ППС

13 февраля 2015 г.

15:14

**Программный проект** -- временное предприятие, предназначенное для создания уникальных продуктов, услуг и результата.

**Программная инженерия** -- система инженерных принципов для создания экономичного ПО, которая надежно и эффективно работает в реальных компьютерах (старое определение).

**Программная инженерия** -- систематическое применение научных и технологических знаний, методов и практического опыта к проектированию, реализации, тестированию и документированию ПО, в целях оптимизации его производства, поддержки и качества (новое определение).

Различают методы, средства и процессы программной инженерии.

**Методы** обеспечивают решение широкого круга задач разработки ПО (планирование и оценка программного проекта, анализ требований, проектирование структур программ, тестирование, сопровождение ПО..).

**Средства** ПИ обеспечивают автоматизированную или автоматическую поддержку методов.

Литература этой лекции:

* 1. «Технология разработки ПО», Орлов

20.02

# Сложность программных систем

Почему ПО присуща сложность:

* 1. Сложность реальной предметной области из который исходит заказ на разработку.
  2. Трудность управления процессом разработки.
  3. Необходимость обеспечить достаточную гибкость программ.
  4. Неудовлетворительные способы описания поведения больших дискретных систем.

# Структура сложных систем

Признаки сложных систем:

* 1. Сложные системы часто являются иерархическими и состоят из взаимосвязанных подсистем, которые в свою очередь могут быть поделены на подсистемы.
  2. Внутрикомпонентные связи обычно сильнее, чем межкомпонентные. Это позволяет нам отделять высокочастотные взаимодействия внутри компонентов от низкочастотной динамики между компонентами.
  3. Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных.
  4. Выбор какие компоненты в данной системе считаются элементарными относительно произволен и в большей степени оставляется на усмотрение исследователя.
  5. Любая работающая сложная система является результатом развития работавшей более простой системы.

Литература этой лекции:

* 1. Гради Буч, «Объектно-ориентированный анализ и проектирование»
  2. Саймон

# Архитектура ПО

**Программная архитектура** — это структура её структур, то есть изложение ее программных элементов, их внешних свойств и установленных между ними отношений.

Внешними свойствами называются те предположения, которые сторонние элементы могут выдвигать в отношении данного элемента, в частности они касаются предоставляемых элементом услуг, рабочих характеристик, устранения неисправностей, совместного использования ресурсов и так далее.

* 1. Архитектура определяет программные элементы. Архитектура это абстракция системы, в которой отсутствует информация об элементах не имеющая отношения к тому как они используются, соотносятся или взаимодействуют с другими элементами.
  2. В состав любой системы может входить и входит целый ряд структур. А следовательно ни одной отдельно взятой структуры, которую можно было бы уверенно назвать архитектурой не существует.
  3. Любую программную систему можно представить как совокупность элементов.
  4. Если поведение отдельно взятого элемента можно зафиксировать или выявить с точки зрения других элементов, то это поведение входит в состав архитектуры.

## Архитектурные образцы, эталонные модели и эталонные варианты архитектур.

**Архитектурный образец** — это описание типов элементов и отношений, изложение ряда ограничений на их использование.

**Эталонная модель** — разделение между отдельными блоками функциональных возможностей и потоков данных.

**Эталонная архитектура** — это результат отображения функциональных схем из эталонной модели на архитектурный образец (программные элементы) и потоки данных между ними.

Автоматически созданный замещающий текст:
丶 4 一 丶 丶 
∕ 

## Почему программная архитектура так важна?

* 1. Архитектура -- это единый язык взаимодействия.
  2. Архитектура -- это начальное проектное решение.
  3. Архитектура -- это переносимая абстракция системы.

Lean

Kanban, как альтернатива Scrum

Доски, отличия от скрам, почему нет таймфреймов, на что делается фокусировка

# Жизненные циклы ПО

Первое появление в 1970 году. Уинстон Райс предложил каскадную модель.

## Каскадная модель

Этапы:

* 1. Планирование
     1. Моделирование
     2. Анализ требований
  2. Проектирование
     1. Архитектура ПО
     2. Структура ПО
     3. Входные и выходные интерфейсы
  3. Кодирование
  4. Тестирование
  5. Сопровождение

Недостатки:

* 1. Реальные проекты часто требуют отклонения от стандартной последовательности шагов.
  2. Результаты доступны только в конце разработки.
  3. Цикл основан на точной формулировке начальных требований.

Преимущества:

* 1. Четкая последовательность действий.
  2. Заказчик получает точное время.

# Стратегии разработки ПО

* 1. Однократный подход (водопадная или каскадная стратегия).
  2. Инкрементальная стратегия. Как каскадная, но разработка идет частями, хотя все требования известны изначально.
  3. Эволюционная стратегия. Как инкрементная, но с риском.

# Спиральная модель

Автоматически созданный замещающий текст:


Этапы:

* 1. Сбор требований, анализ проекта
  2. Корректировка требований заказчиком
  3. Анализ риска на основе начальных требований
  4. Анализ риска на основе реакции заказчика
  5. Переход к комплексной системе
  6. Начальный макет системы
  7. Следующий уровень макета
  8. Сконструированная система
  9. Оценивание заказчиком

# Тяжеловесные и облегчённые процессы

Agile (легковесные)

Манифест гибкой разработки ПО:

* 1. Люди и их взаимодействие внутри команды важнее процессов и инструментов.
  2. Работающее ПО важнее исчерпывающей документации.
  3. Сотрудничество с заказчиком важнее согласований условий контракта.
  4. Готовность к изменениям вроде следования первоначальному плану.

Принципы:

* 1. Удовлетворение клиента за счёт ранней и бесперебойной поставки ценного ПО.
  2. Частая поставка ПО.
  3. Тесное и ежедневное общение с заказчиком.
  4. Проектом занимаются мотивированное личности, которые обеспечены нужными условиями работы и печеньем.
  5. Рекомендуемый метод передачи информации -- личный разговор.
  6. Рабочее ПО лучший измеритель прогресса.
  7. Заказчики, разработчики и пользователи должны поддерживать темп на неопределенный период.
  8. Постоянное внимание технического мастерства и удобство дизайна.
  9. Простота -- это искусство не делать лишней работы.
  10. Лучшие технические решения, архитектура и дизайн получаются у самоорганизующейся команды.
  11. Постоянная адаптация к изменяющимся обстоятельствам.

XP:

* 1. Разработка через тестирование

20.03

# Требования ПО

Требования -- это возможности или условия, которыми должна соответствовать система или проект.

Классификация требований:

* 1. Функциональные -- это требования, которые описывают поведение системы и ее функции. Если функциональные требования оформлены как пользовательские, они, как правило, описывают систему в обобщённом виде. В противоположность этому, функциональные требования оформленные как системные, описывают систему максимально подробно, включая ее функции, исключения и т. д.

Спецификация функциональных требований должна быть комплексной и непротиворечивой, но на практике этого добиться сложно. Причем зачастую выявить несогласованность на начальных этапах очень трудно.

* 1. Нефункциональные требования -- описывают характеристики системы и ее окружение, а не ее поведение. В них обычно приводится перечень ограничений, накладываемых на действия и функции, выполняемые системой (загруженность процессора и т. д.). Многие нефункциональные требования относятся к системе в целом, а не к отдельным ее функциям. Это означает, что они более значимы и критичны, чем отдельные функциональные требования. Ошибка, допущенная в функциональных требованиях может снизить качество системы, а в нефункциональных может привести к неработоспособности системы.

Нефункциональные требования можно разбить на следующие группы:

* 1. Требования к продукту.
  2. Организационные требования.
  3. Внешние требования.

* 1. Требования к предметной области -- характеризуют ту предметную область где будет функционировать система. Могут быть функциональными и нефункциональными.

Уровни описания требований:

* 1. Пользовательские. Должны описывать функциональные и нефункциональные требования так, что они были понятны даже пользователю, не имеющему специальных технических знаний.

* 1. Функциональные.

# Описание требований в контексте модели прецедентов

Прецеденты -- это механизм упрощения этапа формулировки требований для всех заинтересованных лиц. По существу это рассказы об использовании системы в процессе решения поставленной задачи.

# Прецеденты и ощутимый результат

Исполнитель -- сущность, обладающая поведением, например, человек, программа или организация.

Сценарий -- специальная последовательность действий или взаимодействий между исполнителем и системой (экземпляр прецедента). Это один конкретный сценарий использования системы, либо один проход прецедента.

Неформальное определение прецедента -- это набор взаимосвязанных, успешных и неудачных сценариев, описывающих использование системы исполнителем, для решения одной из задач.

Прецедент -- это набор сценариев использования, в котором каждый экземпляр сценария представляет собой последовательность действий, выполняемых системой для достижения ощутимого для конкретного исполнителя результата.

# Типы и форматы прецедентов

## Прецеденты типа «черный ящик»

## Степень формализации

* 1. Сжатые. Аннотация в виде одного абзаца. Обычно описывает только главный успешный сценарий.
  2. Свободный. Неформальное описание прецедента. Описывается в нескольких абзацах, охватывает несколько сценариев.
  3. Развернутый. Наиболее подробный стиль описания, при котором описываются все возможные варианты событий. Предусловия, постусловия и т. д.

## Пояснения к прецедентам

* 1. Вводные элементы (открыта страница такая-то).
  2. Заинтересованные лица и их потребность.
  3. Основной успешный сценарий.
  4. Расширения (альтернативные сценарии).
  5. Специальные требования.

## Задачи и рамки прецедента

В процессе анализа требований к приложению следует сосредоточиться на уровне элементарных бизнес-процессов.

Элементарный бизнес-процесс (EBP) -- это задача, выполняемая одним человеком в одном месте, в одно время в ответ на некоторое событие, добавляющая измеряемое бизнес-значение (ценность) и переводящая данные в некоторое устойчивое состояние.

## Определение основных исполнителей

Для выделения прецедентов используется следующая процедура:

* 1. Определяются рамки системы
  2. Идентифицирование основных исполнителей, потребности которых удовлетворяются с помощью системы. При выявлении этих исполнителей задаются следующие вопросы:
     1. Кто запускает систему?
     2. Кто осуществляет управление пользователями и безопасностью?
     3. Кто контролирует деятельность и производительность системы?
     4. Как выполняется обновление ПО?
  3. Для каждого исполнителя определяется его задача.
  4. Определяются прецеденты, удовлетворяющие потребностям каждого исполнителя и присылаются им имена соответствующих задач.

Роберт Мартин.

# Гибкое проектирование

# Ароматы дизайна

* 1. Жесткость -- это характеристика программы, затрудняющая внесение в нее изменений, даже самых простых. Дизайн жесткий, если единственное изменение вызывает целый каскад других изменений в зависимых модулях. Чем больше модулей приходится изменять тем жёстче дизайн.
  2. Хрупкость -- свойство программы повреждаться во многих местах при внесении единственного изменения. Зачастую проблемы возникают в частях программы, которые не связаны напрямую с изменяемыми.
  3. Косность. Дизайн является косным, если он содержит части, которые могли бы оказаться полезными в других системах, но усилие и риски, сопряжённые с попыткой отделить эти части от оригинальной системы слишком велики.
  4. Вязкость. Бывает двух видов: вязкость программы и вязкость окружения.
  5. Ненужная сложность.
  6. Ненужные повторения.
  7. Непрозрачность -- это трудность модуля для понимания.

# Принципы SOLID

## Принцип единственной обязанности

Том Демарк, 1979.

«У класса должна быть только одна причина для изменения»

Автоматически созданный замещающий текст:


Автоматически созданный замещающий текст:


Стоит ли разделять эти обязанности? Если модификация подразумевает изменения сигнатуры методов управления соединением, то дизайн начинает попахивать жесткостью, т. к. классы, вызывающие Send и Recv придется повторно компилировать и развертывать чаще, чем хотелось бы.

## Принцип открытости-закрытости

Бертен Мейер

«Программные сущности: классы, модули, функции и т. д. должны быть открыты для расширения, но закрыты для модификации»

Описание принципа:

У модулей, согласованных с этим принципом, есть две основные характеристики:

* 1. Открыты для расширения. Это означает, что поведение модуля можно расширить. Когда требования к приложению изменяются мы добавляем в модуль новое поведение, отвечающее изменившимся требованиям. Иными словами, мы можем изменить состав функций модуля.
  2. Закрыты для модификации. То есть расширение поведения модуля не сопряжено с изменениями в исходном коде модуля, то есть бинарное представление остается неизменным.

# Принцип подстановки Лисков (LSP)

Должна быть возможность вместо базового типа подставить любой его подтип.

Если для каждого объекта О1 типа С существует объект О2 типа Т, такой, что для любой программы П типа Т поведение П не изменяется при замене О1 на О2.

Статья Барбары Лисков: «Абстракции данных и иерархии».

# Принцип инверсии зависимости

* 1. модули верхнего уровня не должны зависеть от модулей нижнего уровня; и те и другие должны зависеть от абстракций.
  2. абстракции не должны зависеть от деталей; детали должны зависеть от абстракций.

# Принцип разделения интерфейсов

Клиенты не должны вынуждено зависить от методов, которыми не пользуются

# Паттерны проектирования

Любой паттерн описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем, таким образом, что это решение можно использовать миллион раз, ничего не изобретая заново.

Паттерн состоит из нескольких частей:

* 1. Имя
  2. Задача
  3. Решение задачи
  4. Результаты

# Порождающие паттерны

* 1. Абстрактная фабрика (интерфейсы в джава-приложениях для мак, вин и лин)
  2. Строитель (конвертер ртф куда-либо)
  3. Фабричный метод (создание документов ворд, эксель, пп)
  4. Прототип (нотный стан)

# Структурные паттерны

* 1. Адаптер (фигуры, библиотека текствью)
  2. Компоновщик (флэппи берд: птица, труба, бэкграунд и лэйаут)
  3. Декоратор (текствью, бордер, скролл)
  4. Мост (окна в системах; сервисы и использование для разных протоколов)
  5. Фасад (много библиотек, какие-то нужно объединить)
  6. Посредник (ограничить доступ к новостям, через прокси)

# Паттерны поведения

* 1. Цепочка обязанностей (клик по кнопке вызывает хелп кнопки -> панели -> окна)
  2. Команда (обработчики на чем-либо)
  3. Итератор (список пользователей, как студентов и пользователей и итераторы по ним)
  4. Медиатор (форма с разными типами фильтрации)
  5. Хранитель (undo, redo для текста)
  6. Наблюдатель

DDD -- domain drive design

DDD -- набор принципов и схем, помогающих разработчикам создавать изящные системы объектов. При правильном применении оно приводит к созданию программных абстракций, которые называются моделями предметных областей.

DDD -- не парадигма.

Принципы:

* 1. Единый язык для всей команды
  2. Контекст
  3. Value objects (Объекты значений)
  4. Сущности
  5. Сервисы

# Антипаттерны

* 1. Божественный объект (blob)
  2. Волшебная кнопка
     1. Button1, ВпереТ!!!
     2. Код в обработчике
  3. Раздувание интерфейсов
  4. Золушкина туфелька
  5. Базовый класс-утилита
  6. Полтергейст
  7. Одиночество
  8. Лодочный якорь
  9. Magic numbers