# Технологии грамотной разрабоки программного обеспечения

# О чём обычно говорят на ТРПО

- О важности проектирования
- О командной работе
- О докуметации
- О тестировании
- О различных методиках разработки программ: ‘Waterfall model’, ‘V-model’, ‘Инкрементальная модель’, ‘RAD’, ‘Agile’, ‘Spiral model’.
- Об оценке сложности проектов. ‘Метод аналогий’, ‘Метод функционального размера’.
- Немного о паттернах проектирования. Чаще всего в теории, либо на практике, но в лабораторных условиях

# Зачем нам нужно проектирование

- “Думать уже поздно, пора делать!”
- Чем больше пишешь хорошо спроектированную программу, тем проще становится писать

# Признаки хорошей архитектуры

- Эффективность
- Гибкость, расширяемость
- Тестируемость
- Переиспользуемость

# Как хорошо спроектировать программу?

- Разбить всё на модули
- Модули разбить на пакеты
- Пакеты разбить на детали
- Детали разбить на части
- Части разбить на компоненты
- Компоненты разбить на гаджеты
- Гаджеты разбить на подструктуры
- Подструктуры разбить на запчасти
- Запчасти разбить на атомарные единицы
- Атомарные единицы разбить на пальчики

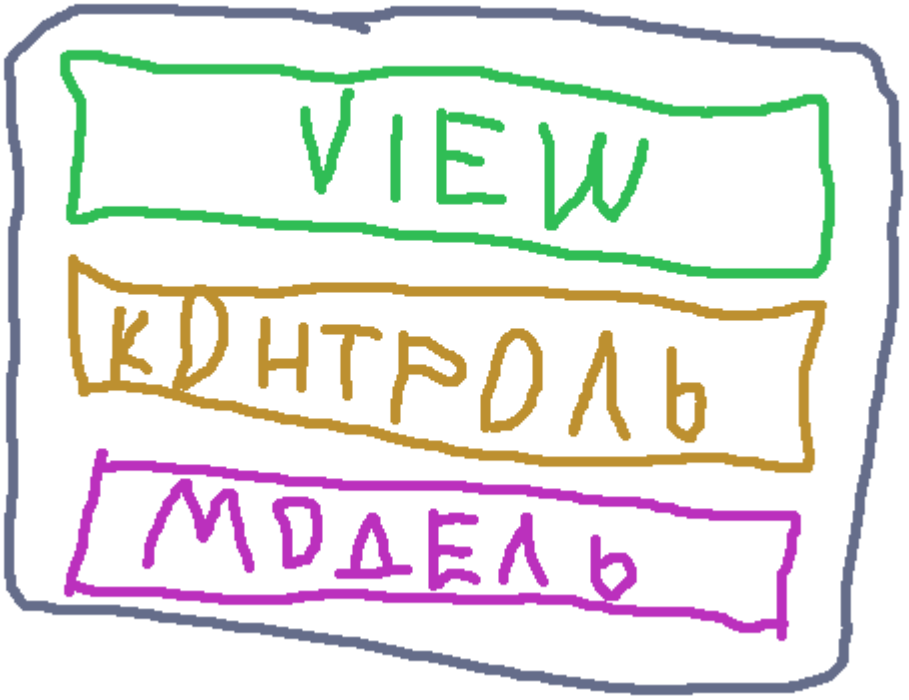
# Как хорошо спроектировать программу?

Можно использовать фреймворк, в котором всё сделано за тебя.

Это работает только если приложение очень маленькое.

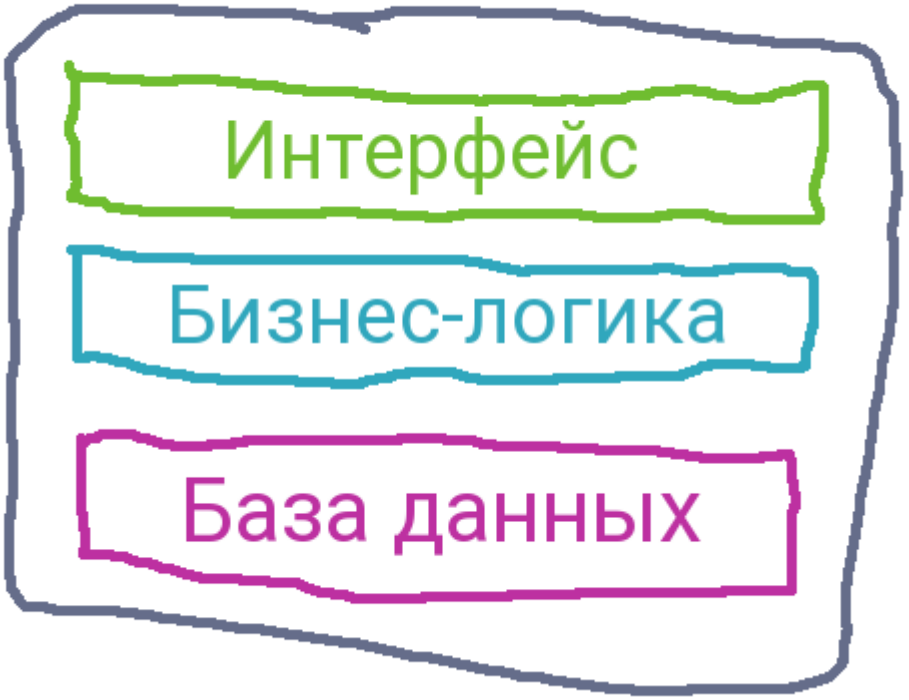
# Иерархическое разбиение

- Описываем задачи системы и делаем раздел под каждую задчу



# Функциональное разбиение

- Придумываем независимые части системы и разбиваем по ним



# S.O.L.I.D.

- Single responsibility
- Open-closed principle
- Liskov’s substitution principle
- Interface segregation principle
- Dependency inversion principle

# Единственная ответственность

Можно ли иметь функцию, которая:

- Устанавливает соединение с сервером
- Читает данные с удалённого сервера
- Пишет данные в файл

# Единственная ответственность

- Можно! Но она должна полагаться на три функции, разделяющие эту ответственность

# Открытость-закрытость

- Добавлять новинки можно без модификации кода

```
1 fn create_post(db: Database, postMessage: string) {
2     if postMessage.starts_with("#") {
3         db.insert_into("POSTS", postMessage);
4     } else {
5         db.insert_into("TAGS", postMessage.slice(1, -1));
6     }
7 }
```

# Принцип замены

```
1  template <typename T>
2  class SuperList : public std::list<T>
3  {
4      using typename Super = std::list<T>;
5
6      void dropLast() override
7      {
8          Super.clear();
9      }
10 }
```

# Разделение интерфейсов

```
1 class BlogClient where
2     readPost  :: Blog -> IO String
3     writePost :: String -> IO Blog
```

# Инвертирование зависимостей

- Модули высокого уровня не должны думать о низкоуровневом

```
2 def performDangerous(s, point, callback):
3     try:
4         response = ioHandler.do(s, point)
5         return callback(response)
6     except IOError as err:
7         sys.stderr.write(err.message, end="\n")
```