраструктуру. • Рекомендуемое значение: если NKC < 0,2 от общего количества классов системы, следует углубить исследование проблемной области. – Количество подсистем NSUB (NumberofSUBsystem) • размещение ресурсов, планирование (с акцентом на параллельную разработку), общие затраты на интеграцию. • Рекомендуемое значение: NSUB > 3. Проектные метрики Лоренца и Кидда – NSS, NKC, NSUB формируют метрический базис фирмы, в который также включаются метрические значения по классами и операциям. – Эти исторические данные могут использоваться для вычисления метрик производительности (среднее количество классов на разработчика или среднее количество методов на человекомесяц). – Совместное применение метрик позволяет оценивать затраты, продолжительность, персонал и другие характеристики текущего проекта.

Метрики Фернандо Абреу • MOOD – Metrics of OO Design • Покрытие базовых механизмов объектноориентированной парадигмы, таких как инкапсуляция, наследование, полиморфизм, посылка сообщений; • Формальное определение метрик, позволяющее избежать субъективности измерения; • Независимость от размера оцениваемого программного продукта; • Независимость от языка программирования, на котором написан оцениваемый продукт. Метрики Фернандо Абреу • Фактор закрытости метода (МНF) Method Hiding Factor – Мv (Сi) — количество видимых методов в классе Сi (интерфейс класса); – Мh (Сi) — количество скрытых методов в классе Сi (реализация класса); – Мd (Сi) = Мv (Сi) + Мh (Сi) — общее количество методов, определенных в классе С, (унаследованные методы не учитываются). – C увеличением МНF уменьшаются плотность дефектов в системе и затраты на их устранение. – Возрастание значения МНF и увеличение качества класса.

,

Фактор закрытости свойства (МНA) Attribute Hiding Factor – Аv (Сi) — количество видимых свойств в классе Сi (интерфейс класса); – Ah(Ci) — количество скрытых свойств в классе Сi (реализация класса); – Ad(Ci) = Аv (Сi) + Ah(Ci) — общее количество свойств, определенных в классе Сi (унаследованные свойства не учитываются) – В идеальном случае все свойства должны быть скрыты и доступны только для методов соответствующего класса – (AHF = 100%).

,

Фактор Наследования метода (МНF) Method Inheritance Factor – Mi (Сi ) — количество унаследованных и не переопределенных методов в классе Сi; – M0(Сi ) — количество унаследованных и переопределенных методов в классе Сi; – Mn(Сi ) — количество новых (не унаследованных и переопределенных) методов в классе Сi; – Md(Сi )= Mn(Сi ) + M0(Сi ) — количество методов, определенных в классе Сi; – Ma(Сi )= Md(Сi)+ Mi(Сi) — общее количество методов, доступных в классе Сi. – MIF = 0 указывает, что в системе нет эффективного наследования.

.

Фактор Наследования свойства (AНF) Attribute Inheritance Factor – Аi (Сi) — количество унаследованных и не переопределенных свойств в классе Сi; – А0(Сi) — количество унаследованных и переопределенных свойств в классе Сi; – An(Ci) — количество новых (не унаследованных и переопределенных) свойств в классе Сi; – Аd(Сi) = An(Ci) + А0(Сi) — количество свойств, определенных в классе Сi; – Аa(Сi) = Аd(Сi)+ Аi (Сi) — общее количество свойств, доступных в классе Сi.

.

фактор полиморфизма (POF) Polymorphism Factor – M0(Сi) — количество унаследованных и переопределенных методов в классе Сi; – Mn(Сi) — количество новых (не унаследованных и переопределенных) методов в классе Сi; – DC(Сi) — количество потомков класса Сi; – Md(Сi) = Mn(Сi) + M0(Сi) — количество методов, определенных в классе Сi. – Умеренное использование полиморфизма уменьшает как плотность дефектов, так и затраты на доработку. Однако при POF > 10% возможен обратный эффект.

.

фактор сцепления (СОF) Coupling Factor. – наличие между классами отношения «клиент-поставщик» (client-supplier). – класс-клиент содержит по меньшей мере одну не унаследованную ссылку на свойство или метод класса-поставщика. – Числитель COF фиксирует реальное количество сцеплений, не относящихся к наследованию. – С увеличением сцепления классов плотности дефектов и затрат на доработку также возрастают. – Сцепления отрицательно влияют на качество ПО. – сцепление увеличивает сложность, уменьшает инкапсуляцию и возможности повторного использования, затрудняет понимание и усложняет сопровождение ПО.

.

1. Унифицированный процесс разработки ПО (RUP). Перечень, последовательность и содержимое этапов разработки. Рабочие процессы и их модели. Примеры итераций, задач, макетов и моделей для RUP.

см. начало вопроса 19

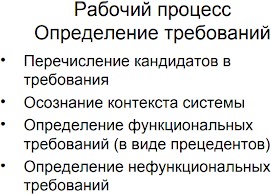
1. Унифицированный процесс RUP. Определение архитектурно- значимых прецедентов. Модель прецедентов, модель анализа (пакеты, классы и кооперации), модель проектирования (диаграммы развертывания, подсистемы, их интерфейсы и активные классы) и модель реализации (подсистемы и компоненты). Трассировка между элементами моделей. Привести определения и примеры моделей.

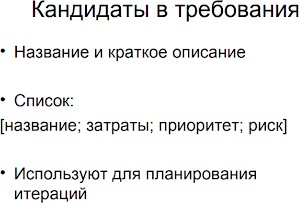
RUP - см. вопрос 19

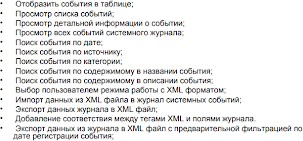
Ответом на этот вопрос являются ответы на вопросы 15-21, так что хз что тут писать еще

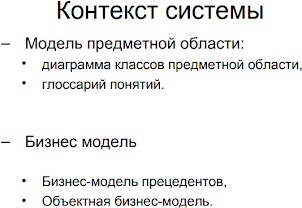
1. Унифицированный процесс RUP. Рабочий процесс Определение требований. Кандидаты в требования. Контекст системы. Функциональные и нефункциональные требования. Модели прецедентов (требований) и предметной области. Нахождение актеров и прецедентов. Определение приоритетов и детализация прецедентов. Создание прототипа пользовательского интерфейса. Структурирование модели прецедентов. Привести определения и примеры моделей.

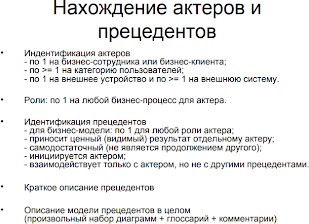
RUP - см. вопрос 19

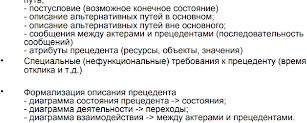








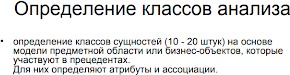
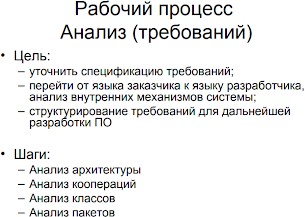


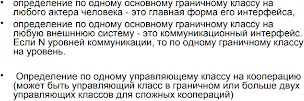




1. Унифицированный процесс RUP. Рабочий процесс Анализ. Модель анализа (пакеты, их зависимости и состав, диаграммы коопераций анализа). Анализ архитектуры, прецедентов, классов и пакетов (деятельность и модели). Привести определения и примеры моделей.

RUP - см. вопрос 19





1. Унифицированный процесс RUP. Рабочий процесс Проектирование. Модель проектирования (подсистемы всех уровней, их состав, интерфейсы и зависимости, диаграмма развертывания и активные классы, примеры диаграмм взаимодействия подсистем). Проектирование архитектуры, прецедентов, классов и подсистем (деятельность и модели). Привести определения и примеры моделей.

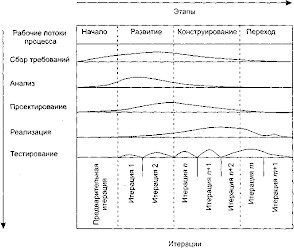
**Унифицированный процесс RUP**

Унифицированный процесс разработки ПО RUP (Rational Unified Process, англ.) — это методология создания программных систем, являющаяся результатом объединения различных подходов к построению программ. RUP вобрал в себя преимущества спиральной и итерационной методологий разработки ПО. Согласно спиральной методологии, требования к программе формируются и уточняются постепенно, в процессе создания программы. В соответствии с инкрементной методологией происходит последовательное приращение функциональности.

* использует uml
* Унифицирует:
  + подходы к разработке;
  + многие методологии;
* Поддерживается Rational / IBM;
* Необходим для:
  + Больших и сложных систем;
  + Больших команд разработчиков с различным опытом;
  + Автоматизации разработки / согласования / модификации / сохранения системы.

RUP имеет следующие особенности:

* управляется прецедентами, т. е. на всех этапах разработки в центре внимания находится прецедент. Прецедент — это прикладная задача, которая подлежит автоматизации при написании программы, например задача добавления сведений о персоне или задача поиска по каталогу;
* ориентирован на архитектуру программы — это означает, что на начальном этапе разработки определяется архитектура будущей системы. На последующих этапах разработки архитектура влияет на принятие проектных решений;
* является итерационно-инкрементным, то есть при разработке происходит декомпозиция всей системы на мини-проекты, называемые итерациями. В результате выполнения очередной итерации появляется инкремент — функциональное приращение.



* Унифицированный процесс состоит из четырех этапов разработки: сбор требований, проектирование, построение, внедрение
* Каждый этап выполняется как серия итераций, каждая из которых обеспечивает приращение функциональных возможностей системы.
* Итерация — это полный цикл разработки фрагмента ПО, в результате которого формируется промежуточная версия, реализующая некоторый функционал. В ходе разработки происходит постепенное усложнение создаваемого ПО, наращивание его функциональных возможностей.
* Каждая итерация состоит из последовательного выполнения пяти рабочих процессов.

**Рабочий процесс Проектирование.**

**. проектирование** — создание статического и динамического представлений системы для выполнения исходных требований (построение проектных моделей);

* Проектирование архитектуры
  + Определение узлов и сетевых конфигураций
  + Определение подсистем и их интерфейсов
    - – Определение прикладных подсистем
    - – Определение сервисных подсистем
  + Определение интерфейсов подсистем
  + Определение архитектурно значимых классов проектирования
  + Определение обобщенных механизмов проектирование
* Проектирование прецедентов Для кооперации:
  + Определение участвующих в кооперации классов проектирования
  + Описание взаимодействия объектов проектирования
  + Определение участвующих подсистем и их интерфейсов
  + Описание взаимодействия подсистем
  + Определение специальных требований к кооперациям
* Проектирование классов

Трассировка классов анализа в классы проектирования.

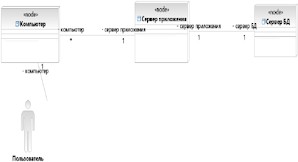
Граничные – в классы форм и их компонентов (производные от стандартных) Сущности – в таблицы БД, файлы, объекты в ОП

Управляющие – в классы реализации бизнес-логики

* Проектирование подсистем Подробнее про проектирование архитектуры.

**Определение узлов и сетевых конфигураций**

*Узел —* физический элемент, который существует в период работы системы и представляет ресурс, обычно имеющий память и возможности обработки. В узле размещается набор компонентов, который может перемещаться от узла к узлу.

* перечень узлов и их конфигураций
  + конфигурация сети (линии связи между узлами; протоколы; характеристики передачи; скорость; качество;...)
* дополнительные требования
* конфигурация для тестирования / моделирования ПО => диаграмма развертывания

**Определение подсистем и их интерфейсов**

* Определение прикладных подсистем

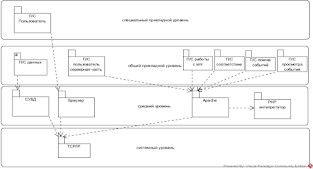
– пакет анализа трассируется в подсистему проектирования

* декомпозиции при совместном использованиеи
* разнесение по узлам
  + Определение подсистем среднего уровня и уровня системного ПО
* ОС, СУБД, коммуникация, распределение, транзакции – Интеграция покупаемых /

создаваемых компонентов

* Оценка эффективности и пригодности
  + Определение зависимостей между подсистемами

=> диаграмма уровней подсистем



Определение интерфейсов подсистем:

* + перечень операций, доступных извне;
  + на основе классов в пакетах анализа (к которым обращаются извне);
  + на основе зависимостей подсистем;
  + начать с подсистем верхнего уровня;
  + Для среднего и нижнего уровня могут быть существующие заранее (существующее ПО)

**Активный класс** *—* класс, чьи объекты имеют один или несколько процессов (или потоков) и поэтому могут инициировать управляющую деятельность. Активный класс похож на обычный класс за исключением того, что его объекты действуют одновременно с объектами других классов.

Своя нить управления, процесс.

* *один или более на узел;*
* один на обмен между узлами;
* для повышения производительности (ответ без задержек);
* для запуска / остановки / конфигурации / снятие блокировки / восстановление и т.д.

активный класс -> исполняемый файл; подсистема -> компонент

**Проектирование классов**

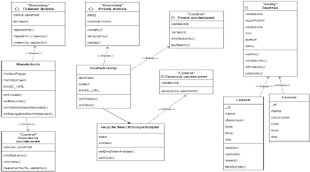
* + - операции (сигратуры в терминах языка);
* атрибуты (стандартные типы);
* отношения (ассоциации, агрегации, обобщение); - методы реализации (алгоритмы)
* состояния
  + - зависимость от обобщенных механизмов проектирования
* требования к реализации
* правильность реализации всех интерфейсов трассировка классов сущностей



Трассировка управляющих классов



Трассировка граничного:

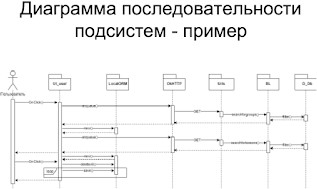


**Проектирование подсистем**









1. Унифицированный процесс RUP. Рабочий процесс Реализация. Модели реализации (компоненты, интерфейсы и подсистемы реализации, стереотипы и зависимости файлов). Реализация архитектуры, подсистем, классов (деятельность и модели). Сборка системы. Тестирование модулей. Привести определения и примеры моделей.

*Унифицированный процесс RUP — см. вопрос 19*

***Рабочий процесс Реализация***

***реализация*** *— процесс генерации программного кода (включая тестирование и отладку модулей);*

Во время реализации создается программный продукт, готовый к использованию. При этом модели, построенные при проектировании, преобразуются по следующей схеме:

* подсистема проектирования трассируется в подсистему реализации, содержащую исполняемые файлы, исходные тексты и т. д.;
* сервисная подсистема – в сервисную подсистему;
* класс проектирования – в один или более (это зависит от языка программирования) файлов компонентов, содержащих исходный текст программы;
* активный класс - в исполняемый компонент;
* один или более проектов кооперации – в билд (сборку);
* модель развертывания и конфигурация сети – в распределенную систему и

развертывание исполняемых компонентов.

Процесс реализации достаточно тривиален, и существенная его часть успешно автоматизирована.

В результате реализации создаются следующие артефакты:

* модель реализации – содержит систему реализации, содержащую множество

подсистем, компонентов и интерфейсов реализации;

* описание архитектуры – содержит перечень наиболее важных компонентов

системы и описывает их размещения по узлам;

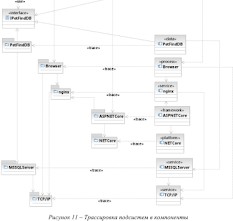
* модель размещения – содержит описание сетевой конфигурации;
* план сборки – содержит перечень прецедентов, которые необходимо реализовать на очередном шаге разработки, и перечень компонентов, которые будут при этом созданы;
* компоненты – физические компоненты программной системы, которые обеспечивают ее работу, например, исполняемые программы, базы данных, файлы настроек и т. д.;
* подсистемы реализации – контейнеры для компонентов, чья конкретная реализация зависит от используемых технологий;
* интерфейсы - содержат перечень доступных извне функций компонентов.

*Процесс реализации состоит из выполнения следующих действий:*

* *реализация архитектуры;*
* *сборка системы;*
* *реализация подсистем;*
* *реализация классов;*
* *тестирование*
* *модулей.*

***Подсистема реализации*** *— это контейнер физических компонентов ПО, конкретная структура которого зависит от языка и технологии разработки. Необходимо обеспечить правильную трассировку подсистемы проектирования в подсистему реализации.*





**Сборка системы**

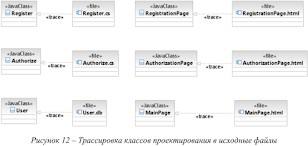
По методологии унифицированного процесса разработка ПО выполняется итерационно. Каждая итерация дает функциональное приращение программного продукта. Итерация состоит из нескольких билдов.

Билд – это реализация нескольких прикладных функций системы, обычно одного или более прецедентов, а также дополнительных требований к ним. В результате создания билда получается несколько компонентов и/или подсистем, которые реализуют этот функционал.

До начала итерации определяется набор и состав ее билдов. Поскольку разработка ведется несколькими командами разработчиков, то в пределах итерации реализация билдов может происходить последовательно и/или параллельно.

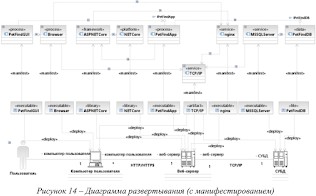
***Реализация класса*** *– это перевод проекта класса в исходный код программы.*

* *описание файловых компонентов;*
* *генерация кода;*
* *реализация операций;*
* *создание компонентов.*





*Также составляется диаграмма размещения с указанием компонентов*



**Тестирование модулей**

**Тестирование** – это проверка правильности работы ПО. Созданные программы запускаются с целью выявить ошибки в их работе. Тестирование модулей позволяет проверить корректность реализации компонентов перед их интегрированием в систему.

Выделяют два вида тестирования модулей:

* тестирование спецификации – «черный ящик»;
* тестирование структуры – «белый ящик».

Тестирование спецификации позволяет на основе описания требований к модулю (прецедентов, которые он реализует) проверить выполнение прикладных функций. Проверяются функциональные и специальные требования. Здесь выявляются ошибки в составлении алгоритмов, передачи входных или выходных данных, инициализации внутренних структур программы. Используются методы функционального тестирования, например методы граничных значений или разбиение на классы эквивалентности.

Тестирование структуры позволяет проверить корректность перевода проекта в программный код, отсутствие синтаксических и семантических ошибок в программе. Здесь выявляются ошибки кодирования алгоритмов и использования

переменных, составления условий или задания циклов. Используются методы структурного тестирования, например методы ветвей и границ или проверки циклов.

1. Унифицированный процесс RUP. Рабочий процесс Тестирование (деятельность и модели). Процедуры, методы и варианты тестирования. Привести определения и примеры.

*Унифицированный процесс RUP — см. вопрос 19*

***Рабочий процесс Тестирование***

***тестирование*** *— проверка работоспособности системы в целом.*

Тестирование — это одна из техник контроля качества, включающая в себя активности по планированию работ (Test Management), проектированию тестов (Test Design), выполнению тестирования (Test Execution) и анализу полученных результатов (Test Analysis).

Процесс тестирования состоит из следующих этапов:

1. Анализ продукта
2. Работа с требованиями
3. Разработка стратегии тестирования и планирование процедур контроля качества
4. Создание тестовой документации
5. Тестирование прототипа
6. Основное тестирование
7. Стабилизация
8. Эксплуатация

## Методы тестирования

***Эквивалентное Разделение (Equivalence Partitioning — EP).*** Как пример, у вас есть диапазон допустимых значений от 1 до 10, вы должны выбрать одно верное значение внутри интервала, скажем, 5, и одно неверное значение вне интервала — 0.

***Анализ Граничных Значений (Boundary Value Analysis — BVA).*** Если взять пример выше, в качестве значений для позитивного тестирования выберем минимальную и максимальную границы (1 и 10), и значения больше и меньше границ (0 и 11). Анализ Граничный значений может быть применен к полям, записям, файлам, или к любого рода сущностям, имеющим ограничения.

***Причина / Следствие (Cause/Effect — CE).*** Это, как правило, ввод комбинаций условий (причин), для получения ответа от системы (Следствие). Например, вы проверяете возможность добавлять клиента, используя определенную экранную форму. Для этого вам необходимо будет ввести несколько полей, таких как «Имя»,

«Адрес», «Номер Телефона» а затем, нажать кнопку «Добавить» — это «Причина». После нажатия кнопки «Добавить», система добавляет клиента в базу данных и показывает его номер на экране — это «Следствие».

***Предугадывание ошибки (Error Guessing — EG).*** Это когда тестировщик использует свои знания системы и способность к интерпретации спецификации на предмет того, чтобы «предугадать» при каких входных условиях система может выдать ошибку. Например, спецификация говорит: «пользователь должен ввести код». Тестировщик будет думать: «Что, если я не введу код?», «Что, если я введу неправильный код?», и так далее.

***Исчерпывающее тестирование (Exhaustive Testing — ET)*** — это крайний случай. В пределах этой техники вы должны проверить все возможные комбинации входных значений, и в принципе, это должно найти все проблемы. На практике применение этого метода не представляется возможным, из-за огромного количества входных значений.

***Попарное тестирование (Pairwise Testing)*** — это техника формирования наборов тестовых данных. Сформулировать суть можно, например, вот так: формирование таких наборов данных, в которых каждое тестируемое значение каждого из проверяемых параметров хотя бы единожды сочетается с каждым тестируемым значением всех остальных проверяемых параметров.

***Матрица соответствия требований (Traceability matrix)*** — это двумерная таблица, содержащая соответствие функциональных требований (functional requirements) продукта и подготовленных тестовых сценариев (test cases). В заголовках колонок таблицы расположены требования, а в заголовках строк — тестовые сценарии. На пересечении — отметка, означающая, что требование текущей колонки покрыто тестовым сценарием текущей строки. Матрица соответствия требований используется QA-инженерами для валидации покрытия продукта тестами. МСТ является неотъемлемой частью тест-плана.

***Тестовый сценарий, тестовый вариант, тест (Test Case)*** — это артефакт, содержащий исходные данные и ожидаемый результат при тестировании. Он также

описывает совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части.

Каждый тестовый сценарий должен иметь 3 части:

* Предусловие - PreConditions - Список действий, которые приводят систему к состоянию пригодному для проведения основной проверки. Либо список условий, выполнение которых говорит о том, что система находится в пригодном для проведения основного теста состояния.
* Описание тестового сценария - Test Case Description - Список действий, переводящих систему из одного состояния в другое, для получения результата, на основании которого можно сделать вывод о удовлетворении реализации, поставленным требованиям.
* Постусловия - PostConditions - Список действий, переводящих систему в первоначальное состояние (состояние до проведения теста — initial state)

Виды Тестовых Сценариев. Тесты разделяются по ожидаемому результату на положительные и отрицательные:

* Положительный тест использует только корректные данные и проверяет, что приложение правильно выполнило вызываемую функцию.
* Отрицательный тест оперирует как корректными, так и некорректными данными (минимум 1 некорректный параметр) и ставит целью проверку исключительных ситуаций (срабатывание валидаторов), а также проверяет, что вызываемая приложением функция не выполняется при срабатывании валидатора.

***Чек-лист (check list)*** — это документ, описывающий что должно быть протестировано. При этом он может быть абсолютно разного уровня детализации. На сколько детальным он будет зависит от требований к отчетности, уровня знания продукта сотрудниками и сложности продукта. Как правило, список содержит только действия (шаги), без ожидаемого результата. Он менее формализован, чем тестовый сценарий. Его уместно использовать тогда, когда тестовые сценарии будут избыточны. Также чек-лист ассоциируются с гибкими подходами в тестировании.

***Дефект (баг)*** – это несоответствие фактического результата выполнения программы ожидаемому результату. Дефекты обнаруживаются на этапе тестирования программного обеспечения (ПО), когда тестировщик проводит сравнение полученных результатов работы программы (компонента или дизайна) с ожидаемым результатом, описанным в спецификации требований.

## Виды тестирования

***Модульное тестирование (Unit Testing).*** Компонентное (модульное) тестирование проверяет функциональность и ищет ошибки в частях приложения, которые доступны и могут быть протестированы по-отдельности (модули программ, объекты, классы, функции и т.д.).

***Интеграционное тестирование (Integration Testing).*** Проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.

***Системное тестирование (System Testing).*** Основной задачей системного тестирования является проверка как функциональных, так и не функциональных требований в системе в целом. При этом выявляются дефекты, такие как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т.д.

***Операционное тестирование (Release Testing).*** Даже если система удовлетворяет всем требованиям, важно убедиться в том, что она удовлетворяет нуждам пользователя и выполняет свою роль в среде своей эксплуатации, как это было определено в бизнес модели системы. Следует учесть, что и бизнес-модель может содержать ошибки. Поэтому так важно провести операционное тестирование как финальный шаг валидации. Кроме этого, тестирование в среде эксплуатации позволяет выявить и нефункциональные проблемы, такие как: конфликт с другими системами, смежными в области бизнеса или в программных и электронных окружениях; недостаточная производительность системы в среде эксплуатации и др. Очевидно, что нахождение подобных вещей на стадии внедрения — критичная и дорогостоящая проблема. Поэтому так важно проведение не только верификации, но и валидации, с самых ранних этапов разработки ПО.

***Приемочное тестирование (Acceptance Testing).*** Формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью: определения удовлетворяет ли система приемочным критериям и вынесения решения заказчиком или другим уполномоченным лицом принимается приложение или нет.

## Типы тестирования

Функциональные виды тестирования.

* Функциональное тестирование (Functional testing). Проверка корректности и

полноты выполнемых системой функций в соответствии с требованиеми к ней. Рассматривает заранее указанное функцинное поведение и основывается на анализе спецификаций функциональности компонента или системы в целом.

* Тестирование пользовательского интерфейса (GUI Testing). Функциональная проверка интерфейса на соответствие требованиям — размер, шрифт, цвет, consistent behavior.
* Тестирование безопасности (Security and Access Control Testing). Это стратегия тестирования, используемая для проверки безопасности системы, а также для анализа рисков, связанных с обеспечением целостного подхода к защите приложения, атак хакеров, вирусов, несанкционированного доступа к конфиденциальным данным.
* Тестирование взаимодействия (Interoperability Testing). Это тестирование, проверяющее способность приложения взаимодействовать с одним и более компонентами или системами и включающее в себя тестирование совместимости (compatibility testing), и интеграционное тестирование.

Нефункциональные виды тестирования.

* Все виды тестирования производительности:
  + нагрузочное тестирование (Performance and Load Testing). Это автоматизированное тестирование, имитирующее работу определенного количества бизнес-пользователей на каком-либо общем (разделяемом ими) ресурсе.
  + стрессовое тестирование (Stress Testing). Оно позволяет проверить насколько приложение и система в целом работоспособны в условиях стресса и также оценить способность системы к регенерации, т.е. к возвращению к нормальному состоянию после прекращения воздействия стресса. Стрессом в данном контексте может быть повышение интенсивности выполнения операций до очень высоких значений или аварийное изменение конфигурации сервера.
  + тестирование стабильности или надежности (Stability / Reliability Testing). Задачей этого вида тестирования является проверка работоспособности приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки.
  + объемное тестирование (Volume Testing) Задачей является получение оценки производительности при увеличении объемов данных в базе данных приложения.
* Тестирование установки (Installation testing) Оно направленно на проверку успешной инсталляции и настройки, а также обновления или удаления программного обеспечения.
* Тестирование удобства пользования (Usability Testing) Это метод тестирования, направленный на установление степени удобства использования, обучаемости, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий.
* Тестирование на отказ и восстановление (Failover and Recovery Testing) Оно проверяет тестируемый продукт с точки зрения способности противостоять и успешно восстанавливаться после возможных сбоев, возникших в связи с ошибками программного обеспечения, отказами оборудования или проблемами связи (например, отказ сети). Целью данного вида тестирования является проверка систем восстановления (или дублирующих основной функционал систем), которые, в случае возникновения сбоев, обеспечат сохранность и целостность данных тестируемого продукта.
* Конфигурационное тестирование (Configuration Testing) Специальный вид тестирования, направленный на проверку работы программного обеспечения при различных конфигурациях системы (заявленных платформах, поддерживаемых драйверах, при различных конфигурациях компьютеров и т.д.).

Связанные с изменениями виды тестирования

* Дымовое тестирование (Smoke Testing) Оно рассматривается как короткий цикл тестов, выполняемый для подтверждения того, что после сборки кода (нового или исправленного) устанавливаемое приложение стартует и выполняет основные функции.
* Регрессионное тестирование (Regression Testing) Это вид тестирования направленный на проверку изменений, сделанных в приложении или окружающей среде (устранение дефекта, слияние кода, миграция на другую операционную систему, базу данных, веб сервер или сервер приложения), для подтверждения того факта, что существующая ранее функциональность работает как и прежде. Регрессионными могут быть как функциональные, так и нефункциональные тесты.
* Повторное тестирование (Re-testing) Тестирование, во время которого исполняются тестовые сценарии, выявившие ошибки во время последнего запуска, для подтверждения успешности исправления этих ошибок.
* Тестирование сборки (Build Verification Test). Тестирование, направленное на определение соответствия, выпущенной версии, критериям качества для начала тестирования. По своим целям является аналогом Дымового Тестирования, направленного на приемку новой версии в дальнейшее тестирование или эксплуатацию. Вглубь оно может проникать дальше, в зависимости от требований к качеству выпущенной версии
* Санитарное тестирование или проверка согласованности/исправности (Sanity Testing) Узконаправленное тестирование достаточное для доказательства того, что конкретная функция работает согласно заявленным в спецификации требованиям. Является подмножеством регрессионного тестирования. Используется для определения работоспособности определенной части приложения после изменений, произведенных в ней или окружающей среде. Обычно выполняется вручную

1. Проектирование программных систем. Модели архитектуры: хранилище данных, клиент-сервер, трехуровневая модель, многоуровневая модель, каналы и фильтры, компонентная и каркасы. Привести определения и примеры.

Проектирование программных систем

**Проектирование** - создание статического и динамического представления системы для выполнения требований;

* На основе требований и моделей определить:
  + Структуры данных (на основе информационной модели)
  + Архитектуру системы (компоненты и их связи)
  + Процедурная разработка (содержание компонентов как последовательности действий)

Методологии проектирования:

* **Структурная** (функциональная) декомпозиция:

Структура ПО как иерархия функций и передача информации по иерархии (вызов) —

низходящее, восходящее, метод расширения ядра

* **Объектная** декомпозиция:

Структура объектов и связи между ними;

поведение ПО как обмен сообщений между объектами (синхронные и асинхронные).



Этапы предварительного проектирования

* + Структурирование системы
    - Выделение подсистем (подсистема как независимый компонент)
    - Определение взаимодействия подсистем
  + Моделирование управления
    - Определение модели связей управления между частями системы
  + Декомпозиция подсистем на модули
    - По любой подсистеме – декомпозиция на модули
    - Определение типов модулей и межмодульные соединения



* хранилище данных,
  + Общие данные
  + п/с работают с общим хранилищем



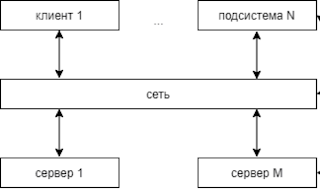




* клиент-сервер,

Клиент

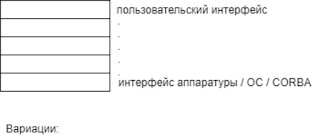
* + выдает запросы Сервер
  + ждет запроса;
  + предоставляет сервис или данные или ресурсы Сеть
  + протокол взаимодействия (например, TCP|IP)

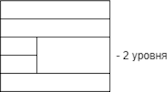


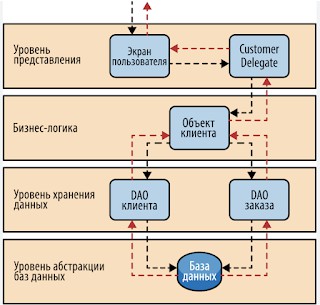
* трехуровневая модель,
  + упрощение модификации уровней
  + возрастает оптимизация системы за счет разделения функций БД и прикладных функций



* многоуровневая модель,
  + Текущий слой использует интерфейс нижнего слоя
  + Нельзя использовать возможности высшего уровня
* Преимущество:
  + можно заменить уровень без изменения остальных (ОС; язык реализации)
* Не определено:
  + Возможность использовать интерфейсы «вниз через уровень»
  + Возможность использовать интерфейсы «соседнего блока»







Архитектура не подразумевает какое-то обязательное количество уровней — их может быть три, четыре, пять и больше. Чаще всего используют трехзвенные системы: с уровнем представления (клиентом), уровнем логики и уровнем данных.

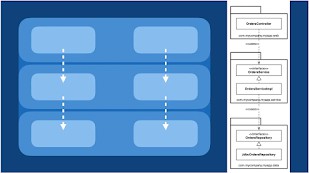
В логически разделённых на слои архитектурах информационных систем наиболее часто встречаются следующие четыре слоя:

* **Слой представления** (слой UI, UIL, пользовательский интерфе́йс, уровень представления в

многоуровневой архитектуре)

* **Слой приложения** (сервисный слой, сервисный уровень или GRASP уровень управления)
* **Слой бизнес-логики** (слой предметной области, BLL, доменный слой)
* **Слой доступа к данным** (слой хранения данных, DAL, слой инфраструктуры; логирование, сетевые взаимодействия и другие сервисы, требующиеся для поддержания конкретного слоя бизнес-логики)

Каждый уровень этой архитектуры выполняет строго ограниченный набор функций (которые не повторяются от слоя к слою) и не знает о том, как устроены остальные уровни. Поэтому «содержимое» уровней можно изменять без риска глобальных конфликтов между слоями.

Одним из самых известных примеров этого паттерна может служить сетевая модель OSI!

* каналы и фильтры,



Часто в системах необходимо преобразовывать потоки дискретных элементов данных от ввода к выводу. На практике многие типы преобразований повторяются неоднократно, поэтому желательно сделать из них независимые, повторно используемые компоненты.

Каждый поток обработки данных – это серия чередующихся фильтров и каналов, начинающаяся источником данных и заканчивающаяся их потребителем. Каналы обеспечивают передачу данных и синхронизацию. Фильтр же принимает на вход данные и обрабатывает их, трансформируя в некое иное представление, а затем передает дальше.

Фильтры можно легко заменять, использовать повторно, переставлять местами, что дает возможность реализовывать множество функций на основе ограниченного набора компонентов. Более того, активные фильтры могут работать параллельно, что приводит к значительному повышению производительности на многопроцессорных системах. Однако есть и недостатки, например, фильтры зачастую тратят больше времени на преобразование входных данных, чем на их обработку.

В качестве примера использования этой архитектуры может служить оболочка UNIX Shell. Другим примером может стать архитектура компилятора, если рассматривать её как последовательность фильтров: лексера, парсера, семантического анализатора и генератора кода.

Не лучший выбор для интерактивных систем, поскольку такая архитектура ориентирована на преобразование данных.

(Архитектура каналов и фильтров применяется в самых разных приложениях, особенно при решении задач, обеспечивающих простую одностороннюю обработку — например, инструменты EDI (электронный обмен данными), ETL (извлечение, преобразование и загрузка).

Пример — компиляторы: последовательно расположенные фильтры выполняют лексический, синтаксический, семантический анализ и создание кода.

* компонентная

Объектно-ориентированный подход (COM / CORBA / JBeans)



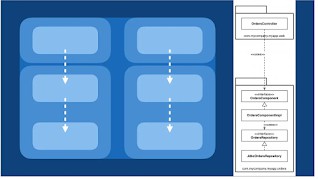
Основное внимание в этом случае уделяется разложению дизайна на отдельные функциональные или логические компоненты, предоставляющие четко определенные интерфейсы, содержащие методы, события и свойства. В данном случае обеспечивается более высокий уровень абстракции, чем при объектно-ориентированной разработке, и не происходит концентрации внимания на таких вопросах, как протоколы связи или общее состояние.

Основная задача архитектуры на основе **компонентов** заключается в обеспечении возможности

**повторного использования компонентов** . Компонент инкапсулирует функциональность и поведение программного элемента в повторно используемую и самораскрывающуюся двоичную единицу. Существует множество стандартных компонентов, таких как COM / DCOM, JavaBean, EJB, CORBA, .NET, веб-сервисы и сеточные сервисы. Эти технологии широко используются в проектировании приложений GUI для локальных компьютеров, таких как графические компоненты JavaBean, компоненты MS ActiveX и компоненты COM, которые можно повторно использовать простым перетаскиванием.

Вместо многоуровневого подхода, горизонтальных срезов, мы разделяем приложение по вертикали на модульные компоненты, так же, как мы сделали с архитектурой на основе функций.

Компонент в этом контексте-это группа связанных функций, которые находятся за красивым и чистым интерфейсом.



* каркасы (framework)
  + Расширяемая библиотека кода (группа связных классов) для эффективного проектного решения в конкретной области
    - Часть работы сделана
    - Есть пустые поля для расширения
    - Различные архитектурные модели
  + Библиотеки:
    - вызов существующих компонентов в своем потоке управления
  + Каркас:
    - имеет структуру и поток управления;
  + – может вызывать новые функции

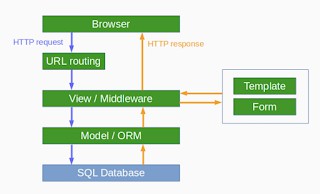
Общая архитектура, представляющая собой некий расширяемый шаблон для приложений в определенной предметной области.

Каркас реализует архитектурный шаблон, зачастую MVC.

В отличие от динамических библиотек, которые предоставляют собой набор ограниченных функций, на фреймворке выстраивается архитектура. Он определяет связь между компонентами.

Фреймворк обычно состоит из множества библиотек, которые он использует внутри себя, и реализует с помощью них структуру/каркас проекта. Также он предоставляет свои библиотеки для решения распространенных задач.

Архитектура Django:



1. Проектирование программных систем. Структурирование системы. Определение моделей архитектуры, управления и интерфейсов взаимодействия. Привести определения и примеры.

Проектирование программных систем. Определение моделей архитектуры, управления

* см. вопросы 22,24

Структурирование системы (этап предварительного проектирования)

* + Определение/ разработка модели архитектуры
  + Влияет на
    - производительность и характеристики системы
  + Определяет:
    - Компоненты
    - Связи компонентов
    - Характер интерфейсов системы;
    - роли и ответственности подсистем Виды интерфейсов
    - Точки соединения и границы компонентов
    - Внешний вид и доступ
  + **API -** интерфейсы прикладного программирования;
    - совокупность функций в физическом компоненте

Для замены компонента реализовать все функции и заново собрать

* + **Иерархия классов**
    - интерфейс сохраняется в производных классах (реализация абстрактных классов)
  + **Компонентные технологии** (COM/CORBA)
    - Реализация выбирается при выполнении
    - IDL – язык определения интерфейса (абстрактный)
    - Реализация на любом языке
    - Необходима поддержка со стороны промежуточного ПО или ОС
  + **Форматы данных**
    - точки соединения в проектах, ориентированных на перемещение данных, а не потока управления
    - Компонент заменяется аналогичным, работающим с теми же типами данных

1. Проектирование программных систем. Модели управления: вызов- возврат, менеджер, широковещательная и с прерываниями. Привести определения и примеры.

Модели управления

* Централизованное управление – одна из подсистем является системным контроллером и управляет другими подсистемами
  + Модель вызов-возврат



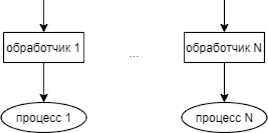
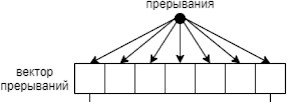
* + Модель менеджера (в системах параллельной обработки)



* Модель событийного управления – внешние события управляют системой
  + Широковещательная модель
    - Подсистема – сообщает обработчику о «ее» событиях (интерес к конкретным событиям, слушатель)
    - Обработчик – если было событие, то пересылает его подсистеме. Функций управления не имеет.



* + Модель, управляемая прерываниями
    - Для любого типа (группы) прерываний – свой обработчик
    - Любой обработчик реагирует на свой тип прерывания и запускает свой процесс



1. Структурный проект. Выделение уровней. Управление зависимостями. Устранение циклической зависимости. Зависимости наследования, уровней, методов. Делегирование. Зависимости реализации и интерфейсов. Привести определения и примеры иллюстрирующих диаграмм.

# Структурный проект

Структурный проект – множество решений для создания эффективной структуры программного обеспечения (ПО) с обоснованием решений.

Особенности

* Касается нефункциональных требований;
* Создает фундаментальные решения верхнего уровня;
* Определяет зависимости и компромиссы;
* Обеспечивает формирование и оценку альтернативных решений.
* Структура ПО – организация компонентов ПО в систему для

**достижения некой цели.**

# Процесс структурного проектирования

* Методология построения алгоритмов, программ и систем, в основе которой лежит выявление структуры задачи, определение составляющих компонентов и выделение связей между ними.
* Происходит итерационно и пошагово;
* Управляется прецедентами;
* Сконцентрирован вокруг структуры.

# Выделение уровней.

Структурный проект как результат:

* Иерархическое выделение уровней модулей и определение связей модулей:
  + Снижается сложность;
  + Снижается сцепление.
* Использование стандартов разработки:
  + Повышается видимость зависимостей;
  + Снижается сложность;
  + Снижается сцепление.
* Модель / шаблон уровней
* Шаблоны проектирования

Управление зависимостями.

# Зависимости

Семантическое отношение между двумя предметами, в котором изменение в одном предмете (независимом предмете) может влиять на семантику другого предмета (зависимого предмета).

Главная цель разработки программного обеспечения – свести количество зависимостей к минимуму. Ненужные зависимости должны быть устранены из структурного проекта.

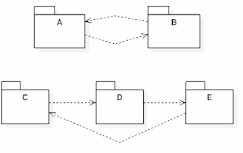
Причины зависимости:

* Импорт – А (часть А) обращается к В (части В);
* Изменение В приводит к изменению А (перекомпиляция);
* А может быть использован только с В (повторно)



# Устранение циклической зависимости.

Циклическая зависимость



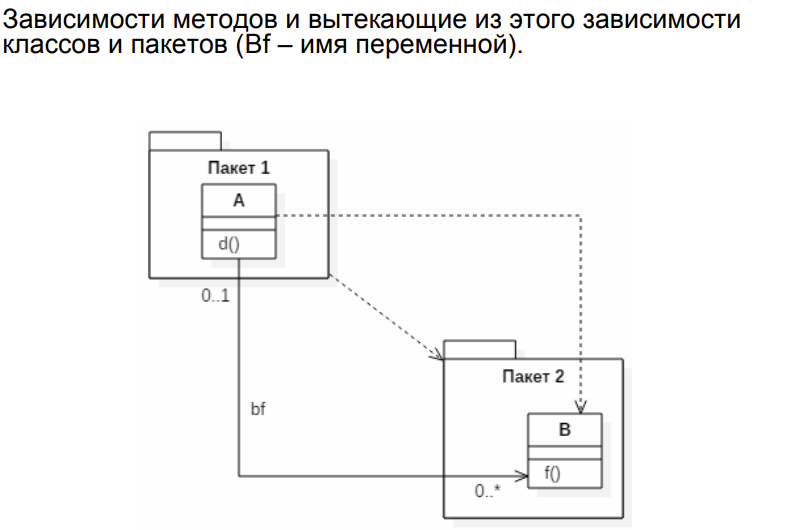
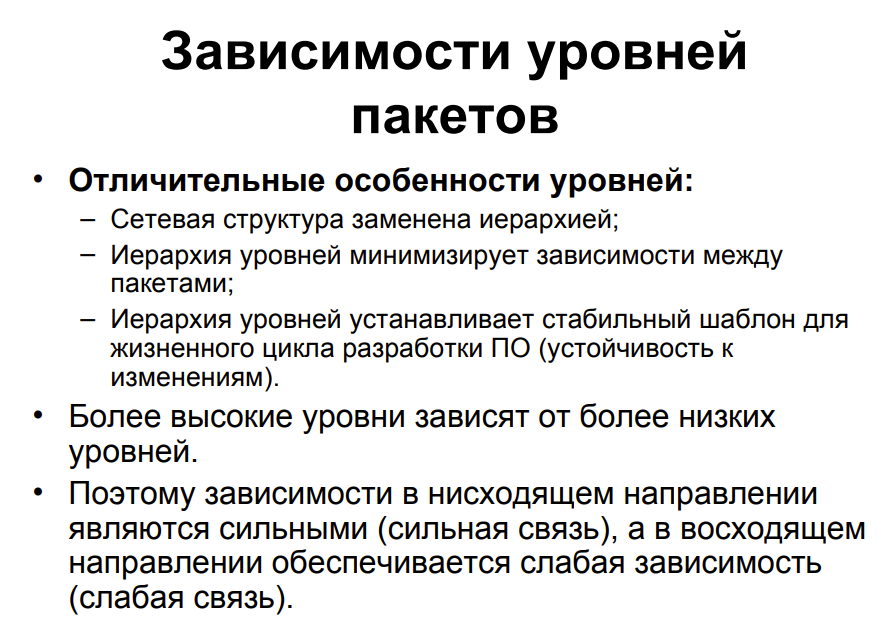
# Циклическую зависимость между пакетами следует устранять при проектировании

**Устранение циклической зависимости через интерфейс**

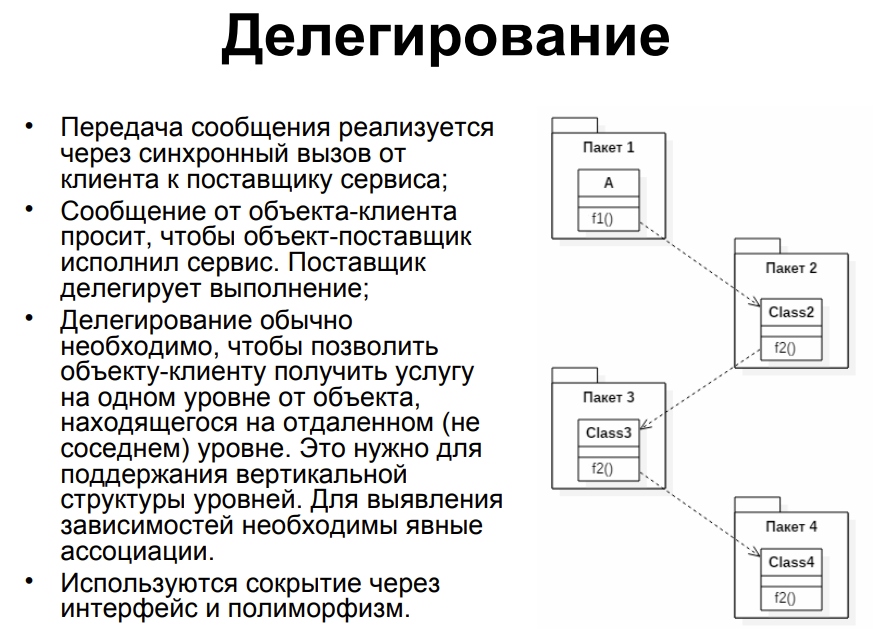
Для того чтобы нарушить цикл, интерфейс и класс, который реализует его, должны находиться в различных пакетах

# Зависимости наследования, уровней, методов.





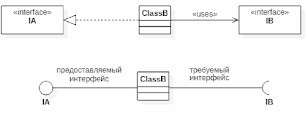
Делегирование.



Зависимости реализации и интерфейсов.

Предоставленные интерфейсы определяются в UML 2.0 отношением зависимости между классом и интерфейсом, реализованным данным классом. Это называется зависимостью реализации.

Требуемые интерфейсы определены в UML 2.0 отношением зависимости между классом (или интерфейсом) и требуемым интерфейсом. Это называется зависимостью использования



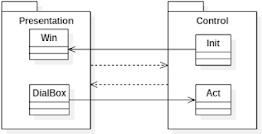
# Структурный проект. Интерфейсы предоставляемые и требуемые. Доминирующий класс. Использование интерфейсов для устранения циклических зависимостей. Отделенный интерфейс. Привести примеры иллюстрирующих диаграмм классов и взаимодействия.

**Доминирующий класс.**

Доминирующий класс – класс, который реализует главные интерфейсы и абстрактные классы в пакете.

**Использование интерфейсов для устранения циклических зависимостей.**

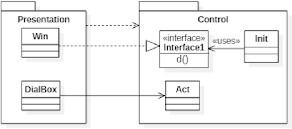
Устранение циклической зависимости через интерфейс



# Для того чтобы нарушить цикл, интерфейс и класс, который реализует его, должны находиться в различных пакетах.

**Отделенный интерфейс.**

Отделенный интерфейс – паттерн проектирования Фаулера



1. Структурный проект. Синхронные и асинхронные сообщения. Организация подписки. Знакомства. Пакет знакомств. Привести примеры иллюстрирующих диаграмм классов и взаимодействия.

Объектная декомпозиция: Структура объектов и связи между ними; поведение ПО как обмен сообщений между объектами (синхронные и асинхронные).

# Синхронные и асинхронные сообщения.

Существует два варианта передачи сообщений:

* **Синхронная передача.** При синхронной передаче сообщения объект-клиент A

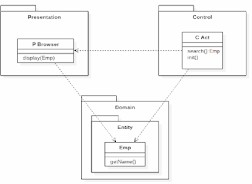
просит объекта-поставщика В выполнить сервис, который может быть делегирован. При этом A зависит от B, потому что A ожидает некоторые результаты выполнения от B.

* **Асинхронная передача (оповещение).**

При асинхронной обработке события для объекта-издателя характерно:

* + объект-издатель – отправитель сообщения;
  + объект-издатель имеет список подписчиков и метод их добавления;
  + вызывается извне по событию (например, от GUI), преобразует внешнее событие в свой формат и рассылает его подписчикам;
  + не знает подписчиков.
* Для объекта-подписчика характерно:
  + Интересуется событиями, для которых он имеет свои обработчики.
* Для объекта-регистратора характерно:
  + наличие необязательно, может быть подписчиком;
  + добавляет подписчика издателю как параметр метода (может быть передан как интерфейс) для нисходящих зависимостей

# Организация подписки. Знакомство

* Знакомство соответствует ситуации, когда объект передает объект другому объекту как аргумент его метода.
* А знакомится с В, если С передаёт В для А как параметр.

# Пакет знакомств

Знакомство возникает тогда, когда внутри метода сообщение посылается объекту, который является параметром этого метода

Для не соседних уровней используется пакет знакомств

Пакет знакомств — это автономный пакет, состоящий только из интерфейсов.

Низкий уровень является устойчивым, его трудно изменить, но при этом легко расширить.

1. Проектирование программных систем. Интерфейсы взаимодействия на основе: API, иерархии классов, форматов данных и компонентных технологий. Привести определения и примеры.

# Виды интерфейсов

* Точки соединения и границы компонентов
* Внешний вид и доступ

# API - интерфейсы прикладного программирования

**API (Application programming interface) -** описание способов взаимодействия одной компьютерной программы с другими.

API упрощает процесс программирования при создании приложений, создавая абстракцию базовой реализации и предоставляя только объекты или действия, необходимые разработчику. Если графический интерфейс для почтового клиента может предоставить пользователю кнопку, которая выполнит все шаги для выборки и выделения новых писем, то API для ввода/вывода файлов может дать разработчику функцию, которая копирует файл из одного места в другое, не требуя от разработчика понимания операций файловой системы, происходящих за кулисами

# SOAP API

SOAP – Simple Object Access Protocol, т. е. простой протокол доступа к объектам. Клиент и сервер обмениваются сообщениями посредством XML. Это менее гибкий API, который был более популярен в прошлом.

# RPC API

Такие API называются системой удаленного вызова процедур. Клиент выполняет функцию (или процедуру) на сервере, и сервер отправляет результат обратно клиенту.

# Websocket API

это еще одна современная разработка web API, которая использует объекты JSON для передачи данных. WebSocket API поддерживает двустороннюю связь между клиентскими приложениями и сервером. Сервер может отправлять сообщения обратного вызова подключенным клиентам, что делает его более эффективным, чем REST API.

# REST API

На сегодняшний день это самые популярные и гибкие API- интерфейсы в Интернете. Клиент отправляет запросы на сервер в виде данных. Сервер использует этот клиентский ввод для запуска внутренних функций и возвращает выходные данные обратно клиенту. Давайте рассмотрим API REST более подробно ниже.

# Иерархия классов

* + интерфейс сохраняется в производных классах (реализация абстрактных классов)

# Компонентные технологии (COM/CORBA)

* + Реализация выбирается при выполнении
  + IDL – язык определения интерфейса (абстрактный)
  + Реализация на любом языке
  + Необходима поддержка со стороны промежуточного ПО или ОС

# Форматы данных

* + точки соединения в проектах, ориентированных на перемещение данных, а не потока управления
  + Компонент заменяется аналогичным, работающим с теми же типами данных

1. Структурные шаблоны. MVC: подсистемы, их назначение и схема взаимодействия. Требования и особенности. Реализация обратного вызова. Привести примеры иллюстрирующих диаграмм классов и взаимодействия.

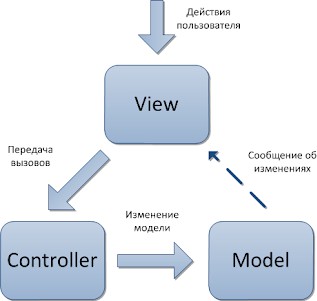
# Структурные шаблоны

Структурные шаблоны — шаблоны проектирования, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры.

Структурные шаблоны уровня класса используют наследование для составления композиций из интерфейсов и реализаций. Простой пример — использование множественного наследования для объединения нескольких классов в один. В результате получается класс, обладающий свойствами всех своих родителей. Особенно полезен этот шаблон, когда нужно организовать совместную работу нескольких независимо разработанных библиотек.

# MVC

**Model-View-Controller** (MVC) — это базовый шаблон проектирования, который отделяет логику пользовательского интерфейса от бизнес-логики.



**MVC** состоит из трех компонентов:

* **View** (представление)
* **Model** (модель)
* **Controller** (контроллер)

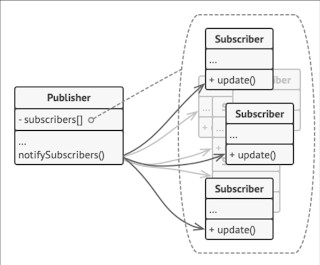
Уровень **Models** - это самый низкий уровень вашего приложения. Он располагается поверх хранилища данных и выполняет манипуляции с данными на уровне базы данных. Более того, современные идиомы проектирования реализуют большую часть логики вашего приложения на уровне модели, что означает, что модели не только управляют своими собственными атрибутами, но и отвечают за обработку всех побочных эффектов в задней части вашего приложения, когда происходит изменение.

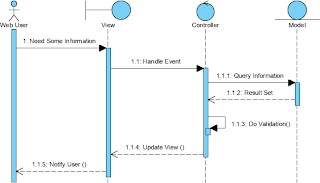
Уровень **View** (представление, пользовательский интерфейс) - уровень, который видит конечный пользователь, он напрямую связан со всем, что видят пользователи вашего приложения, виды рендерятся на основе данных, хранимых в **Model**.

Уровень **Controller** (контроллер) - обозначает действия пользователя, все кнопки, существующие на уровне **View** привязаны к **Controller, Controller** получая действия пользователя, обрабатывает их соответствующим образом и взаимодействует с **Model**

# Реализация обратного вызова

Обратный вызов в MVC происходит с помощью паттерна Observer, в паттерне Observer есть слушатели и наблюдаемый объект. В рамках данного паттерна на изменения подписываются несколько объектов. В случае появления сообщения, оно рассылается всем, заранее подписанным объектам

Наблюдатель — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах



1. Структурные шаблоны. PCMEF: подсистемы, их назначение и схема взаимодействия. Требования и особенности. Делегирование. Привести примеры иллюстрирующих диаграмм классов и взаимодействия.

Структурные шаблоны —см. 29

PCMEF (Presentation, Control, Mediator, Entity, Foundation) – уровневый структурный шаблон, организованный в виде вертикальной иерархии (представление, управление, посредник, сущность, основание)

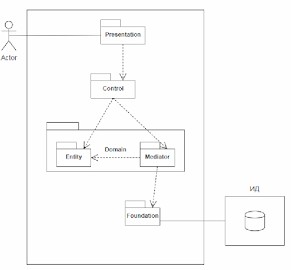
Presentation представляет собой GUI в виде стандартных классов (или их производных).

* + В **Control** хранится логика, обработка событий, алгоритмы и сеансы
  + В **Entity** хранятся бизнес-объекты (контейнеры) из БД в ОП
  + **Mediator** является посредником между Entity и Foundation вне

Control

* + **Foundation** осуществляет коммуникацию с базой данных или

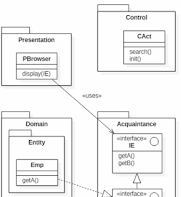
web-сервисами.



1. Структурные шаблоны. PCMEF+: подсистемы, их назначение и схема взаимодействия. Требования и особенности. Привести примеры иллюстрирующих диаграмм классов и взаимодействия.

PCMEF+

* пакет Acquaintance (знакомство) отделяет зависимости знакомств в изолированный пакет, который может управляться независимо.
* Пакет Acquaintance состоит только из интерфейсов.
* IE1 (интерфейс пакета Acquaintance с методами getA() и getB()) расширен интерфейсом IE2 (интерфейс пакета Acquaintance с методом getC()), который в свою очередь реализован классом Emp.





Зависимости между пакетами будут следующие:

1. Foundation использует интерфейсы Acquaintance, реализованные в пакетах

Domain, Control и Presentation;

1. Domain использует интерфейсы Acquaintance, реализованные в пакетах

Control и Presentation;

1. Entity использует интерфейсы Acquaintance, реализованные в пакете

Presentation;

1. Presentation использует интерфейсы Acquaintance, реализованные в пакетах

Domain и Foundation;

1. Control использует интерфейсы Acquaintance, реализованные в пакете

Foundation.

1. Паттерны проектирования. Назначение и принцип описания. Типы и виды паттернов (бизнес-логика, БД, Gof). Параметры паттернов. Привести определения и примеры.

Паттерны работы с базой данных:

* Row Data Gateway (Шлюз записи данных)
* Table Data Gateway (Шлюз таблицы данных)
* Active Record (Активная запись)
* Data Mapper (Преобразователь данных) Паттерны бизнес-логики:
* Transaction Script (Сценарий транзакций)
* Domain Model (Модель предметной области)
* Table Module (Модуль таблицы)
* Service Layer (Слой служб)

Паттерны GOF (поведенческие):

* Команда (Command)
* Стратегия (Strategy)
* Шаблонный метод (Template method)
* Наблюдатель (Observer)
* Цепочка обязанностей (Chain of responsibility)
* Итератор (Iterator)
* Интрерпретатор (Interpreter)
* Посредник (Mediator)
* Состояние (State)
* Хранитель (Memento)
* Посетитель (Visitor)

1. Паттерны для реализации бизнес-логики. Сценарий транзакций и модель предметной области. Назначение, особенности реализации и область применения. Примеры диаграмм классов и взаимодействия для указанных паттернов.

**Transaction script**

Способ организации бизнес-логики по процедурам, каждая из которых обслуживает один запрос, инициируемый слоем представления. Сценарий транзакции организует логику вычислительного процесса в виде единой процедуры, которая выполняет все действия, необходимые для выполнения прикладной функции. Каждой функции приложения ставится в соответствие собственный сценарий транзакции (общие подзадачи могут быть вынесены в подчиненные процедуры).

Вся бизнес-логика делится на процедуры:

* 1 класс – одна процедура (соответствует функции приложения).
  + · Базовый класс-класс с методом run() от которого наследуются другие классы для реализации отдельных функций.

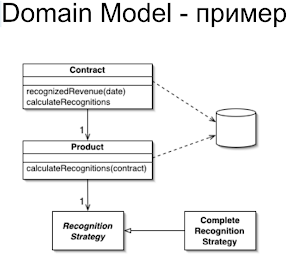




**Domain model**

Предусматривает создание сети взаимосвязанных объектов, каждый из которых представляет некую осмысленную сущность. Означает наполнение приложения слоем объектов, описывающих сущности реального мира и их взаимодействие. Одни объекты соответствуют элементам данных, которыми оперируют в этой области, а другие реализуют те или иные бизнес-правила.

Функции тесно сочетаются с данными, которыми они манипулируют. Объект содержит данные (для одной записи/сущности БД) и бизнес-логику их обработки. Объект выполняет обработку только своих данных, для обработки данных другого объекта вызывает методы другого объекта. Выполнение прикладной функции приложения будет последовательностью вызовов методов всех задействованных объектов. Один объект хранит данные для одной записи БД.



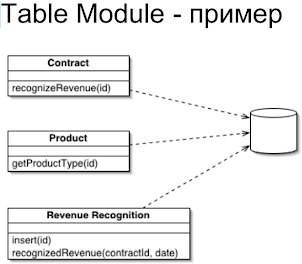
1. Паттерны для реализации бизнес-логики. Модуль таблицы и слой служб. Назначение, особенности реализации и область применения. Примеры диаграмм классов и взаимодействия для указанных паттернов.

**Table Module (Модуль таблицы)**

Предусматривает создание по одному классу на каждую таблицу базы данных, и единственный экземпляр класса содержит всю логику обработки данных таблицы.

Основное отличие модуля таблицы от модели предметной области состоит в том, что в соответствии с моделью предметной области придется сконструировать по одному объекту на каждую запись БД, а при использовании модуля таблицы понадобится всего

один объект, представляющий одновременно записи таблицы БД.

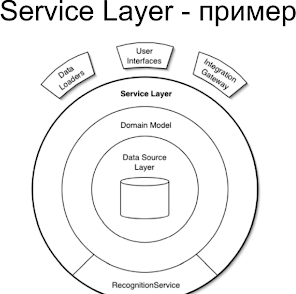


**Service Layer (Слой служб)**

Определяет границу между приложением и слоем сервисов, который образует набор доступных операций и управляет ответом приложения в каждой операции.

Определяет для приложения границу и набор допустимых операций с точки зрения взаимодействующих с ним клиентских модулей.

Он инкапсулирует бизнес-логику приложения, управляя транзакциями и управляя ответами в реализации этих операций.





1. Паттерны для работы с БД. Шлюз таблицы и шлюз записи. Назначение, особенности реализации и область применения. Примеры диаграмм классов и взаимодействия для указанных паттернов.

## Table Data Gateway

(Шлюз таблицы данных)

* Объект выступает в качестве шлюза между данными в приложении и в БД.
* Объект (Класс) содержит методы для доступа к отдельной таблице или представлению (view): выборка, обновление, вставка, удаление (CRUD) и вызов хранимых процедур.
* Один объект работает сразу со всеми записями в таблице.
* Может быть один шлюз на несколько таблиц
* М.б. Класс со статическими методами
* Вход методов: параметры для формирования запроса
* Для insert - все поля (атрибуты) объекта
* Для update – атрибуты сущности БД + ид
* Для delete - часть параметров (ид) для корректного определения объекта и удаления его
* Выход методов: поиск: Record set, Коллекция, Объект; или признак удачной операции



## Row Data Gateway

(Шлюз записи данных)

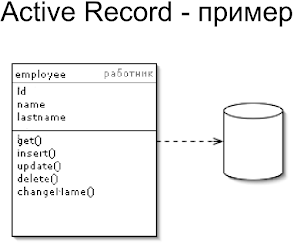
* Объект выступает в роли шлюза к отдельной записи в источнике данных. Один экземпляр на одну запись.
* 1 объект - это одна запись из таблицы БД,
* атрибуты объекта включают все поля записи из БД,
* методы CRUD работают без параметров,
* методы CRUD берут данные для работы из свойств объекта,
* методы поиска статичны, либо прописываются отдельно (class PersonFinder).



1. Паттерны для работы с БД. Активная запись и преобразователь данных. Назначение, особенности реализации и область применения. Примеры диаграмм классов и взаимодействия для указанных паттернов.

**Active Record (Активная запись)**

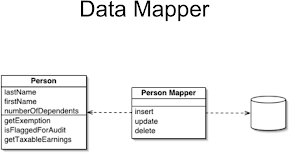
* + Объект, выполняющий роль оболочки для строки таблицы БД или представления. Инкапсулирует доступ к БД и добавляет к данным логику домена
  + атрибуты объекта включают все поля записи из БД,
  + методы CRUD работают без параметров,
  + методы CRUD берут данные для работы из свойств объекта,
  + содержит бизнес- логику обращения с данными.
  + Отличие от шлюза записи – наличие бизнес-логики



**Data Mapper**

(Преобразователь данных)

* + Слой преобразователей для передачи данных между объектами и таблицами БД, сохраняя их независимо друг от друга и преобразователей
  + Объекты не нуждаются в знании о существовании БД. Они не нуждаются в SQL-коде и в информации о структуре БД.
  + Вход методов: объекты бизнес-логики (CUD) и параметры запроса поиска (R)
  + Выход методов: объекты, коллекция объектов (поиск)



1. Паттерны GoF. Порождающие, структурные и поведенческие. Абстрактная фабрика и фасад. Назначение, особенности реализации и область применения. Примеры диаграмм классов и взаимодействия для указанных паттернов.

Кооперация (сотрудничество) - это средство представления комплексных решений в разработке ПО на внешнем архитектурном уровне.

Паттерн – это параметризованная настраиваемая кооперация для решения типичной проблемы в определенном контексте.

Паттерны — это наборы готовых решений с возможностью повторного использования.

Дает:

* уменьшение затрат на анализ и проектирование ПО,
* повышение качества и правильности разработки на логическом уровне. GOF – Gang of four (Команда четырех): Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.

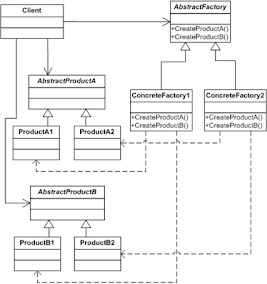
Паттерны GOF включают в себя 23 паттерна на высоком уровне абстракции, которые разделяют на 3 вида:

* Порождающие
* Структурные
* Поведенческие

Паттерн **Абстрактная фабрика** относится к порождающим паттернам. Абстрактная фабрика (Abstract factory) - класс, который представляет собой интерфейс для создания семейства компонентов системы.

* (+) Изолирует конкретные классы (имена неизвестны клиенту).
* (+) Гарантирует сочетаемость продуктов (в одном семействе).
* (+) Упрощает замену семейств продуктов.
* (-) Сложно расширять фабрику. Использование:
* Когда система не должна зависеть от способа создания новых объектов
* Когда создаваемые объекты должны использоваться вместе и являются взаимосвязанными

## Диаграмма классов:



Абстрактные классы AbstractProductA и AbstractProductB определяют интерфейс для классов, объекты которых будут создаваться в программе.

Конкретные классы ProductA1 / ProductA2 и ProductB1 / ProductB2 представляют конкретную реализацию абстрактных классов

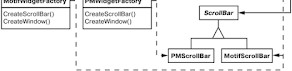
Абстрактный класс фабрики AbstractFactory определяет методы для создания объектов. Причем методы возвращают абстрактные продукты, а не их конкретные реализации.

Конкретные классы фабрик ConcreteFactory1 и ConcreteFactory2 реализуют абстрактные методы базового класса и непосредственно определяют какие конкретные продукты использовать

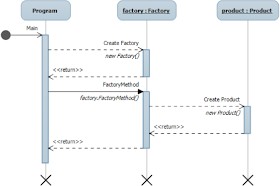
Класс клиента Client использует класс фабрики для создания объектов. При этом он использует исключительно абстрактный класс фабрики AbstractFactory и абстрактные классы продуктов AbstractProductA и AbstractProductB и никак не зависит от их конкретных реализаций

**Конкретный пример**





**Диаграмма взаимодействия**



Паттерн **Фасад** относится к структурным паттернам.

Представляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Определяет интерфейс более высокого уровня, который упрощает исполнение подсистемы.

Проблема:

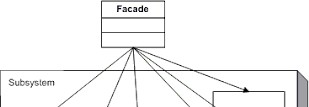
* простой интерфейс к сложной подсистеме (сокрытие деталей; вид системы по умолчанию с возможностью обратиться прямо);
* увеличение независимости и переносимости (отдельные подсистемы и классы);
* разделение подсистем на слои. Фасад для входа на любой уровень. Назначение:
* Разделение классов и подсистем => уменьшение сцепления подсистем.
* Упрощение переноса на другую платформу.
* Может быть прямое обращение к классам Подсистемы.

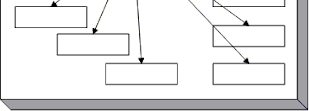
Фасад - альтернатива абстрактной фабрики.

Фасад может быть создан Абстрактной фабрикой (создание и исполнение элементов подсистемы).

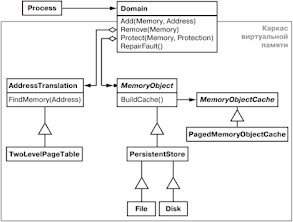
Фасад может быть одиночкой.

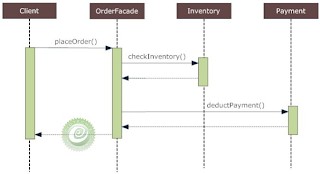
## Диаграмма классов:





Фасад знает класс подсистемы, делегирует запрос клиента классам подсистемы. Класс подсистемы реализует функциональность подсистемы, не знает о фасаде. **Конкретный пример**



**Диаграмма взаимодействия (пример)**

1. Паттерны GoF. Порождающие, структурные и поведенческие. Строитель и посредник. Назначение, особенности реализации и область применения. Примеры диаграмм классов и взаимодействия для указанных паттернов.

Паттерны GOF см. вопрос 37

Паттерн **Строитель** относится к порождающим паттернам. Инкапсулирует создание объекта и позволяет разделить его на различные этапы.

Отделяет конструирование сложного объекта от его представления, т.ч. в результате одного процесса конструирования могут быть созданы различные

представления. Проблемы:

* алгоритм создания сложного объекта не должен зависить от того, из каких частей он состоит и как они стыкуются;
* конструирование должно обеспечить различные представления объекта.

Назначение:

* Позволяет изменять внутреннее представление продукта

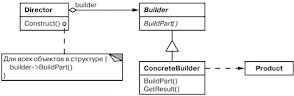
(распорядитель не знает классы внутри продукта).

* Изолирует код, реализующий конструирование, и код - представление.
* Тонкий контроль за строительством (по шагам).

Абстрактная фабрика -> сразу создается семейство, возвращает результат сразу. Строитель -> создается по шагам, возвращает продукт в конце.

Часто строитель делает компоновщика.

## Диаграмма классов:



Product: представляет объект, который должен быть создан. В данном случае все части объекта заключены в списке parts.

Builder: определяет интерфейс для создания различных частей объекта Product ConcreteBuilder: конкретная реализация Buildera. Создает объект Product и определяет интерфейс для доступа к нему

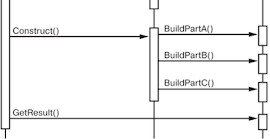
Director: распорядитель - создает объект, используя объекты Builder

## Конкретный пример



**Диаграмма взаимодействия**





Паттерн **Посредник** относится к поведенческим паттернам.

Определяет объект, содержащий способ взаимодействия множества объектов. Способствует уменьшению числа связей между объектами, позволяя им работать без явных ссылок друг на друга. Это, в свою очередь, дает возможность независимо изменять схему взаимодействия.

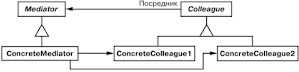
Проблемы:

* связи между объектами сложны, четко определены, не структурированы, сложны для понимания;
* нельзя повторно использовать объект, т.к. он сильно сцеплен;
* поведение между классами должно быть настраиваемо без порождения подклассов.

Назначение:

* (+) уменьшается количество подклассов;
* (+) уменьшается сцепление коллег -> 0 (независимое изменение коллег);
* (+) упрощаются протоколы взаимодействия объектов (М-М) => (1-М);
* (+) абстракция и инкапсуляция кооперации объектов;
* (-) централизация управления (монолит, сложен).

## Диаграмма классов:



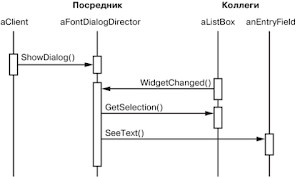
Mediator: представляет интерфейс для взаимодействия с объектами Colleague Colleague: представляет интерфейс для взаимодействия с объектом Mediator ConcreteColleague1 и ConcreteColleague2: конкретные классы коллег, которые обмениваются друг с другом через объект Mediator

ConcreteMediator: конкретный посредник, реализующий интерфейс типа Mediator

## Конкретный пример





**Диаграмма взаимодействия (пример)**

1. Паттерны GoF. Порождающие, структурные и поведенческие. Одиночка и стратегия. Назначение, особенности реализации и область применения. Примеры диаграмм классов и взаимодействия для указанных паттернов.

Паттерны GOF см. вопрос 37

Паттерн **Одиночка** относится к порождающим паттернам. Гарантирует, что у класса есть единственный экземпляр, и предоставляет глобальную точку доступа к этому экземпляру.

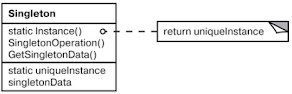
Задача:

* должен быть ровно один экземпляр некоторого класса, доступный всем;
* единственный экземпляр должен расширяться путём порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

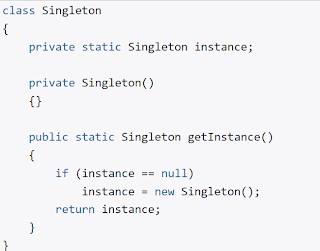
Назначение:

* Контролирует доступ к единственному экземпляру.
* Уменьшает количество имен (вместо глобальной переменной).
* Может быть более одного экземпляра без модификации приложения.
* Может быть расширение операций и представлений без модификации приложения.
* С помощью одиночки могут быть реализованы: абстрактная фабрика, строитель, прототип

## Диаграмма классов:



В классе определяется статическая переменная - ссылка на конкретный экземпляр

данного объекта и приватный конструктор. В статическом методе getInstance() этот конструктор вызывается для создания объекта, если, конечно, объект отсутствует и равен null.

Паттерн **Стратегия** относится к поведенческим паттернам. Определяет семейство алгоритмов, инкапсулирует каждый из них и делает взаимозаменяемыми.

Позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются. Проблемы:

* n возможных классов с различным поведением; надо задать 1 из них;
* нужно n вариантов алгоритмов (может быть иерархия классов);
* надо скрыть действие алгоритма от клиента;
* много поведений с условием => любая ветка в отдельный класс. Применение: Текст в окне -> разбиение на строки (просто; по количеству символов; с оптимизацией)

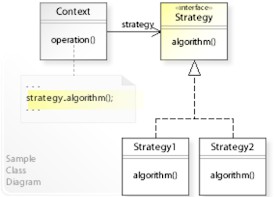
(+) Семейство родственных алгоритмов (иерархия классов общих алгоритмов). (+) Альтернатива подклассам контекста.

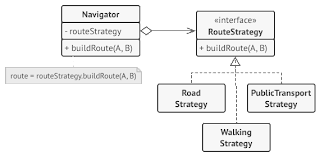
(+) Избавление от условных операторов. (+) Набор реализаций (выбор).

(-) Клиент должен знать о стратегиях для их выбора. (-) Увеличение количества объектов.

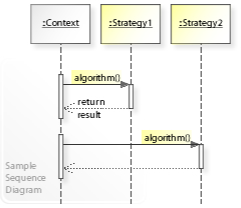
(-) Избыток параметров от контекста к конкретной стратегии (из- за унификации).

## Диаграмма классов:



**Конкретный пример**

**Диаграмма взаимодействия**



1. Паттерны GoF. Порождающие, структурные и поведенческие. Компоновщик и шаблонный метод. Назначение, особенности реализации и область применения. Примеры диаграмм классов и взаимодействия для указанных паттернов.

Паттерны GOF см. вопрос 37

Паттерн **Компоновщик** относится к структурным паттернам. Обеспечивает представление иерархий часть/целое, объединяя объекты в древовидные структуры

Проблема:

* объединение маленьких компонентов в крупные (контейнеры),

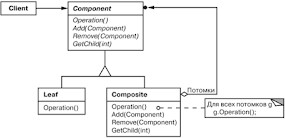
объединение компонентов в большие и в огромные;

* клиент по-разному работает с этими объединениями. Применяется:
* построение иерархии целое-часть;
* унификация использования составных/ индивидуальных объектов.

(+) иерархия из примитивов и контейнеров;

(+) облегчает добавление новых разновидностей компонентов; (+) упрощает организацию клиента (унифицирует обращения); (-) сложно накладывать ограничения на объекты в Композиции.

## Диаграмма классов:

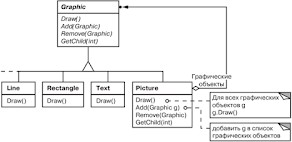


Component: определяет интерфейс для всех компонентов в древовидной структуре Composite: представляет компонент, который может содержать другие компоненты и реализует механизм для их добавления и удаления

Leaf: представляет отдельный компонент, который не может содержать другие компоненты

Client: клиент, который использует компоненты

## Конкретный пример



Паттерн **Шаблонный метод** относится к поведенческим паттернам. Определяет основу алгоритма и позволяет подклассам переопределить некоторые шаги алгоритма, не изменяя его структуру в целом

Проблема:

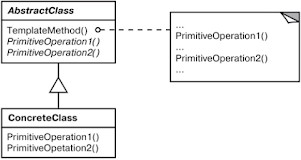
* Однократное использование общей части алгоритма (изменяется только часть алгоритма в подклассах);
* Выделение общей части кода (для избежания дублирования в подклассах);
* Управление расширениями подклассов (шаблонный метод вызывает операции зацепки в определенных точках).

Применение:

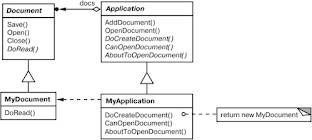
Основа повторного использования (особенно для библиотеки классов). Фабричный метод вызывается шаблонным методом.

Шаблонный метод использует наследование для модификации части алгоритма, а Стратегия - делегирование .

## Диаграмма классов:



Конкретный пример



1. Паттерны GoF. Порождающие, структурные и поведенческие. Мост и итератор. Назначение, особенности реализации и область применения. Примеры диаграмм классов и взаимодействия для указанных паттернов.

Паттерны GOF см. вопрос 37

Паттерн **Мост** относится к структурным паттернам. Отделяет абстракцию от реализации так, что их можно изменять независимо.

Проблема:

* без постоянной привязки абстракции к реализации (изменение реализации во времени выполнения);
* абстракция и реализация должны быть расширены подклассами;
* изменение абстракции ≠> изменение реализации;
* cокрытие от класса интерфейса реализации;
* скрытое разделение реализации среди n объектов. Назначение:
* Отделяет абстракцию и реализацию.
* Увеличивает степень расширяемости.
* Сокрытие деталей реализации от клиента.
* Мост создается изначально.
* Адаптер соединяет существующие.
* Для создания/конфигурации моста используется абстрактная фабрика.

## Диаграмма классов:

Abstraction: определяет базовый интерфейс и хранит ссылку на объект Implementor. Выполнение операций в Abstraction делегируется методам объекта Implementor RefinedAbstraction: уточненная абстракция, наследуется от Abstraction и расширяет унаследованный интерфейс

Implementor: определяет базовый интерфейс для конкретных реализаций. Как правило, Implementor определяет только примитивные операции. Более сложные операции, которые базируются на примитивных, определяются в Abstraction ConcreteImplementorA и ConcreteImplementorB: конкретные реализации, которые унаследованы от Implementor

Client: использует объекты Abstraction

## Конкретный пример

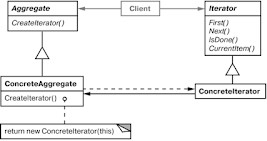


Паттерн **Итератор** относится к поведенческим паттернам. Предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления.

Применение:

* + Когда необходимо осуществить обход объекта без раскрытия его внутренней структуры
  + Когда имеется набор составных объектов, и надо обеспечить единый интерфейс для их перебора
  + Когда необходимо предоставить несколько альтернативных вариантов перебора одного и того же объекта

## Диаграмма классов:



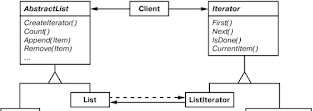
Iterator: определяет интерфейс для обхода составных объектов

Aggregate: определяет интерфейс для создания объекта-итератора ConcreteIterator: конкретная реализация итератора для обхода объекта Aggregate. Для фиксации индекса текущего перебираемого элемента использует целочисленную переменную \_current

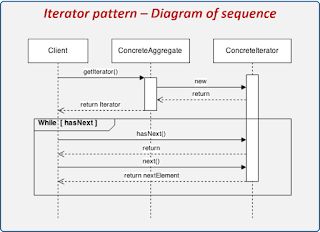
ConcreteAggregate: конкретная реализация Aggregate. Хранит элементы, которые надо будет перебирать

Client: использует объект Aggregate и итератор для его обхода

**Конкретный пример**





**Диаграмма взаимодействия**

1. Паттерны GoF. Порождающие, структурные и поведенческие. Приспособленец и наблюдатель. Назначение, особенности реализации и область применения. Примеры диаграмм классов и взаимодействия.

Паттерны GOF см. вопрос 37

Паттерн **Приспособленец** относится к структурным паттернам. Использует разделение для поддержки множества мелких объектов.

Применение:

* Когда приложение использует большое количество однообразных объектов, из-за чего происходит выделение большого количества памяти
* Когда часть состояния объекта, которое является изменяемым, можно вынести во вне. Вынесение внешнего состояния позволяет заменить множество объектов небольшой группой общих разделяемых объектов.

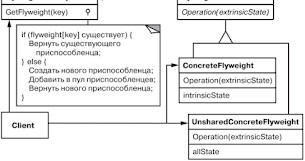
Ключевым моментом здесь является разделение состояния на внутренне и внешнее. Внутреннее состояние не зависит от контекста. В примере с символами внутреннее состояние описывается кодом символа из таблицы кодировки. Так как внутреннее состояние не зависит от контекста, то оно может быть разделяемым и поэтому выносится в разделяемые объекты.

Внешнее состояние зависит от контекста, является изменчивым. В применении к символам внешнее состояние может представлять положение символа на странице. То есть код символа может быть использован многими символами, тогда как положение на странице будет для каждого символа индивидуально.

При создании приспособленца внешнее состояние выносится.

## Диаграмма классов:





Flyweight: определяет интерфейс, через который приспособленцы-разделяемые объекты могут получать внешнее состояние или воздействовать на него ConcreteFlyweight: конкретный класс разделяемого приспособленца. Реализует интерфейс, объявленный в типе Flyweight, и при необходимости добавляет внутреннее состояние. Причем любое сохраняемое им состояние должно быть внутренним, не зависящим от контекста

UnsharedConcreteFlyweight: еще одна конкретная реализация интерфейса, определенного в типе Flyweight, только теперь объекты этого класса являются неразделяемыми

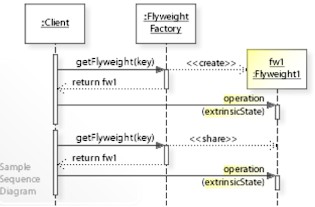
FlyweightFactory: фабрика приспособленцев - создает объекты разделяемых приспособленцев. Так как приспособленцы разделяются, то клиент не должен создавать их напрямую. Все созданные объекты хранятся в пуле. В примере выше для определения пула используется объект Hashtable, но это не обязательно.

Можно применять и другие классы коллекций. Однако в зависимости от сложности структуры, хранящей разделяемые объекты, особенно если у нас большое количество приспособленцев, то может увеличиваться время на поиск нужного приспособленца - наверное это один из немногих недостатков данного паттерна.

Если запрошенного приспособленца не оказалось в пуле, то фабрика создает его. Client: использует объекты приспособленцев. Может хранить внешнее состояние и передавать его в качестве аргументов в методы приспособленцев

## Конкретный пример

**Диаграмма взаимодействия**

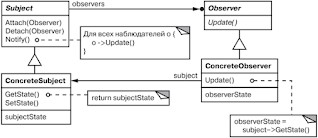


Паттерн **Наблюдатель** относится к поведенческим паттернам. Определяет зависимость 1-М между объектами таким образом, чтобы при изменении 1 объекта все зависящие от него оповещались об этом и автоматически обновлялись.

Назначение:

* Когда система состоит из множества классов, объекты которых должны находиться в согласованных состояниях
* Когда общая схема взаимодействия объектов предполагает две стороны: одна рассылает сообщения и является главным, другая получает сообщения и реагирует на них. Отделение логики обеих сторон позволяет их рассматривать независимо и использовать отдельно друга от друга.
* Когда существует один объект, рассылающий сообщения, и множество подписчиков, которые получают сообщения. При этом точное число подписчиков заранее неизвестно и процессе работы программы может изменяться

## Диаграмма классов:



Subject: представляет наблюдаемый объект. Определяет три метода: Attach() (для добавления наблюдателя), Detach() (удаление набюдателя) и Notify() (уведомление наблюдателей)

ConcreteSubject: конкретная реализация интерфейса Subject. Определяет коллекцию объектов наблюдателей.

Observer: представляет наблюдателя, который подписывается на все уведомления наблюдаемого объекта. Определяет метод Update(), который вызывается

наблюдаемым объектом для уведомления наблюдателя. ConcreteObserver: конкретная реализация интерфейса Observer.

## Диаграмма взаимодействия

.

1. Паттерны GoF. Порождающие, структурные и поведенческие. Декоратор и посетитель. Назначение, особенности реализации и область применения. Примеры диаграмм классов и взаимодействия для указанных паттернов.

Паттерн – это параметризированная настраиваемая кооперация для решения типичной проблемы в определенном контексте.

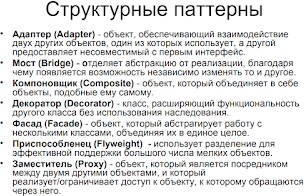
Паттерны — это наборы готовых решений с возможностью повторного использования. **Паттерны GOF – Gang of four(команда четырех) -** Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж

Виды паттернов:

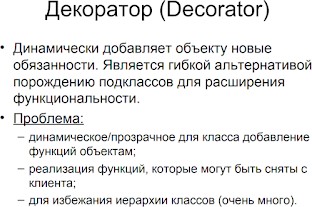
* Порождающие
* Структурные
* Поведенческие

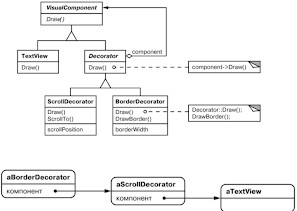




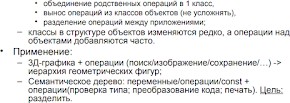




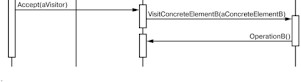


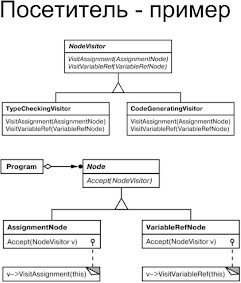








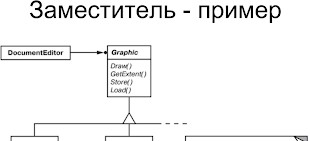
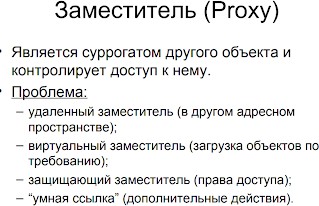




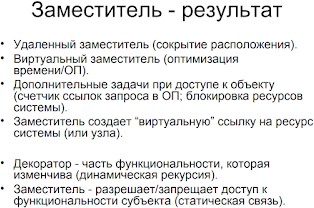
1. Паттерны GoF. Порождающие, структурные и поведенческие.

Заместитель и состояние. Назначение, особенности реализации и область применения. Примеры диаграмм классов и взаимодействия для указанных паттернов.

***см. 43***

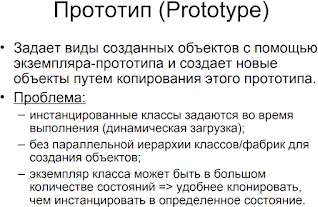


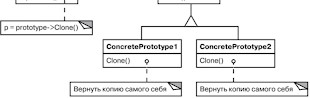


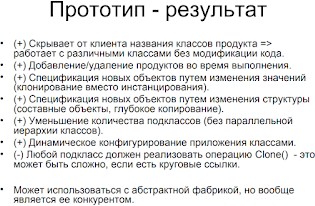
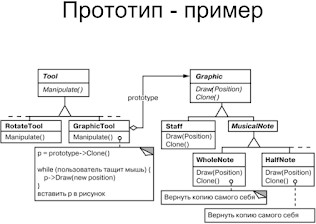


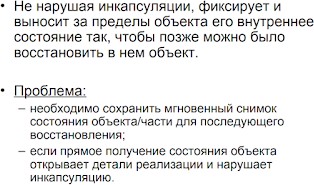
1. Паттерны GoF. Порождающие, структурные и поведенческие. Прототип и хранитель. Назначение, особенности реализации и область применения. Примеры диаграмм классов и взаимодействия для указанных паттернов.

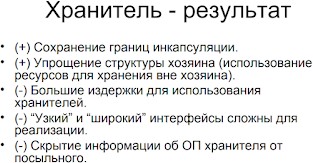
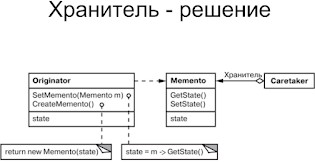
***см.43***







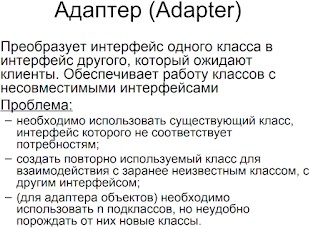






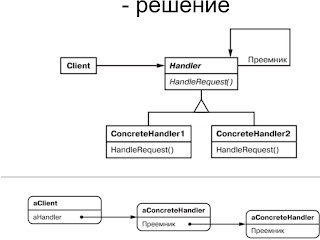
1. Паттерны GoF. Порождающие, структурные и поведенческие. Паттерны классов и объектов. Адаптер и цепочка обязанностей. Назначение, особенности реализации и область применения. Примеры диаграмм классов и взаимодействия для указанных паттернов.

***см.43***

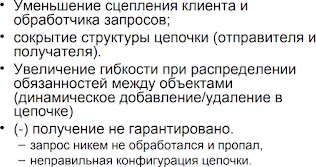






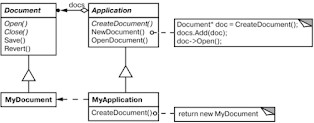


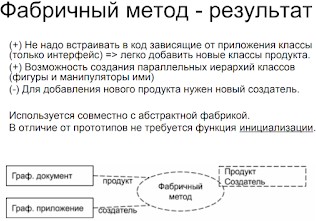




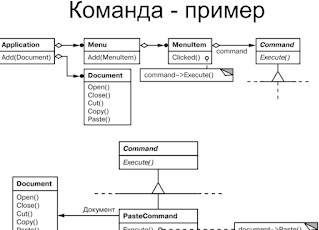
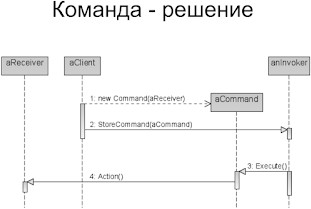
1. Паттерны GoF. Порождающие, структурные и поведенческие. Фабричный метод и команда. Назначение, особенности реализации и область применения. Примеры диаграмм классов и взаимодействия для указанных паттернов.

***см.43***







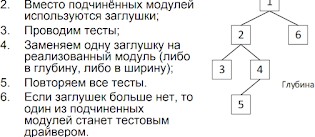


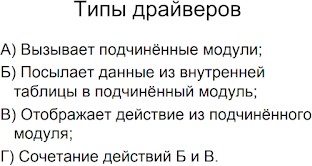


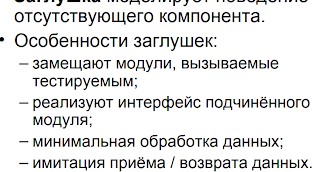
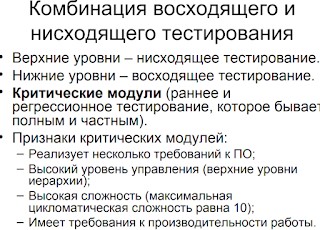
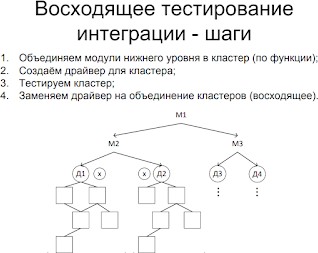


1. Тестирование ПО. Восходящее и нисходящее тестирование. Драйверы и заглушки. Автоматизация тестирования. Тестовый вариант. Привести определения и примеры.









Автоматизация тестирования — это методика автоматической проверки и утверждения программного продукта, например, веб-приложения. Она позволяет удостовериться, что продукт соответствует установленным стандартам качества в отношении оформления кода, функциональности (бизнес-логики) и удобства пользователя.

Существуют разные методики тестирования.

* Модульное тестирование: проверка работы отдельных модулей кода, например функций.
* Интеграционное тестирование: проверка способности нескольких фрагментов кода работать вместе без нежелательных последствий.
* Сквозное тестирование: проверка приложения на соответствие ожиданиям пользователя.
* Глубокое тестирование: неструктурированный подход к анализу многочисленных компонентов приложения с точки зрения пользователя для выявления функциональных и визуальных проблем.

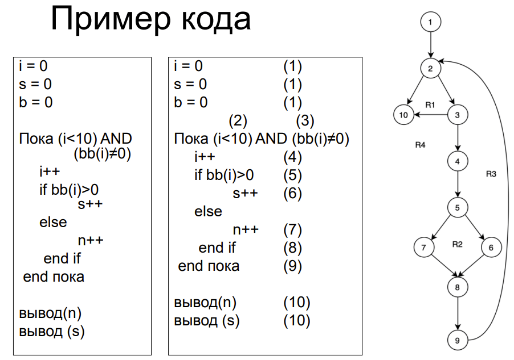
Тестовый вариант см 53

1. Тестирование ПО. Виды тестирования: модульное, интеграционное, регрессионное и системное. Виды системного тестирования. Привести определения и примеры.

См 21

1. Тестирование ПО. Виды тестирования. Тестирование базового пути. Тестирование условий. Примеры и определения. Привести определения и примеры.

Из 49 вопроса и дополнительно. Операционное тестирование (Release Testing). Даже если система удовлетворяет всем требованиям, важно убедиться в том, что она удовлетворяет нуждам пользователя и выполняет свою роль в среде своей эксплуатации, как это было определено в бизнес моделе системы. Следует учесть, что и бизнес-модель может содержать ошибки. Поэтому так важно провести операционное тестирование как финальный шаг валидации. Кроме этого, тестирование в среде эксплуатации позволяет выявить и нефункциональные проблемы, такие как: конфликт с другими системами, смежными в области бизнеса или в программных и электронных окружениях; недостаточная производительность системы в среде эксплуатации и др. Очевидно, что нахождение подобных вещей на стадии внедрения — критичная и дорогостоящая проблема. Поэтому так важно проведение не только верификации, но и валидации, с самых ранних этапов разработки ПО. Приемочное тестирование (Acceptance Testing). Формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью: определения удовлетворяет ли система приемочным критериям и вынесения решения заказчиком или другим уполномоченным лицом принимается приложение или нет.

Тестирование базового пути• Цель: – оценка комплексной сложности ПО => количество тестовых вариантов (для тестирования базового множества путей). Гарантия – однократного выполнения каждого оператора ПО при тестировании;– выполнение каждого условия по True/False ветви. Методика тестирования базового пути• Шаг 1. Создается потоковый граф – Нумерация операторов. – Отображение операторов в граф.• Шаг 2. Цикломатическая сложность– а) V(G) = 4 региона– б) 12-10+2 = 4– в) 3+1 = 4• Шаг 3. Базовое множество независимых путей – а) 1-2-10 (обр. 11-го) – б) 1-2-3-10 (i=0, bb=[0]) – в) 1-2-3-4-5-6-8-9-2… (i>0, bb[i]>0) – г) 1-2-3-4-5-7-8-9-2… (i>0, bb[i]<0)•

Подготовка тестовых вариантов для каждого независимого пути: • а)

– ИД: bb(1,2,3,4,-5, 6,7,-8,9,10, 11)

– ОЖ.РЕЗ.: n = 2, S = 8, i = 10

• б)

– ИД: bb(0)

– ОЖ.РЕЗ.: n = 0, S=0, i = 0

• в)

– ИД: bb (1,2,3,4,-5,0)

– ОЖ.РЕЗ.: n = 1, S = 4, i = 5

• г)

– ИД: bb (-1,-2,-3,4,-5,0)

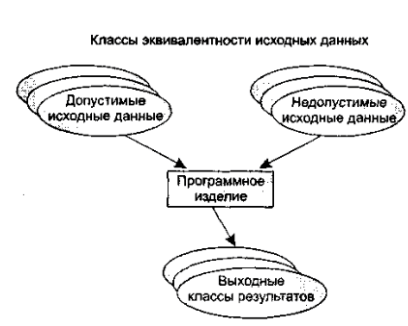
– ОЖ.РЕЗ.: n = 4, S = 1, i = 5

• Существуют независимые пути, являющиеся частью другого пути. Тестирование условий• Цель:– проверка логических условий ПО (охват всех ветвей ПО).• Простое условие (ПУ)– булевы переменные или выражение отношений.• Выражения отношений– E1 <оператор отношения> E2– E1, E2 – арифметические выражения .– Оператор отношения: >,=,<,• Составное условие (СУ) cостоит из объединения:– простых условий;– булевых операторов (AND, OR, NOT);– ( ).• Условие без выражения отношения - это булевоевыражение.Ошибки в условиях:• Виды ошибок:– Булевого оператора(некорректный/отсутствующий/избыточный).– Булевой переменной.– Булевой ( ).– Оператора отношения.– Арифметического выражения.• Тест каждого условия в ПО. Плюсы:– оценка тестового покрытия условия,– генерация дополнительных тестов.• Ошибки в условии => другие ошибки в ПО.Методики тестирования условий• Тестирование ветвей– Для составного условия C проверка:• а) каждое простое условие в нем• б) TRUE (ветви)• в) FALSE (ветви)• Тестирование области определения: – Для E1 <оператор отношения> E2: ⇒ ошибка в операторе ( при корр. E1, E2) | при min(|E1-E2|) n переменных=> 2 в степени n тестов. Эффективно при малом n.

1. Тестирование ПО. Виды тестирования. Тестирование данных. Тестирование циклов. Привести примеры и определения.

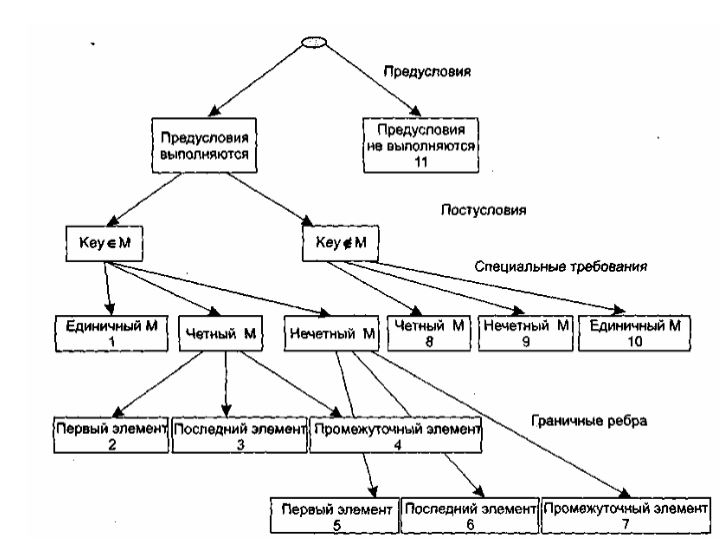
 Тестирование циклов• Виды циклов:– Простые– Вложенные– Объединенные– Неструктурированные Тестирование простых циклов• n – количество повторов.• Выбирают 1 из методов:– Прогон всего цикла,– 1 проход,– 2 прохода,– M<n проходов,– (n-1), n, (n+1) проходов. Тестирование вложенных циклов1. Внутренний цикл – min параметры для остальныхциклов.2. Для внутреннего цикла тесты простого + тесты для исключённых значений и значений за рабочимдиапазоном.3. Следующий объемлющий цикл. Его тест при Min значениях параметров для внешних циклов; типовые значения для вложенныхциклов.4. И так далее, от внутренних к внешним. Тестирование объединенных циклов• Если циклы независимы, то тест простого цикла,• иначе методика вложенных. Тестирование неструктурированных циклов• Не тестируются и,должны быть переделаны.

1. Тестирование ПО. Виды тестирования. Разбиение по классам эквивалентности. Анализ граничных значений. Привести примеры и определения.

Разбиение по эквивалентности• Входная область данных ПО делится на классы эквивалентности.• Для каждого класса эквивалентности – 1тестовый вариант.• Классы эквивалентности:– Набор данных с общими свойствами;– Обрабатываются одинаково;– Для каждого набора из классов эквивалентности один набор операторов/связей ПО.• Классы эквивалентности определяются по спецификации на программу. Виды условий ввода для классов эквивалентности• Определенное значение;• Диапазон значений;• Множество конкретных величин;• Булево условие. Правила формирования классов эквивалентности - 1• Если условие ввода задает конкретное значение a, то определяется один допустимый и два недопустимых класса эквивалентности.– V\_Class={ a };– 2\_Class1={ x | для любого x: x < a};– 2\_Class2={ y | для любого y: y < a}.• Если условия ввода задает диапазон n…m, то определяются один допустимый и два недопустимых по условиям ввода.– V\_Class = {n…m} – допустимый класс эквивалентности;– 2\_Class1={x | x < n} – первый недопустимый класс;– 2\_Class2={x |x > m} – второй недопустимый класс. Правила формирования классов эквивалентности - 2• Если условие ввода задает множество значений{a,b,c}, то определяется один допустимый и один недопустимый класс эквивалентности.– V\_Class={ a,b,c };– 2\_Class={ x | для любого x: (x<>a) & (x<>b) &(x<>c)}.• Если условие ввода задает булево значение, например true, то определяются один допустимый иодин недопустимый класс эквивалентности.– V\_Class={true};– 2\_Class={false}.• Класс эквивалентности => тестовые варианты– Каждый тестовый вариант должен покрывать (проверять)max количество свойств класса эквивалентности.

1. Тестирование ПО. Назначение и цели. Тестовый вариант. Критерии разработки и оценки качества. Функциональное и структурное тестирование. Особенности и возможности. Привести примеры и определения.

Тестирование ПО• Процесс выполнения ПО с целью обнаружения ошибок.• Шаги процесса задаются тестами.• Тест (Тестовый вариант):– набор исходных данных + условия запуска ПО,– набор ожидаемых результатов работы ПО.• Исчерпывающее тестирование– проведение всех возможных тестовых вариантов (все исходные данные, все варианты обработки);– невозможно из-за больших ресурсов.• Хороший тестовый вариант– с большой вероятностью обнаружение новой ошибки.• Успешный тест – нашел новую ошибку.• Цель разработки тестов– систематическое обнаружение различных классов ошибок при min затратах времени и стоимости. Возможности тестирования• Тестирование может:– Найти ошибку;– Установить соответствие функции по ее назначению;– Демонстрировать реализацию требований к характеристикам ПО;– Демонстрировать качество ПО (через надежность).• Тестирование не может:– Доказать отсутствие ошибок.• Много ошибок -> усилить тестирование.• Мало ошибок -> а) надежное ПО; б) плохие тесты.Принципы тестирования• Функциональное – черный ящик– Позднее тестирование, модульное и ПО.– Известно: функции ПО.– Исследование: работа каждой функции на ее области определения.• Структурное – белый ящик– Раннее тестирование, модульное.– Известно: внутренняя структура ПО.– Исследование: внутренние элементы ПО и связи между ними. Структурное тестирование• Объект тестирования:– внутреннее поведение ПО;– корректность построения всех элементов ПО и правильность их взаимодействия;• Анализ управляющих и информационных (реже)потоков.• Характеризуется степенью покрытия логики(исходного кода) ПО.• Исчерпывающее тестирование затруднительно.Особенности тестирования «Белого ящика»• Анализ управляющей структуры ПО.• ПО полностью проверено <= исчерпывающее тестирование путей (маршрутов) ее графа управления.• Тестовые варианты:– Гарантируется проверка всех независимых маршрутов ПО;– Проходятся ветви True, False для всех логических решений.– Выполняются все циклы (в пределах их границ).– Анализируется правильность внутренних структур данных. Результаты «Белого ящика»• Плюсы - Учитывает особенности ошибок:– (Алгоритм → ПО) => синтаксические и семантические ошибки– Предположения о вероятности потока управления/ данных могут быть неправильными => типовой путь слабо проработан.– Количество ошибок min в центре, max на границе ПО;– Результаты ПО могут зависеть не от входных данных, а от внутренних состояний ПО.• Минусы:– Много независимых путей;– Цикл – k (=20) раз; в цикле n (=5) ветвлений → путей = 1014(3170 лет)– Исчерпывающее тестирование не гарантирует выполнение функций ПО;– Могут быть пропущены пути в ПО;– Нельзя найти ошибки, которые зависят от исходных данных.

 Пример тестовых вариантов

• Каждый лист задает отдельный тестовый вариант:

• ТВ1:

– ИД: М = (15), K = 15

– ОР: R = True, I = 1.

• ТВ2:

– ИД: М = (15,20,21,40), K = 15

– ОР: R = True, I = 1.

• ТВ9:

– ИД: М = (15,20,21), K = 100

– ОР: R = False, I = null.

• ТВ11:

– ИД: М = (1,3,2,5,10,7), K = 1

– ОР: сообщение об ошибке.

1. Автоматизированные средства управления проектами, их назначение и возможности. Создание графика, версий и итераций; определение сюжетов и задач; планирование времени и ресурсов; отслеживание состояния проекта; взаимодействие участников проекта, их роли и возможности. Последовательность и состав действий при управлении проектом. Привести примеры и определения.

jazz.net. Система управления проектами нужна для того, чтобы организовать деятельность и сделать работу, учитывая все требования. К примеру, надо решить поставленные задачи, выполнить всё вовремя и ограничиться выделенным бюджетом. Для достижения успехов управляющий должен знать все тонкости проекта, обладать навыком организации работы, решения спорных моментов, сплочения коллектива для достижения общей цели. График сгорания задач (Burndown chart) — Диаграмма, показывающая количество сделанной и оставшейся работы [8.] (см. Рисунок 3). Синим на диаграмме отмечена идеальная линия выполнения задач, на которую и следует опираться, а красным - реальная история выполнения задач. По шкале Y отмечают количество запланированных баллов (как на рисунке), идеальные часы, количество задач и так далее. По шкале X отмечают количество дней до окончания периода выполнения задач. График обновляется ежедневно с тем, чтобы в простой форме показать подвижки в работе над спринтом. График должен быть общедоступен. Существуют разные виды диаграммы: диаграмма сгорания работ для спринта — показывает, сколько уже задач сделано и сколько ещё остаётся сделать в текущем спринте. диаграмма сгорания работ для выпуска проекта — показывает, сколько уже задач сделано и сколько ещё остаётся сделать до выпуска продукта; обычно строится на базе нескольких спринтов. В процессе планирования выпуска элементы переносятся из очереди продукта в выпуск. Владелец продукта проводит исследование продукта, на основании которого выполняет ранжирование сюжетов. Количество помещаемых в выпуск сюжетов зависит от объема работы, который требуется выполнить за запланированное время, которое, в свою очередь, зависит от доступности (занятости) коллектива. План выпуска впоследствии можно пересмотреть и перенести элементы обратно в очередь продукта. Если необходимо, другие элементы можно перенести в выпуск на основании отзыва заинтересованного лица (заказчика). Спринт — итерация в скрам, в ходе которой создаётся функциональный рост программного обеспечения. Жёстко фиксирован по времени. Длительность одного спринта от 2 до 4 недель. В отдельных случаях, к примеру, согласно Scrum стандарту Nokia, длительность спринта должна быть не более 6 недель. Тем не менее, считается, что чем короче спринт, тем более гибким является процесс разработки, релизы выходят чаще, быстрее поступают отзывы от потребителя, меньше времени тратится на работу в неправильном направлении. С другой стороны, при более длительных спринтах команда имеет больше времени на решение возникших в процессе проблем, а владелец проекта уменьшает издержки на совещания, демонстрации продукта и т. п. Разные команды подбирают длину спринта согласно специфике своей работы, составу команд и требований, часто методом проб и ошибок. Для оценки объема работ в спринте можно использовать предварительную оценку, измеряемую в очках истории. Предварительная оценка фиксируется в резерве проекта. На протяжении спринта никто не имеет права менять список требований к работе, внесенном в резерв проекта. Владелец продукта может добавить элементы очереди, называемые сюжетами (Story), непосредственно на вкладке Запланированные элементы. Сюжет (Story) — это задание верхнего уровня, в которое можно записать общие сведения о функции или идее продукта. Деление работы на дорожки в плане выпуска, разбиение сюжетов на меньшие задачи и назначение задач отдельным участникам коллектива. Из резерва проекта выбираются задачи, обязательства по выполнению которых за спринт принимает на себя команда. Предоставьте оценку для задачи, щелкнув на поле Оценка, выбрав единицу времени и указав значение. Для просмотра отчета об объеме работы над дорожкой выполните следующие действия: 5.2.1. В верхней части страницы выберите Общие отчеты. 5.2.2. В разделе Задания выберите Объем работы для создания отчета. Если все участники коллектива обновляют свои задания надлежащим образом, линия стремится к нулю (оставшаяся работа) при завершении работы (см. Рисунок 40). Величина объема работы должна быть видима всем сотрудникам в любое время. Остальное из первых вопросов взять.

1. Автоматизированные средства разработки ПО. Построение моделей требований, анализа, проектирования и реализации (виды моделей и используемые нотации); прямое и обратное проектирование; проверка моделей и согласование моделей; использование профилей и стереотипов. Последовательность и состав действий при разработке ПО. Привести примеры и определения.

17-19 вопросы. Под прямым проектированием имеется в виду генерация кода из содержимой диаграммы, а обратное проектирование – обновление диаграммы в соответствии с изменениями, внесенными в код. Несоответствия в репозитории могут отрицательно сказаться на производительности, что может ощущаться в более сложных операциях, таких как, например, сравнение моделей. Данную функцию проверки целостности следует использовать в основном при наличии общего репозитория, а также в случае локального проекта, разрабатываемого одним человеком. Возможно, только с той разницей, что вероятность несоответствия выше в первом случае.Если у вас произошел сбой импорта XMI, сбой сети или другое непредвиденное событие, нарушающее целостность информации в модели, рекомендуется запустить функцию проверки целостности проекта, чтобы убедиться, что данные вашего проекта структурно завершены. Вы можете выбрать различные элементы для проверки. При проверке целостности проверяются все записи базы данных и гарантируется отсутствие «осиротевших» записей, неточных или неустановленных идентификаторов. Эта функция НЕ проверяет соответствие UML, а только отношения данных и структуру репозитория. Вы можете сначала запустить средство проверки целостности в режиме отчета, чтобы узнать, нужно ли что-то исправить, а затем снова запустить его в режиме «восстановить/очистить». Когда Enterprise Architect «очищает» модель, он пытается восстановить потерянные пакеты и элементы и создает новый пакет на корневом уровне модели с именем \_recovered\_. Проверьте все найденные элементы и, при необходимости, перетащите их обратно на место. Если они не нужны, удалите их. Стереотипы позволяют проектировщикам расширять словарь UML для создания новых элементов моделирования, получаемых из существующих, но имеющих определенные свойства, которые подходят для конкретной проблемы предметной области или для другого специализированного использования.

1. Автоматизированные средства тестирования ПО. Виды автоматического тестирования: модульное, нагрузочное, веб- серверное, пользовательского интерфейса; задание исходных параметров тестирования, результаты тестов и их интерпретация; покрытие кода. Привести примеры и определения.

Модульное тестирование получило такое название, так как функции программы разбиваются на отдельные тестируемые участки поведения, которые можно протестировать в качестве отдельных модулей. Нагрузочное тестирование - вид тестирования производительности, сбор показателей и определение производительности и времени отклика программно-технической системы или устройства в ответ на внешний запрос с целью установления соответствия требованиям, предъявляемым к данной системе (устройству). Для веб-тестов производительности предлагается использовать вкладку Performance в Chrome DevTools (см. рисунок 15). Панель отображает линию времени использования сети, выполнения JavaScript кода и уровень загрузки памяти. После первоначального построения графиков линии времени, будут доступны подробные данные о выполнение кода и всем жизненном цикле страницы. Будет возможность ознакомится с временем исполнения отдельных частей кода, появится возможность выбрать отдельный промежуток на временной шкале и ознакомится с тем какие процессы происходили в этот момент. Тестирование пользовательского интерфейса (GUI Testing) - функциональная проверка интерфейса на соответствие требованиям — размер, шрифт, цвет, consistent behavior. Тестовый сценарий, тестовый вариант, тест (Test Case) — это артефакт, содержащий исходные данные и ожидаемый результат при тестировании. Он также описывает совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части. Тесты разделяются по ожидаемому результату на положительные и отрицательные:  Положительный тест использует только корректные данные и проверяет, что приложение правильно выполнило вызываемую функцию.  Отрицательный тест оперирует как корректными, так и некорректными данными (минимум 1 некорректный параметр) и ставит целью проверку исключительных ситуаций (срабатывание валидаторов), а также проверяет, что вызываемая приложением функция не выполняется при срабатывании валидатора. Чтобы определить, какая часть кода проекта в действительности тестируется закодированными тестами, такими как модульные тесты, можно воспользоваться возможностью покрытия кода в Visual Studio. Для обеспечения эффективной защиты от ошибок тесты должны выполнять ("покрывать") большую часть кода.

1. Автоматизированные средства ведения версий. Возможности по работе с локальным и удаленным репозиторием; версии и фиксации; ветки и работа с ними; слияния и разрешения конфликтов; варианты отката изменений; теги и комментирование; возможности по просмотру истории изменений. Последовательность и состав действий при работе с локальным и удаленным хранилищем. Привести примеры и определения.

 git init

Эта команда используется для создания GIT репозитория. Пример

использования:

git init

 git add

Команда git add может быть использована для добавления файлов в индекс. К примеру, следующая команда добавит файл под названием temp.txt присутствующий в локальном каталоге в индекс:

git add temp.txt

 git clone

Команда git clone используется для клонирования репозитория. Если репозиторий находится на удаленном сервере, используется команда такого рода:

git clone имя.пользователя@хост:/путь/до/репозитория

И наоборот, для клонирования локального репозитория используйте:

git clone /путь/до/репозитория

 git commit

Команда git commit используется для коммита изменений в файлах проекта. Обратите внимание, что коммиты не сразу попадают на удаленный репозиторий.

Применение:

git commit –m “Сообщение идущее вместе с коммитом”

 git status

Команда git status отображает список измененных файлов, вместе с файлами, которые еще не были добавлены в индекс или ожидают коммита.

 git push

git push еще одна из часто используемых git команд. Позволяет поместить изменения в главную ветку удаленного хранилища связанного с рабочим каталогом.

Например:

git push origin master

Команда, обновляющая удаленную ветку с обновлением всей истории, не смотря на расхождения.

git push -f

Команда, которая обновляет удаленные ссылки с помощью локальных ссылок, одновременно отправляя объекты, необходимые для выполнения заданных ссылок с созданием удаленной ветки.

git push –set-upstream <remote> <branch>

 git checkout

Команда git checkout может быть использована для создания веток или переключения между ними. К примеру, следующий код создаст новую ветку и переключится на нее:

command git checkout -b <имя-ветки>

Чтобы просто переключиться между ветками используйте:

git checkout <имя-ветки>

 git remote

Команда позволяет пользователю подключиться к удаленному репозиторию. Данная команда отобразит список удаленных репозиториев, настроенных в данный момент:

git remote –v

Эта команда позволит пользователю подключить локальный репозиторий к

удаленному серверу:

git remote add origin <адрес.удаленного.сервера>

 git branch

Команда git branch может быть использована для отображения, создания или удаления веток. Для отображения всех существующих веток в репозитории введите:

git branch

Для удаления ветки:

git branch –d <имя-ветки>

 git pull

Команда pull используется для объединения изменений, присутствующих в

удаленном репозитории, в локальный рабочий каталог.

 git merge

Команда git merge используется для объединения ветки в активную ветвь.

Применение:

git merge <имя-ветки>

 git tag

Используется для маркировки определенных коммитов с помощью простых

меток. Примером может быть эта команда:

git tag 1.1.0 <вставьте-commitID-здесь>

 git log

Запуск команды git log отобразит список всех коммитов в ветке вместе с

соответствующими сведениями. Пример результата:

commit 15f4b6c44b3c8344caasdac9e4be13246e21sadw

Author: Alex Hunter <alexh@gmail.com>

Date: Mon Oct 1 12:56:29 2016 -0600

 git reset

Команда git reset используется для сброса индекса и рабочего каталога до

последнего состояния коммита. Применение:

git reset --hard HEAD

 git revert

Команда git revert совершается переход к указанному коммиту, обращаются его изменения и создается новый, «обратный» коммит. Затем позиция указателей обновляется — они перемещаются к этому коммиту в конце ветки.

 git stash

Возможно одна из самых малоизвестных команд git. Она помогает в сохранении изменений на временной основе, эти изменения не попадут в коммит сразу.

 git show

Для просмотра информации о любом git объекте используйте команду git show.

 git fetch

git fetch позволяет пользователю доставить все объекты из удаленного

репозитория, которые не присутствуют в локальном рабочем каталоге. Пример

применения:

git fetch origin

 git ls-tree

Команда git ls-tree используется для просмотра дерева объекта вместе с

названием, режимом каждого предмета и значением SHA-1. К примеру:

git ls-tree HEAD

 git rebase

Команда git rebase используется для применения коммитов в другой ветке.

Например:

git rebase master

Продолжить операции на следующий commit.

git rebase –continue

Остановить операцию с возвращением в предыдущее состояние.

git rebase –abort

1. Автоматизированные средства создания документации. Принципы и возможности автоматического составления документации; возможности по комментированию исходных кодов: проекты, модули, файлы, классы и их компоненты; параметры комментариев; настройки генератора документации. Привести примеры и определения.

Для создания документации достаточно просто писать комментарии в коде, придерживаясь нескольких простых правил.

Использовать Doxygen просто – для этого надо просто запустить программу, указав ей путь к файлу с настройками. Файл с настройками представляет собой текстовой файл, который можно редактировать как в текстовом редакторе, так и с помощью специальных программ, например Doxygate. В настройках описывается внешний вид документации, какие сущности и отношения между ними следует включать в нее, имя проекта, путь к анализируемым файлам и так далее.

Комментирование в стиле Doxygen:

Doxygen поддерживает несколько стилей комментариев, например:

/\*\*

Комментарии

\*/

Обратите внимание, что последовательность символов /\*\* сообщает программе, что начинается комментарий предназначенный для нее. Начиная с этого места и до завершающих символов \*/ следуют комментарии.

Есть несколько директив, которые помогают Doxygen составить грамотную документацию. Вот основные:

@brief

\brief Краткое описание комментируемой сущности.

Пример:

@brief Функция для поиска пользователя в базе данных

@param

\param Параметры передаваемые функции.

Пример:

@param name Имя пользователя

@return

\return Возвращаемое функцией значение.

Пример:

@return Информация о пользователе

@throw

\throw Исключения выбрасываемые функцией.

Пример:

@throw DatabaseError Если произошла ошибка при подключении

к базе данных

Вместо собачки @ можно использовать слэш \.

Пример комментариев для класса:

/\*\*

@brief Класс для работы с базой данных

Осуществляет подключение к базе при создании

и закрывает соединение при уничтожении

\*/

class Database

{

public:

/\*\*

@brief Конструктор

@param connectionString Строка подключения к базе данных

@throw ConnectionError Если подключение не удалось

\*/

explicit Database(const std::string& connectionString);

/\*\*

@brief Поиск пользователя в базе данных

Данная функция делает выборку из базы данных

по имени пользователя и возвращает структуру с информацией

о нем. Ожидается, что соединение с базой данных установлено

и пользователь существует

@param name Имя пользователя

@return Информация о пользователе

@throw DatabaseError Если произошла ошибка при подключении

к базе данных

@throw InvalidRequest Если пользоваделя в базе не существует

\*/

CustomerPtr GetCustomer(const std::string& name);

};

Для разделения краткой и детальной информации можно использовать пустую строку (см. выше) или запись вида:

/// Класс для работы с базой данных

/\*\* Осуществляет подключение к базе при создании

и закрывает соединение при уничтожении \*/

первая строка будет краткой информацией, остальные – детальной.

Комментарии указываются перед объектам, которые описывают.

Для написания комментария после объекта (на той же строке) используют запись вида:

Int number; ///< Комментарий

Для комментирования на уровне проекта, а не файла служат следующие директивы:

@mainpage Основная страница проекта, то с чего начинается просмотр документации.

Пример:

@mainpage Приложение для учета пользователей

@page Дополнительные страницы проекта. Я рассматриваю их как логически обособленные части проекта.

Пример:

@page Database Database

Утилиты для работы с базой данных

@ref Ссылки на страницы проекта, с их помощью можно организовать ссылки с главной страницы проекта на страницы подпроектов.

Пример:

@ref Database Утилиты для работы с базой данных

Для организации четкой структуры я рекомендую в каждом подпроекте создавать файл description.h в котором будет директива @page, описание для чего нужен этот подпроект и принципы работы с ним.

В свою очередь в корне проекта я также создаю файл description.h в котором немного рассказано о проекте в целом и приведены ссылки на его части:

/\*\*

@mainpage Приложение для учета пользователей

Состоит из следующих частей:

- @ref Database Утилиты для работы с базой данных

\*/

Чтобы привести фрагмент кода (например пример работы с классом) используется конструкция @code – @endcode:

/\*\*

Пример использования Foo:

@code

Foo f();

f.Run();

@endcode

\*/

Списки можно создавать с помощью символа - (минус). Пример:

/\*\*

Список:

- Первый пункт

- Второй пункт

\*/

Перечисления я оформляю так:

/\*\*

@brief Режимы устройства

\*/

enum Mode

{

Mode\_On, /\*\*< Включено \*/

Mode\_Off /\*\*< Выключено \*/

};

Можно указать в первом окне настроек (вкладка Project):

- путь к рабочему каталогу (например, каталог пользователя),

- название проекта,

- путь к исходному коду (папка с исходниками проекта),

- флаг рекурсивного сканирования,

- путь к создаваемой документации.

Указать на вкладке Mode:

- Язык программирования проекта.

Указать на вкладке Output:

- Формат выходных документов (простой HTML и RTF).

Перейти к панели Expert и в ее вкладках указать:

- Выходной язык

- Кодировка