Лекция

Введение в Компонентно-Ориентированное Проектирование

Процесс разработки

- *“C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.”*

Bjarne Stroustrup

Процесс разработки

- Анализ исходной задачи
- Создание общего проекта
- Документация
- Тестирование
- Поддержка

По сложности превосходят

- Написание кода
- Отладку кода

Процесс разработки

- Уметь фокусироваться на отдельных частях
- Уметь рассмотреть проблему в целом

- Методы ООП
- Абстракция данных

Трудно воплотить идеи в реальной практике

- **Самое важное** – понимать, что вы хотите сделать
- **Успешный процесс** разработки – длительный процесс
- Разрабатываемые системы **всегда** эволюционируют в направлении предела сложности
- Ничто не может заменить **интеллект**, **опыт** и **вкус**
- **Экспериментирование** неотъемлемая часть разработки
- Проектирование и программирование – процессы **итеративные**
- Невозможно чётко **отделить фазы**: проектирования, программирования и тестирования

Сложность

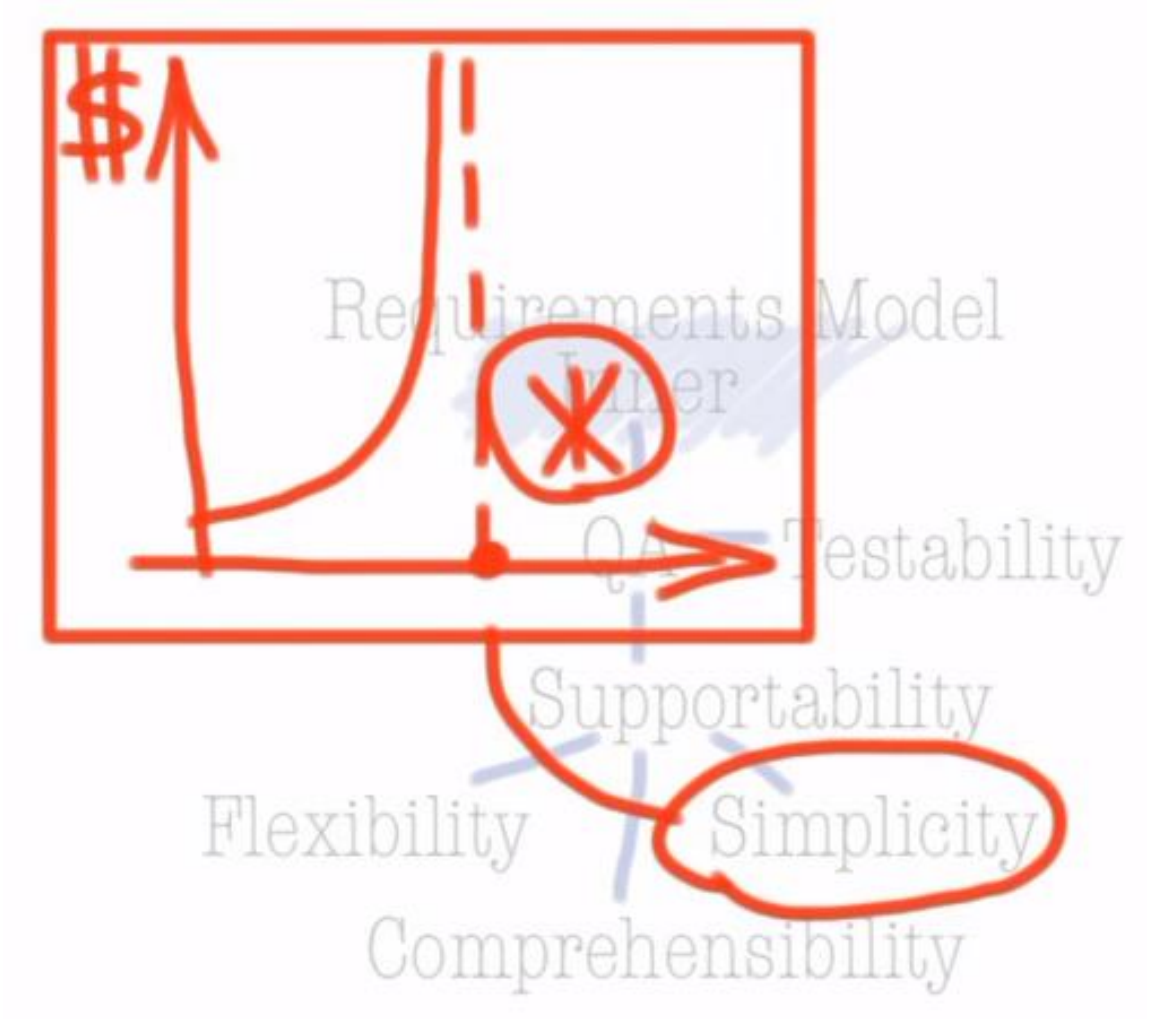
- *“If you think it's simple, then you have misunderstood the problem*.”

Bjarne Stroustrup

Сложность

- Самая большая проблема (в разработке ПО) – **сложность**.
- В *борьбе со сложностью* есть одно универсальное правило -- "*разделяй и властвуй*"
- Задачу, которую удалось корректно разделить на 2 относительно независимые подзадачи - можно считать, более чем на половину решенной.

Важность простоты



Вечная борьба со сложностью

Выделение в задаче ясных

- Модулей или компонент
- Сущностей или классов
- Отделение реализации от интерфейса

Выявление всех действий с чётко определёнными каналами взаимодействия

Любое разделение выполняется значительно легче, чем из эффективное взаимодействие

Вечная борьба со сложностью

- *“I have always wished for my computer to be as easy to use as my telephone; my wish has come true because I can no longer figure out how to use my telephone.”*

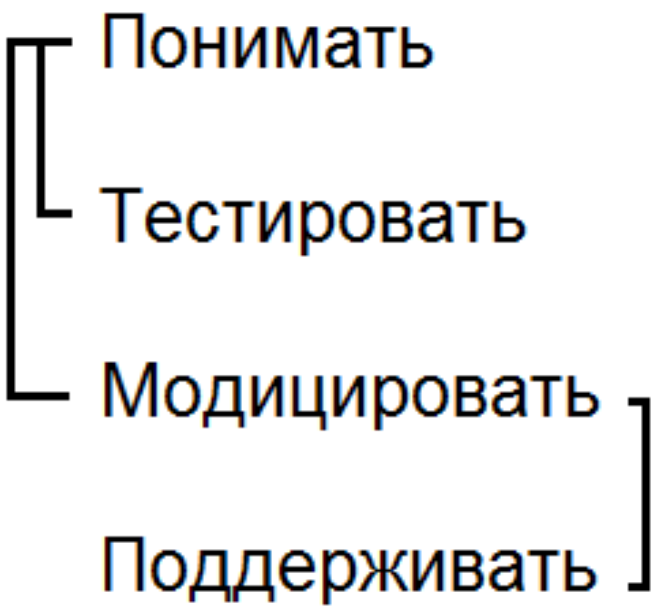
Bjarne Stroustrup

Важны ли

- внутренняя структура программы

и

- качество коллектива разработчиков?



Последовательность строго определённых шагов

Входные данные
+
Преобразования

=

Выходные результаты

Последовательность строго определённых шагов

Входные данные

+

Преобразования

+

(*Человеческий фактор*)

=

Выходные результаты

Проектирование

- “*Design and programming are human activities; forget that and all is lost*.”

Bjarne Stroustrup

Проектирование

- *“Design and programming are human activities; forget that and all is lost.”*

Bjarne Stroustrup

Проектирование – это человеческая деятельность, забыв про это, ничего не выйдет

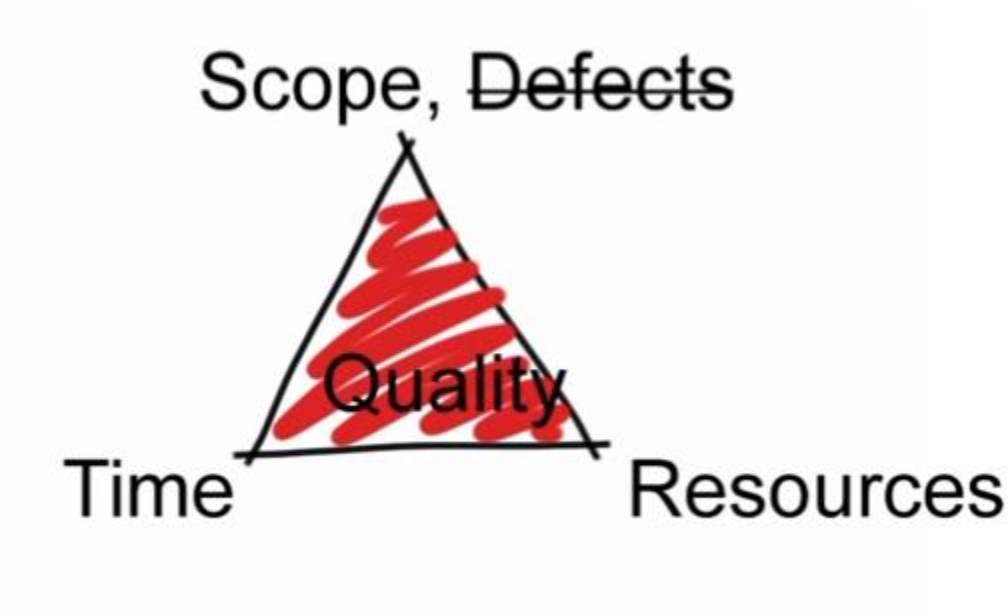
Проектирование

- Без дисциплины никуда
  - планирование
  - специализация
  - формальные связи
  - рекомендации форматирования и тестирования

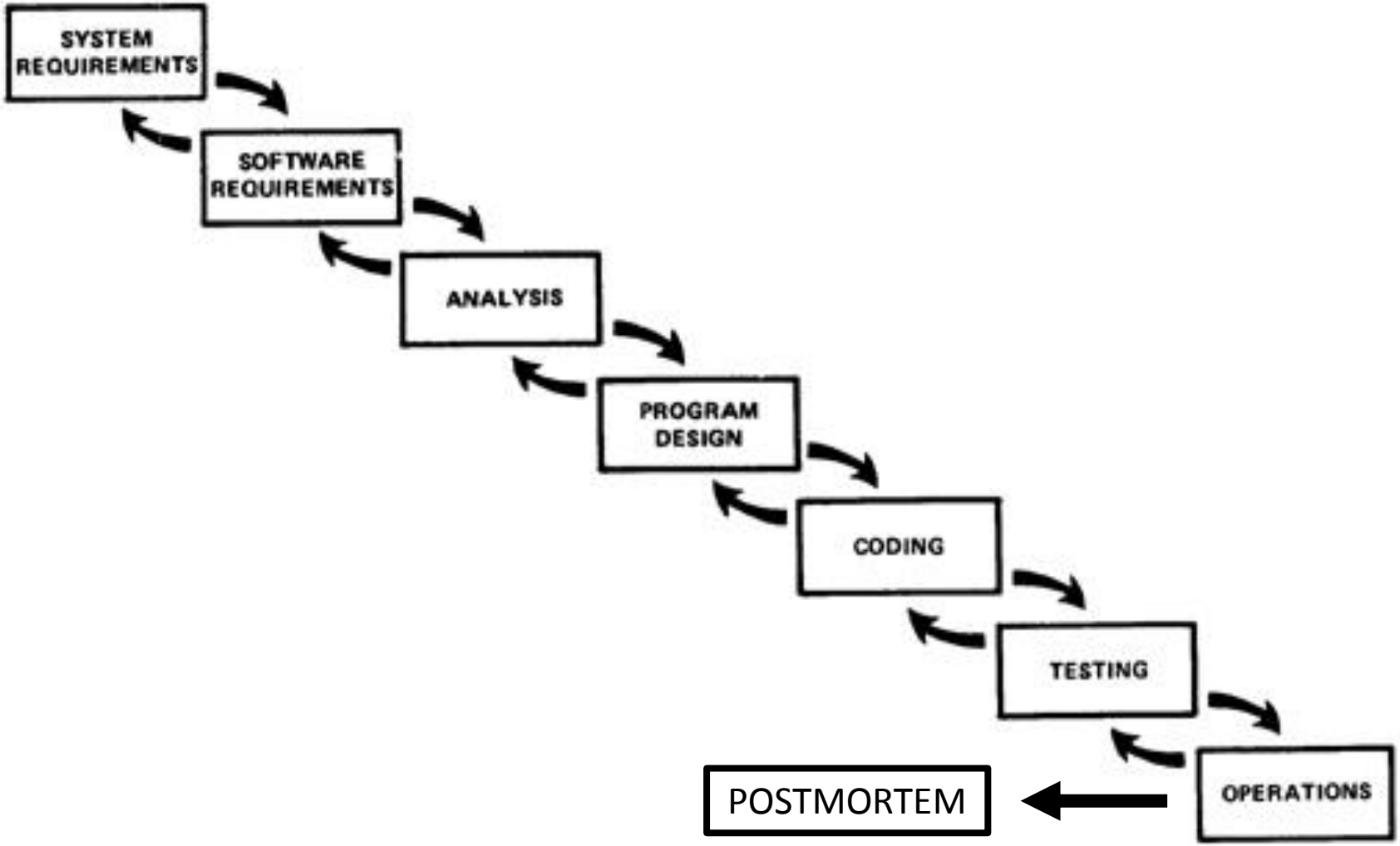
Проектирование

- Вопросы проектирования систем на пределе возможностей коллектива разработчиков на последней грани их ресурсов – менее амбициозные проекты слабо нуждаются в подобном
- Если проект полностью для вас определён и понятно как его выполнять – **браться за него не стоит**

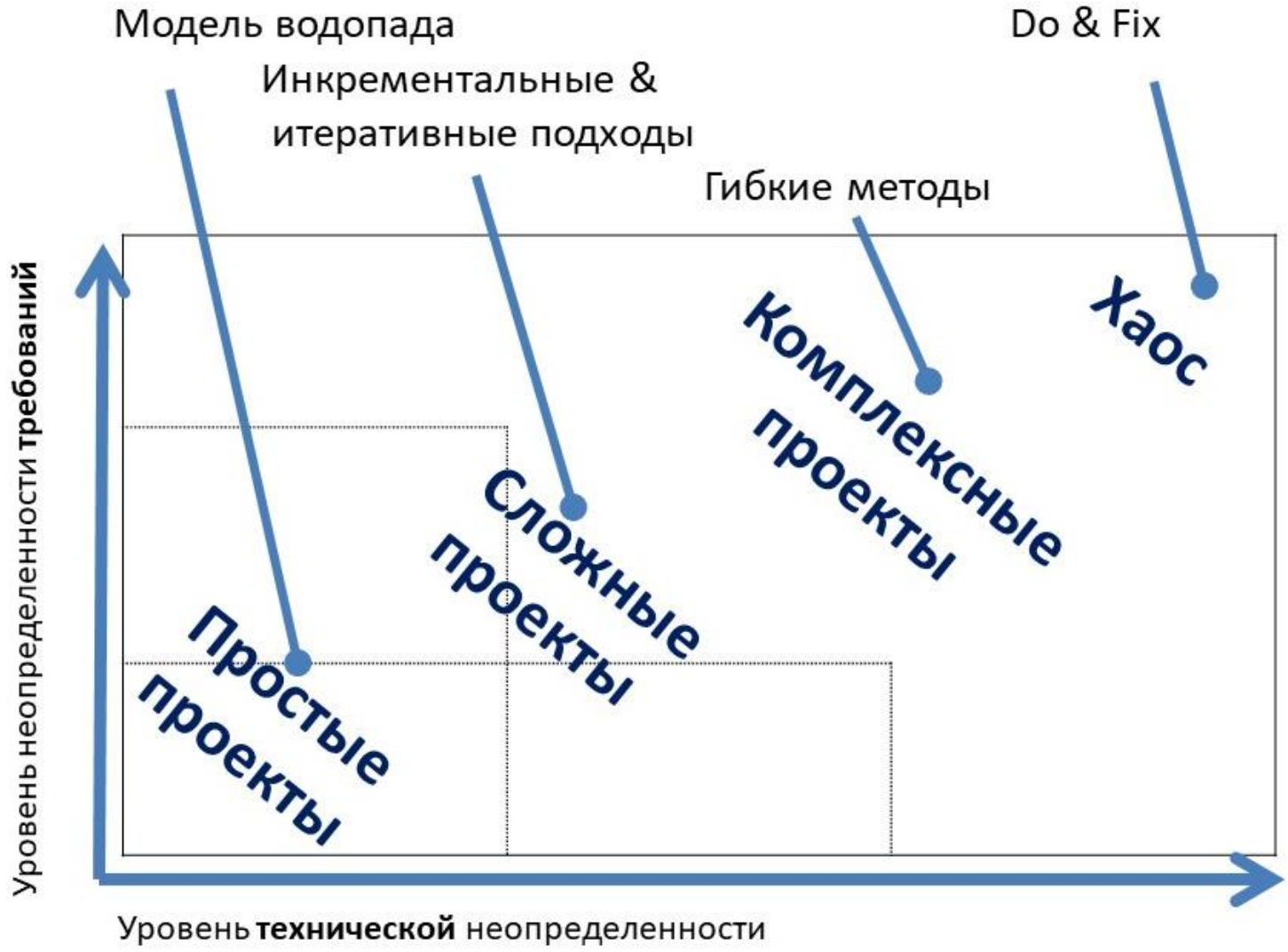
Решение задачи в ограничениях

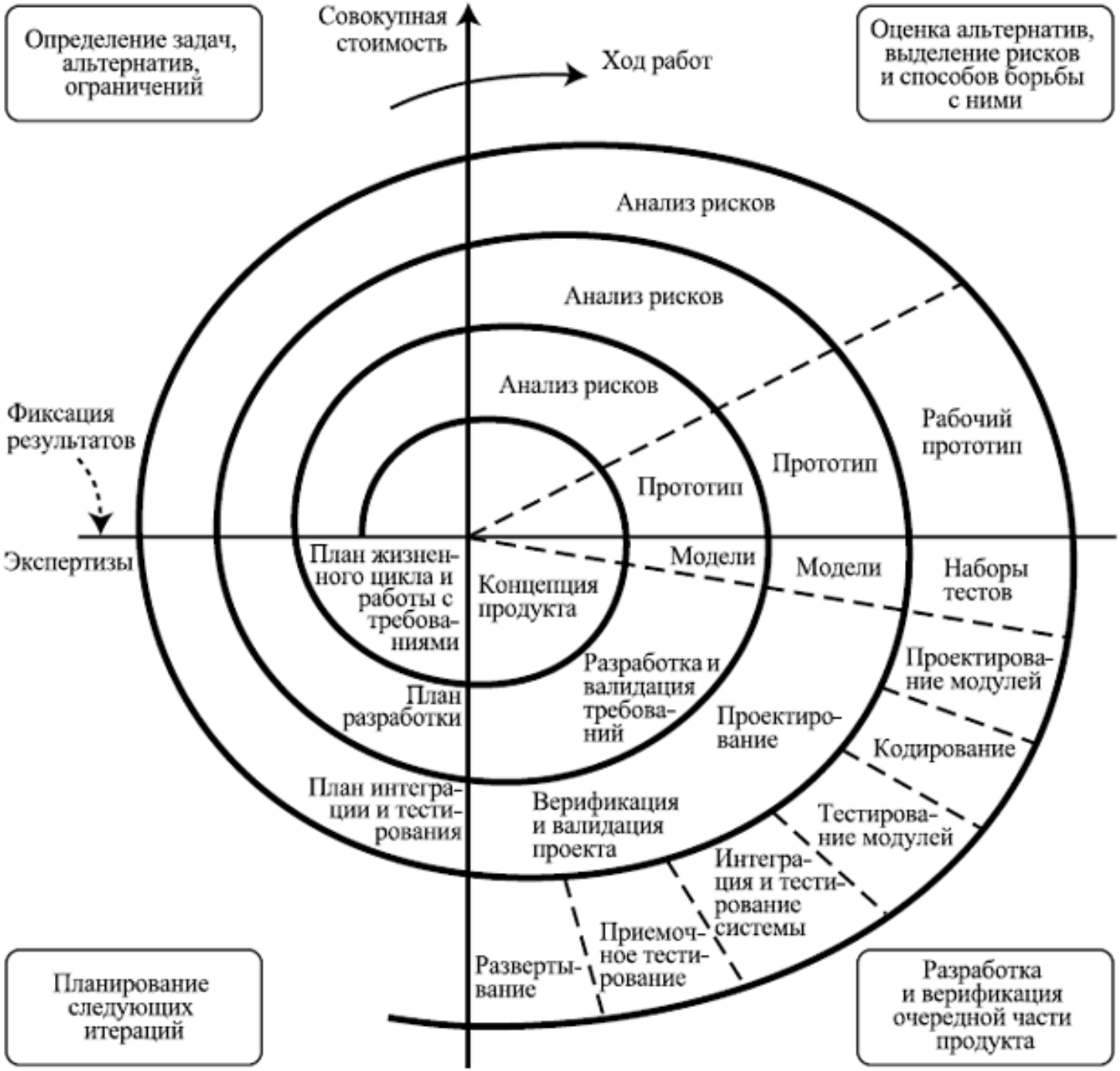


Waterfall


SYSTEM REQUIREMENTS
SOFTWARE REQUIREMENTS
ANALYSIS
PROGRAM DESIGN
CODING
TESTING
OPERATIONS
POSTMORTEM

Сокращение неопределённости




Определение задач, альтернатив, ограничений
Совокупная стоимость
Ход работ
Оценка альтернатив, выделение рисков и способов борьбы с ними
Анализ рисков
Анализ рисков
Анализ рисков
Фиксация результатов
Прототип
Прототип
Рабочий прототип
Экспертизы
План жизненного цикла и работы с требованиями
Концепция продукта
Модели
Модели
Наборы тестов
Разработка и валидация требований
План разработки
Проектирование
Проектирование модулей
Кодирование
План интеграции и тестирования
Верификация и валидация проекта
Тестирование модулей
Интеграция и тестирование системы
Приемочное тестирование
Развертывание
Планирование следующих итераций
Разработка и верификация очередной части продукта

Проектирование

- Ориентироваться на самую ближайшую цель — значит **планировать неудачу**.
- Разрабатывать нужно стратегию организации процесса, множество последовательных результатов — **планировать серию успехов**.

Проектирование

- Ориентироваться на самую ближайшую цель — значит **планировать неудачу**.
- Разрабатывать нужно стратегию организации процесса, множество последовательных результатов — **планировать серию успехов**.
- Отсутствие чёткой реалистичной цели
- **Опасность итеративного подхода** - подменить необходимость думать последовательностью плохо сходящихся изменений
- Если вы не знаете, чего хотите достичь – ничего не поможет!

Проектирование

- **Проект** - это конечный результат проектирования
- **Проект** – это не просто набор объявлений классов, есть МНОЖЕСТВО ньюансов
- **Разработка ПО** - процесс итеративный
- Конкретный проект должен иметь **начало** и **конец**
- **Проект** почти всегда переделывается
- Программы должны **производиться и обслуживаться**, несмотря на смену персонала и менеджеров

Проектирование

Узкоспециализированные слои, жёстко встроенные в структуру разработки

- Дешёвые кодеры
- Умные проектировщики
- Разрыв зарплат и недоверие

Недостаточное взаимопонимание и взаимодействие

- → упущенные возможности,
- → задержки,
- → отсутствие каналов обратной связи
- → повторение проблем (никто не накапливает опыт)
- → скованная инициатива
- → отсутствие профессионального роста
- → карьерных перспектив

Проектирование

- Люди, ответственные за разные виды деятельности, должны
  - эффективно взаимодействовать,
  - менять амплуа и отделы,
  - расти и совершенствоваться
- **Главная задача организации разрабатывающей ПО** – создание среды, где могут
  - проявиться таланты,
  - развиваться способности,
  - предлагаться идеи всеми,
  - все вместе могут радоваться результатам

Long live your project!

- Выбрать на правильное решение
- Может быть то, которое мы не рассматриваем, или то, которое нам неизвестно
- Предыдущий опыт деформирует видение

- **Идеал** легче провозгласить,
  чем реально ему следовать.»

Архитектура

- **Самые важные решения**
  - Решения бывают разные, на разных уровнях абстракции, масштабах
  - Как не ослепнуть? Как не запутаться? во всём множестве всевозможных решений
  - Могу ли я это решение отложить?
  - Могу ли я это решение делегировать студенту?
- Приоритеты.
  - **Не делаем лишнего**.
  - Фокус, а остальное бесполезно
  - Требования.
- Как описывать?
  - Документация.
  - Пояснительные записки. **Почему выбрали именно это?**

PointOfView

- Каждая точка зрения – **чудовищный объём экспертизы**!

- Определили самую главную область, в которой находятся самые важные решения.

- Остальное…
- Используем готовые решения.

Используем готовые решения

- Диаграмма классов
  - **Предметная модель** (Domain Design, которую симулировать классами) – Фаулер
  - **Искусственные классы** – Ларман
  - **Шаблоны проектирования** – Банда Четырёх
- Диаграмма обработки
  - **Алгоритмы** – Кнут, Корман
  - **Последовательности обработки классов** (sequences)
- Распределённость обработки
  - **Модель многопоточного взаимодействия** (Concurrence Design)
  - Корректное многопоточное приложение.
- Диаграмма деплойментов

Система (компоненты) –

Концепции не существуют в изоляции – определяются в контексте других концепций

Компонента – логичекая единица (набор/библиотека классов в ООП) проекта, документации и повторного использования, объединена по общему критерию.

4 точки зрения + 1 (RUP)

Логика, Физика, Статика, Динамика + Функциональность – всё точки зрения

Розански, Вудс "Архитектура программных систем"

Метрики

материально показывают характеристики кода:

- поддерживаемость
- тестопригодность
- гибкость
- осознаваемость
- вариативности
  (пригодность для внесения изменений)

СТАТИКА (данные, модели данных)

- **Гибкость** – за счёт уменьшения зависимости данных одного класса от других, **локализация изменчивости**
  - уменьшения связности,
  - появляются абстракции
  - лучшая осознанность
  - лучшая тестопригодность
  - Следующих шаг к повторному использованию или вариативности

- Вера в единственно верный способ реализации -- разновидность детской болезни

Живём в мире жёстко-статического ООП

- *“There are more useful systems developed in languages deemed awful than in languages praised for being beautiful--many more.”*

Bjarne Stroustrup

Живём в мире жёстко-статического ООП

- Если есть две оси вариативности в одном куске кода, один кусок должен вести себя по разному в двух разных осях изменчивости

- **vector** хранит любой число **items**
- **items** должны быть разных типов

- Приняв решение,
  перейдя на нестрогие парадигмы
  (*динамическую типизацию*), получаем новую гибкость
- Потом возвращаемся обратно к частичной строгой типизации и пишем тесты, много тестов!

Живём в мире жёстко-статического ООП

- *“I do not think that safety should be bought at the cost of complicating the expression of good solutions to real-life problems*.”

Bjarne Stroustrup

Компилятор помогает нам

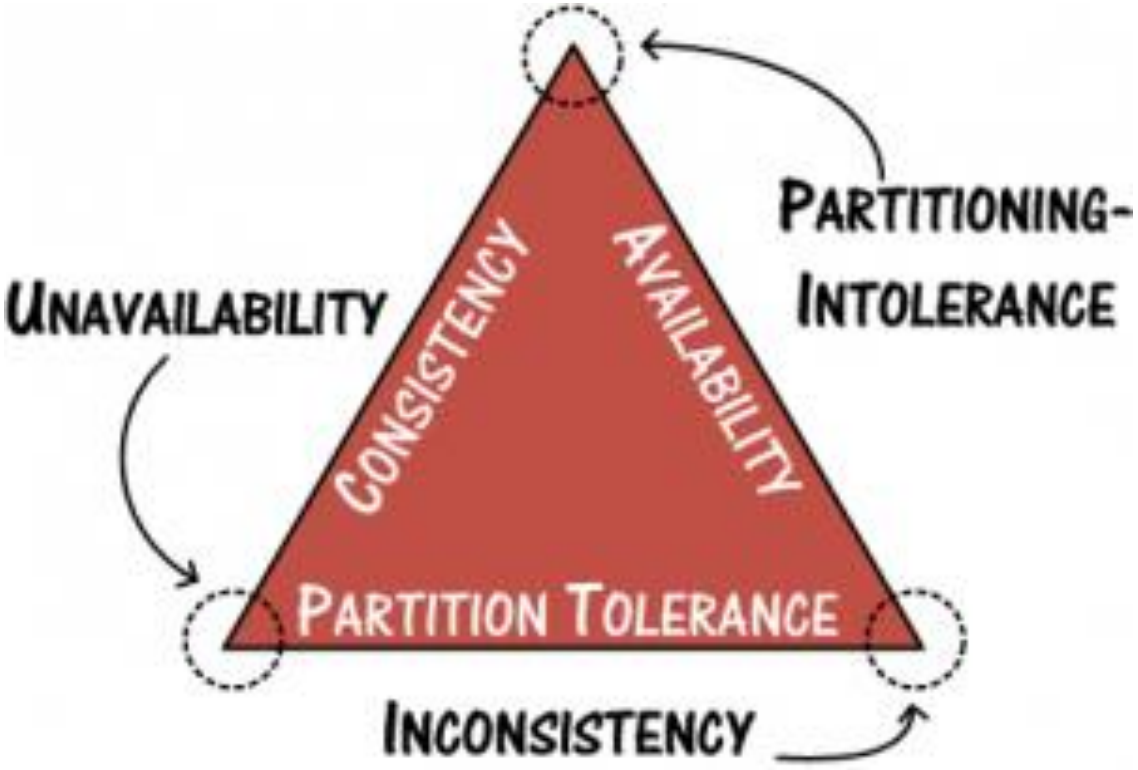
- **Компилятор помогает нам**, проверяет соответствия типов.
- Но он не может смотреть шире, он **не может проверить контракт**
- Контракт сложнее, чем явно определённые преобразования типов.
- (новый, изменённый) объект должен **вписываться в ожидания других компонентов системы**, о всех полиморфных объектах данного интерфейса.
- Здесь эффективно помогают только **грамотно разработанные тесты**

Будут ли меняться требования?

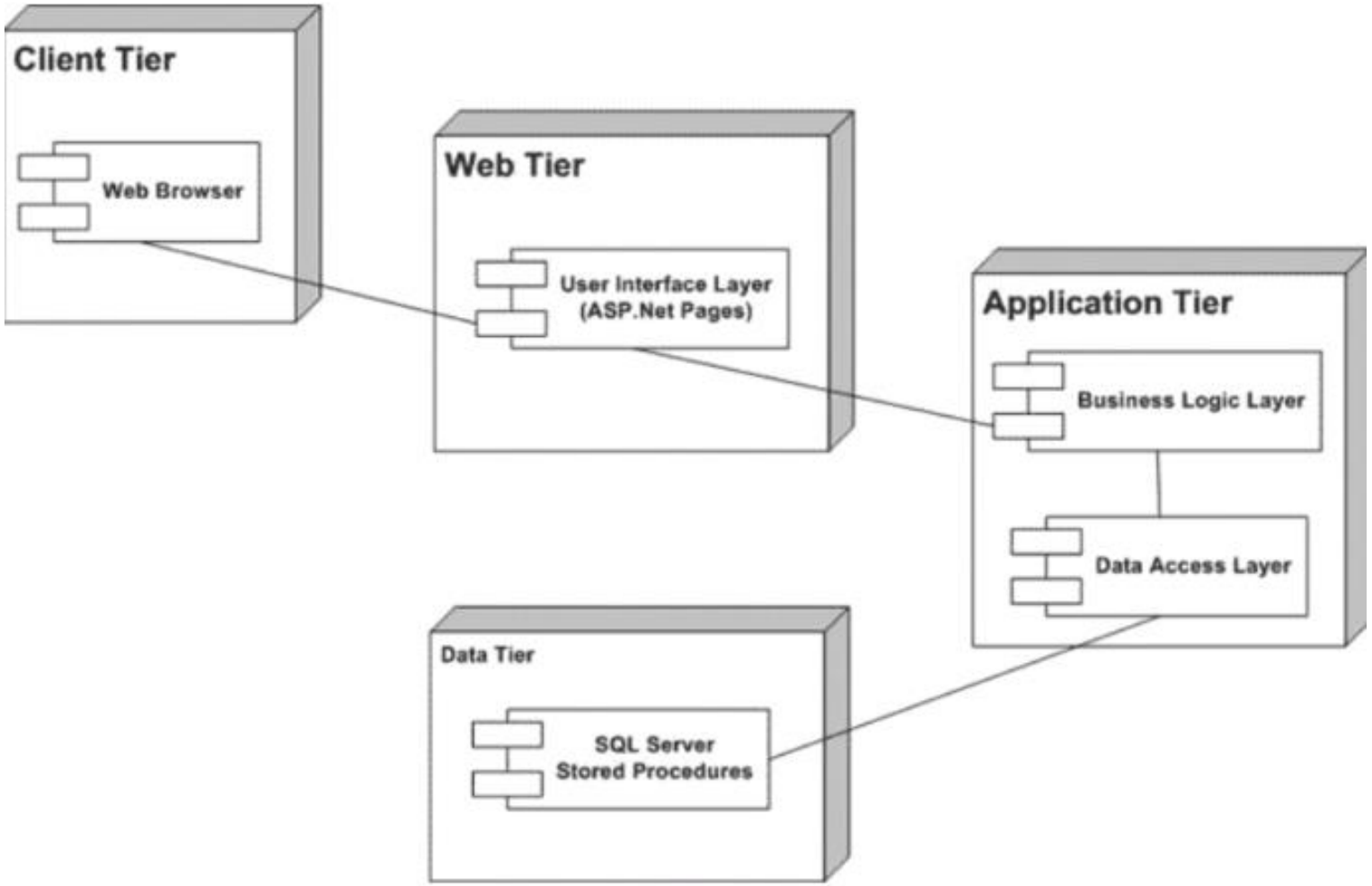
- Требования есть?
  искореняем неопределённость!
- Делаем проще
  ~~или на всякий случай давайте сделаем сложнее~~?
- Получаем обратную связь
  если продукта нет, нельзя понять, что в действительности нужно
- Взять чужую библиотеку
  ~~или быстренько написать свою~~? Depends

ДИНАМИКА (sequence, алгоритмы)

CAP-теорема



DEPLOYMENT


Client Tier
Web Browser
Web Tier
User Interface Layer
(ASP.Net Pages)
Application Tier
Business Logic Layer
Data Access Layer
Data Tier
SQL Server
Stored Procedures

Предыдущий опыт

- Ну, что здесь может быть на так?
  (а в интересном для Вас поле – даа, это сложности..)

- **Фокусироваться** не на том, что кажется сложным, но **на том, что нужно (критично важно) для проекта**, даже если это просто и очевидно..

- Проверить все шишки, которые подстерегли и больно ударили во всех прошлых проектах и успокоиться..
  Или **проверить все неочевидные сложности, которые можно ожидать** в текущем проекте?

Предыдущий опыт

- *“The connection between the language in which we think/program and the problems and solutions we can imagine is very close. For this reason restricting language features with the intent of eliminating programmer errors is at best dangerous.”*

Bjarne Stroustrup

Архитектура

- *“Maybe "just one little global variable" isn't too unmanageable, but that style leads to code that is useless except to its original programmer.”*

Bjarne Stroustrup

Архитектура

- как **набор предположений и ожиданий**
- **Контракт** – ожидания от определённого компонента
- **Рекурсивные контракты:**
  - 3 компонента + БД
  - Вся остальная система строится из предположения, что БД даст определённый контракт
  - Каждый компонент, в свою очередь состоит из каких-то компонент
  - Какие-то из них готовые, от которых мы что-то ожидаем
  - в конце обязательно упрёмся во все готовые элементы

Как понять, то что неизвестно?

- *“People who think they know everything really annoy those of us who know we don't.”*

Bjarne Stroustrup

Как понять, то что неизвестно?

- Начиная работать, не знаем лучшего способа построения
- **Внешние эксперты**
- **Экспериментирование**
  - **Прототипирование**
    - Желаемый результат – проникновение в проблему
    - Никаких жёстких требований к быстродействию и ресурсам системы – фокус на **непонятных** частых системы
    - Рабочая версия в кратчайшие сроки вместо исследования проектных альтернатив («иллюзия почти-завершённости»)
    - После использования прототип **нужно** выбросить
    - Прототип, превращённый в конечный продукт – пожиратель энергии и времени
  - **Математические модели/симуляторы**
    - Строгие, доказательные
  - **Перепроектирование старых выпущенных систем**

Далее архитектура тормоз

Требования изменились

- Если нужная гибкость изначально не вложена в реализацию, что-то изменить становится очень сложно

Бесконечно переделывать системы

- Если систему переделывать не нужно
- Значит новые фичи не нужны
- Значит продукты никто не пользуется

Делай проще, делай легче, кайфуй

- Потому что всё равно придётся переделывать
- Нельзя предугадать всё
- Куда в итоге нужно будет переделывать узнать невозможно

Если всё же вы хотите предусмотреть многое, вы уже пишите не систему, но фрэймворк, который позволяет создать нужный класс разных систем, но
Вы всё равно ошибётесь в предсказаниях!

Далее архитектура тормоз

- *“One of the things I really like about programming languages is that it's the perfect excuse to stick your nose into any field. So if you're interested in high energy physics and the structure of the universe, being a programmer is one of the best ways to get in there. It's probably easier than becoming a theoretical physicist*.”

Bjarne Stroustrup

Эффективнось

- Дональд Кнут
  «Преждевременная оптимизация - корень всех зол»
- Ясная структура проекта, оставшаяся после многих итераций его переработки, помогает в нахождении узких мест и располагает к анализу эффективности
- Но зло – не любая оптимизация.
- *«Сначала измеряй – потом меняй»*
- Интуиция разработчика часто обманчива
- Необходимо методично избегать в коде программы неэффективных конструкций, или требующих много сил для придания им эффективности

Пишем не программу

решаем задачу заказчика!

- **Концепеция** (**vision**) – список бизнес-целей
- **Метрика успеха**
- **Приоритеты**
  - Фокус на ключевой функциональности
  - Быстрый старт
- Вовлечённость заказчика
- Специализация исполнителя

Пишем не программу

- “***Legacy code*** *often differs from its suggested alternative by* ***actually working*** *and* ***scaling***.”

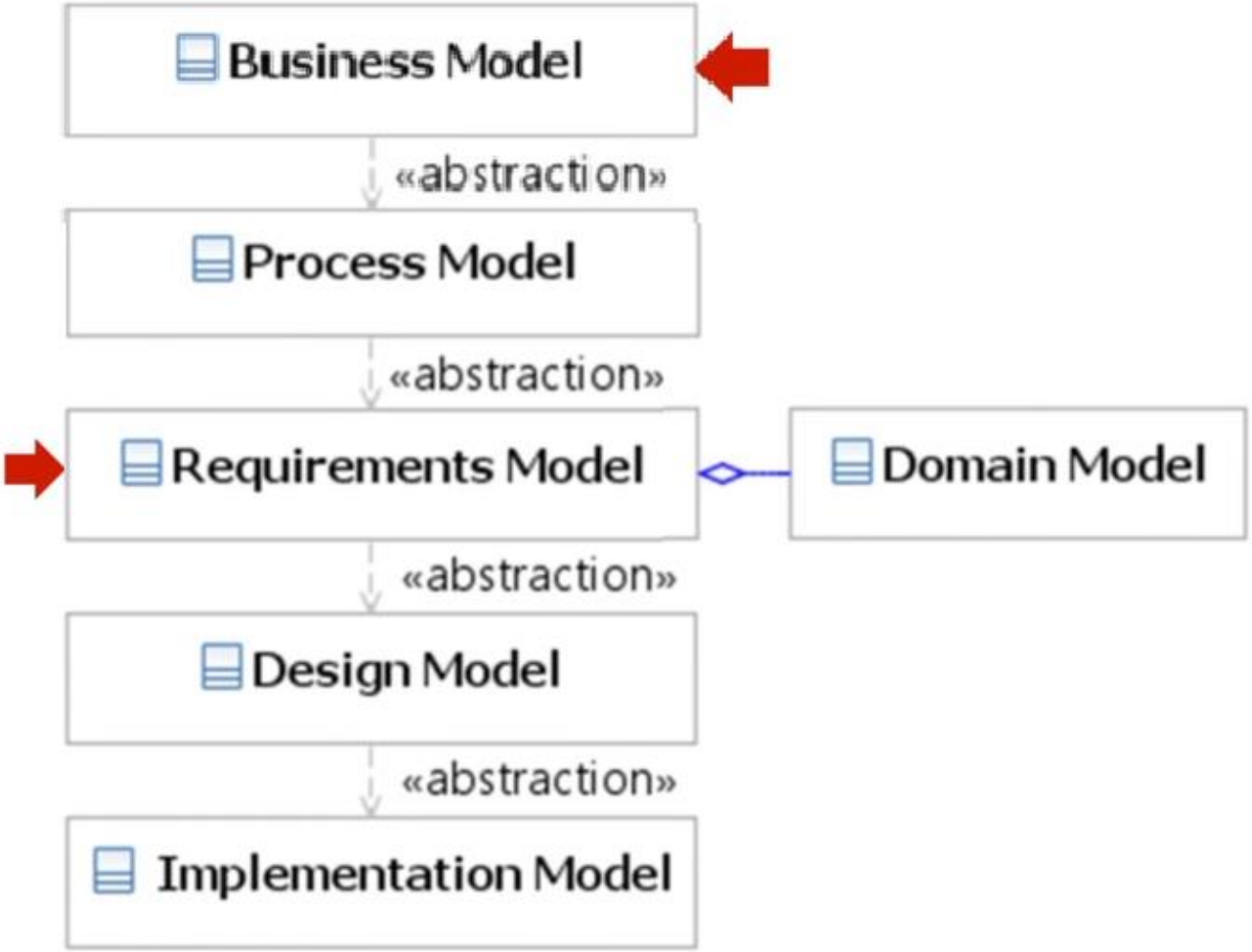
Bjarne Stroustrup

Пишем не программу

- **Плохоработающая система лучше идеальной нереализованной системы**
- Есть идеально спроектированные системы, которыми никто не пользуется
- Есть криво-реализованные проекты, которыми ежесуточно пользуются миллионы людей

- Важно не только всё сделать хорошо, ещё успеть вовремя и попасть в нужды

Откуда рождается архитектура



Happy path :)

main flow

- Желаемое поведение системы
  - 20%
- Альтернативные сценарии (ошибки)
  - 80%

Тестирование

- «*Как тестировать*?» – ответа нет
- «*Когда тестировать*?» – общий ответ: как можно раньше (и как можно чаще).
- «*Достаточный объём тестирования*?» – на практике недотестированные проекты встречаются гораздо чаще, перетестированных
- Количество ресурсов, выделенных под тестирование по сравнению с проектирование и реализацией зависит от природы системы
- *На тестирование нужно **больше ресурсов**! чем на превоначальную реализацию системы.*

Тестирование

- *Проект системы* должен вселять уверенность, что абсолютное большинство ошибок будет устранено комбинацией статических проверок, статического анализа, модульного тестирования и тестирования системы, как целого.
- *Стратегия тестирования* должна быть частью общего проекта.
  - Логгирование (отладочная печать)
  - Журналирование изменений состояний
  - Механизм проверочных утверждений (assertions)
- Как только систему можно запустить – **нужно начать тестирование**.
- *Дополнительный тестирующий код*, легко может быть удалён из конечной версии системы

Тестирование и зависимости

- Зависимости классов
- Как обеспечить изоляцию?

Основные подходы:

- **Dummy.** Передаются в тестируемые классы/методы/функции в виде параметра без поведения, внутри как правило с ним ничего не происходит
- **Stub.** Подменяется внешняя зависимость, игнорируются все данные, входящие в stub из тестируемого объекта.
- **Fake.** Замена легковесной реализацией.
- **Mock.** Объекты, которые имитируют поведение реальных.
  Полезно, когда настоящие объекты непрактично/невозможно вставлять в юнит-тест, но поведение нужно сохранить. **behavioral tests**

Тест упал, что делать дальше?

- 1. tracing и logging
- 2. **strace** и похожие утилиты
- 3. **gdb** и похожие отладчики
- 4. **aSan** или address sanitaizer
- 5. **valgrind**

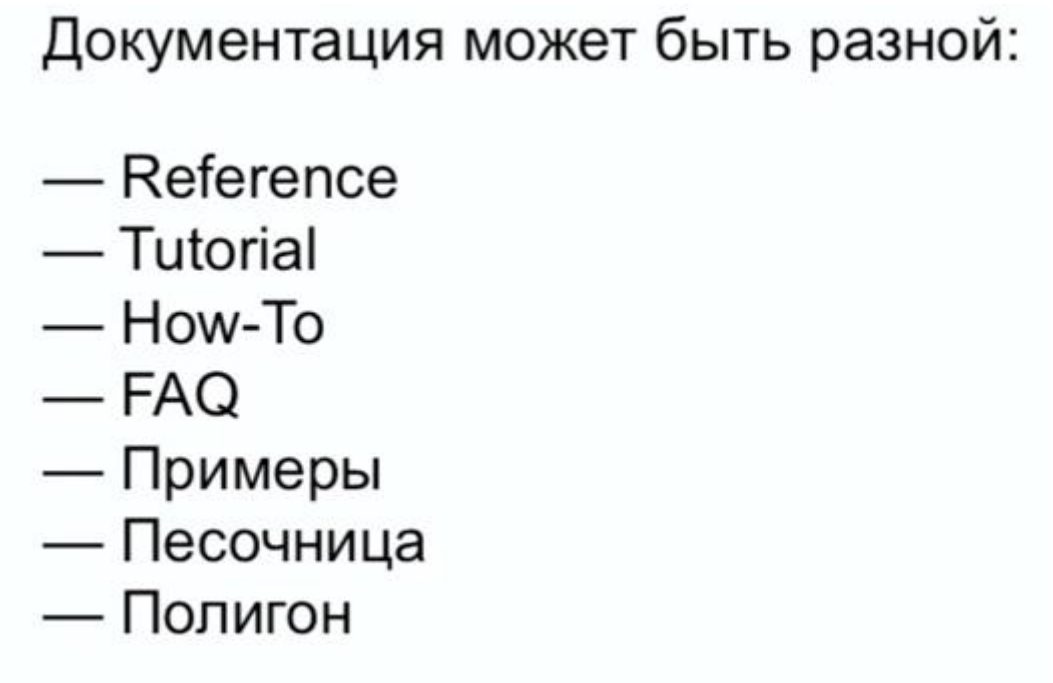
Документация

- *“If you do anything useful it will haunt you forever after, and if you have a major success you get decades of **hard manual labor** - meaning you have to work on the manual.”*

Bjarne Stroustrup

Документация API

Писать [публичную] документацию подчас гораздо сложнее, чем код



Компонентно-Ориентированное Программирование (КОП)

- **КОП** — это парадигма программирования направленная на повышение надёжности коммерческих бизнес-систем.
- **Компонент** - независимый модуль для повторного использования и разворачивания
  - более крупная единица, чем объект (объект – это конструкция уровня языка программирования)
  - содержит множественные классы
  - не зависит от языка программирования (в большинстве случаев)
- Суть сводится к возможности контролировать взаимодействие проектируемых и выполняемых модулей на предмет согласованности информационных структур.
  - Частично воплощена в Java, C#, Ada
  - Прямое применение в Modula-2, Oberon, Oberon-2

Reuse. Методология повторного использования

- Термин **reuse** — повторное использование — активно используется программистами на практике.
- Если вы написали хорошую библиотеку функций для работы с 3D-графикой, то теперь можете использовать её во всех своих приложениях, где возникает необходимость рисовать пространство. Это просто и очевидно.
- «*Я видел дальше других, не потому, что я сам высокого роста, просто я стоял на плечах гигантов*» — Исаак Ньютон
- **Метасистемный переход** — это изменение (повышение уровня) организации системы, при котором элементарными объектами новой системы становятся системы предыдущего уровня.
- Метасистемные переходы в живом мире:
  - от белков — к живым клеткам,
  - от клеток — к тканям и органам,
  - от органов — к живым существам,
  - от живых существ — к сообществам и экосистемам.

Reuse

- **Reuse** — это оперирование имеющимися методами, инструментами, алгоритмами и решениями с целью получения новых методов, инструментов для решений новых более обширных задач.
  - **процедурное программирование** заключается в разбиении программы на процедуры (функции, действия), при котором для описания новых, высокоуровневых процедур используются имеющиеся (более) низкоуровневые процедуры.
  - **мета-алгоритм**, который оценивает ситуацию на входе и использует наиболее подходящий для этой ситуации алгоритм из имеющихся уже реализованных алгоритмов решения одной и той же задачи, которые успешны в различных случаях.
  - **модульное программирование**, создание самостоятельных моделей – кирпичиков, из которых строятся сложные системы. **Модуль** – *готовый **стабильный** набор инструментов* (на который можно опереться), обеспечивающий успех в решении широкого класса задач на следующем уровне абстракции.

При решении задач

- Наиболее полно используйте существующие инструменты и стандарты
- Предоставляйте результаты своего труда в виде максимально общего набора инструментов, оформленного в соответствии со стандартами
  - в виде простых, но достаточно мощных *общих функций и классов*,
  - снабженных *документацией*,
  - и *тестами* самопроверки подтверждающими стабильность и согласованность описания и функциональности
  - с простыми очевидными *примерами использования*
  - обозначенными *путями интеграции* с другими инструментами (системами)
- Именно **благодаря методу reuse** удаётся решать немыслимо сложные задачи

Не любой код может использоваться повторно!

- **Работает** - реально используется
- **Понятно** – есть норм. структура, комментарии, док.
- **Экономично** – затраты на разработку и сопровождение меньше, чем собственный проект
- **Доступно**
- **Совместимо** – подключаемо, настраиваемо
  - Подгонка компоненты, корректировка, универсализация, улучшение совместимости, всё всплывёт, когда компонента, будет **подключена хотя бы раз**
- **Повторное использование** – результат проектирования
- В крупной компании должен быть **отдел**
  - коллекционировать опыт, создать систему хранения проектов и документации, нести затраты на сбор, группировку, додокументирование, поддержку и **популяризацию** программных фрагментов

СПАСИБО ЗА ВНИМАНИЕ!