Программная инженерия

Анализ и Проектирование ПО

Проектирование ПО

Что такое проектирование ПО?

- **Проектирование ПО** – это осознанный выбор решений о логической организации составных частей программного комплекса.
- Проект может быть представлен в виде модели UML, неформального документа или набросков представляющих определенные аспекты дизайна.
- Проектирование – это творческий процесс, который тяжело описать в виде формальных структурированных методов.

Проектирование архитектуры

Разработка и документация архитектуры ПО обеспечивает следующие преимущества:

1. **Взаимодействие с заказчиком -** высокоуровневое представление системы, может быть использовано для дискуссии о проектируемой системе с заказчиками и пользователями.
2. **Системный анализ -** может быть выявлено, сможет ли система соответствовать заявленным требованиям производительности, качества поддержки и др.
3. **Высокоуровневое повторное использование ПО -** архитектура систем с похожими требованиями часто похожа. Может иметь место повторное использование программного обеспечения высокого уровня.

Влияние нефункциональных требований на архитектуру ПО

- **Производительность** – локализация критически-важных операций в малом наборе подсистем с минимальным объемом взаимодействия между ними. Возможно использование относительно больших модулей.
- **Безопасность** – слоистая архитектура, на нижних слоях которой скрываются наиболее критические операции, с учетом высокоуровневых методов обеспечения безопасности.
- **Доступность** – избыточность используемых компонентов, для возможности их замены и обновления без остановки системы.
- **Поддерживаемость** – мелкомодульная архитектура слабосвязанных компонентов, которые с легкостью могут быть заменены; отказ от структур с разделением данных.

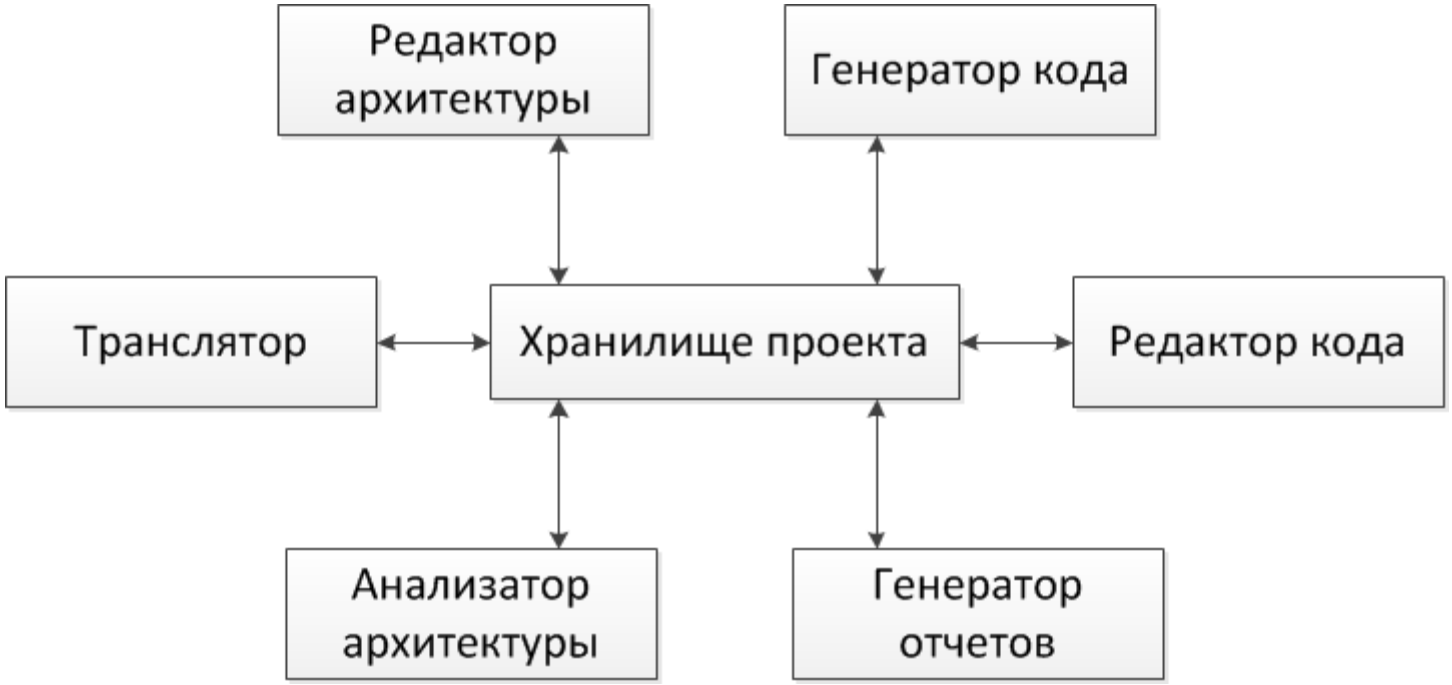
Естественно, данные требования могут противоречить друг другу (производительность VS поддерживаемость). Но при поиске оптимальной архитектуры должны быть расставлены приоритеты и коэффициенты тех или иных требований для выработки компромисса

Общие методы системной организации

- **Системная организация** описывает общую стратегию, используемую для структурирования системы.
- Выделяют следующие модели системной организации:
  - Модель репозитория
  - Клиент-серверная модель
  - Многослойная модель

Модель репозитория (общего хранилища)

- Весь объем данных программной системы хранится в едином **общем хранилище (базе данных)**, доступ к которому может быть получен из любого программного компонента



Достоинства модели репозитория

- Эффективный метод обработки больших объемов данных
- Нет проблемы с обменом данными между компонентами
- Резервное копирование, безопасность, ограничение доступа централизованы
- Модель использования общих данных ясно видна из схемы репозитория

Недостатки модели репозитория

- Все компоненты должны подчиняться модели данных репозитория
- Проблемы развития системы, масштабируемости, изменения модели
- Невозможно установить политику безопасности или резервного копирования для каждого отдельного компонента
- Затруднительно распределение репозитория по нескольким машинам

Клиент-серверная модель

- Система организуется в виде набора серверов, предоставляющих определенные сервисы клиентам. Основные компоненты модели:
  - Набор серверов, предоставляющих сервисы другим подсистемам
  - Набор клиентов, использующих сервисы
  - Промежуточная среда, обеспечивающая взаимодействие между клиентом и сервером

Пример клиент-серверной модели

Достоинства и недостатки клиент-серверной модели

- Клиент-серверная модель опирается на распределенную архитектуру и эффективно реализуется в РВС.
- Легко можно добавлять новые сервера или обновлять старые без заметных изменений в инфраструктуре.
- Но невозможно так же легко разделять общие данные между клиентами и серверами. В связи с этим необходимо применять специальные форматы данных для обмена.

Многослойная модель (модель абстрактной машины)

- Система распределяется на слои, каждый из которых представляет набор определенных сервисов.
- Каждый слой можно представить в виде «абстрактной машины» язык которой определяется набором сервисов, предоставляемых на соответствующем слое.

Пример многослойной модели

Достоинства и недостатки многослойной модели

- Поддержка инкрементальной разработки
- Возможность замены слоев
- При изменении затрагиваются только смежные слои

НО

- Сложность организации архитектуры
- Низкая скорость работы

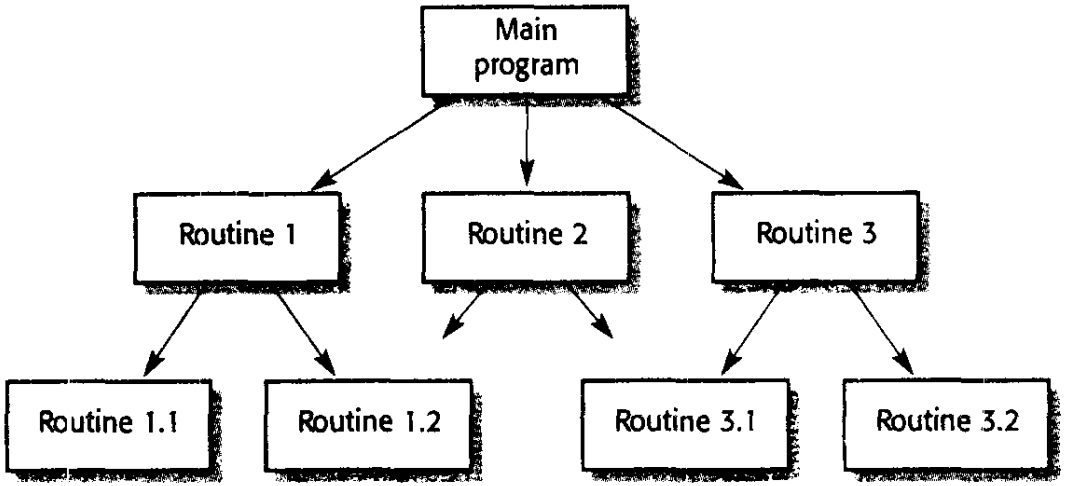
Методы модульной декомпозиции

Методы модульной декомпозиции

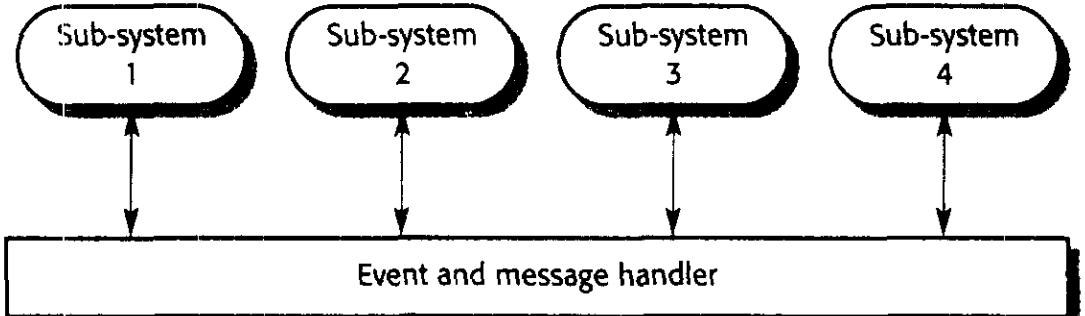
- Объектно-ориентированная декомпозиция: система представляется в виде набора взаимодействующих объектов.
- Функционально-ориентированная декомпозиция: система представляется в виде функциональных модулей, на вход которых поступают одни входные данные, а выходят выходные.

Методы организации контроля

- **Системы с централизованным контролем:** одна подсистема отвечает за контроль, запускает и останавливает другие подсистемы.
- **Событийно-ориентированные системы**: каждая подсистема может отвечать на внешние события (из других подсистем или из окружающей среды)



Централизованная система: «Запрос - ответ»

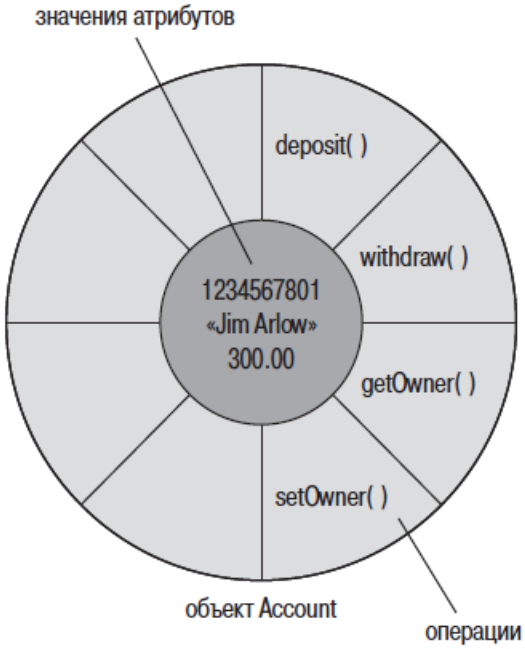


Событийно-ориентированные система

Принципы Объектно-ориентированного проектирования

ОО-ПРОЕКТИРВОАНИЕ

- ОО-проектирование основывается на концепции объекта: сущности, сохраняющей свое внутреннее состояние и поддерживающей операции для управления внутренним состоянием.



Процесс объектно-ориентированного решения задачи

- ОО анализ – формирование объектно-ориентированной модели предметной области.
- ОО проектирование – развитие ОО модели программной системы, для достижения определенных требований.
- ОО реализация – реализация разработанной модели посредством ОО языка программирования.

SOLID

- **SOLID** - аббревиатура пяти основных принципов дизайна классов в объектно-ориентированном проектировании
  - *Single responsibility:* На каждый объект должна быть возложена одна единственная обязанность.
  - *Open-closed:* Программные сущности должны быть открыты для расширения, но закрыты для изменения.
  - *Liskov substitution:* Классы в программе должны поддерживать замену их наследниками без изменения свойств программы.
  - *Interface segregation:* Много специализированных интерфейсов лучше, чем один универсальный.
  - *Dependency inversion:* Модули верхнего уровня не должны завесить от модулей нижнего уровня.

Принцип единственной обязанности (*Single responsibility principle*)

- На каждый класс должна быть возложена одна единственная обязанность.
- Все методы и атрибуты класса должны быть ориентированы на реализацию данной обязанности.

Принцип открытости/закрытости (*Open/closed principle*)

- Программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения

- Два варианта значения принципа:

  1. *принцип Мейера:* разработанная реализация класса требует только исправления ошибок, а новые или изменённые функции требуют создания нового класса (можно применять наследование – **наследование реализации**)

  2. *Полиморфный принцип:* наследование может быть только от абстрактных базовых классов. Спецификации интерфейсов могут быть переиспользованы, но реализации изменяться не должны.

Принцип подстановки Барбары Лисков (*Liskov substitution principle*)

- Функции, которые используют базовый тип, должны иметь возможность использовать подтипы (наследники) базового типа не зная об этом.
- Классы в программе должны поддерживать замену их наследниками без изменения свойств программы.
- Поведение наследуемых классов не должно противоречить поведению, заданному базовым классом, то есть поведение наследуемых классов должно быть ожидаемым для кода, использующего переменную базового типа.

Пример противоречия принципу SLP

- Сформируйте иерархию классов, содержащую понятия «прямоугольник» и «квадрат»

```
class Rectangle {
public:
   virtual void setHeight(int newHeight);
   virtual void setWidth(int newWidth);
   virtual int height() const;       // возвращают текущие значения
   virtual int width() const;
...
};
void makeBigger(Rectangle& r)        // функция увеличивает площадь r
{
   int oldHeight = r.height();
   r.setWidth(r.width() + 10);       // увеличить ширину r на 10
   assert(r.height() == oldHeight); // убедиться, что высота r
}                                    // не изменилась
```

Пример противоречия принципу SLP (2)

```
class Square: public Rectangle {…};
Square s;
...
assert(s.width() == s.height()); // должно быть справедливо для
                                 // всех квадратов
makeBigger(s); // из-за наследования, s является
               // Rectangle, поэтому мы можем
               // увеличить его площадь
assert(s.width() == s.height()); // По-прежнему должно быть справедливо
                                 // для всех квадратов
```

- Таким образом, принцип SLP – не наследуй квадрат от прямоугольника!

Принцип разделения интерфейсов (*Interface segregation*)

- Много специализированных интерфейсов лучше, чем один универсальный.
- Клиенты не должны зависеть от методов, которые они не используют.
- Слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе.

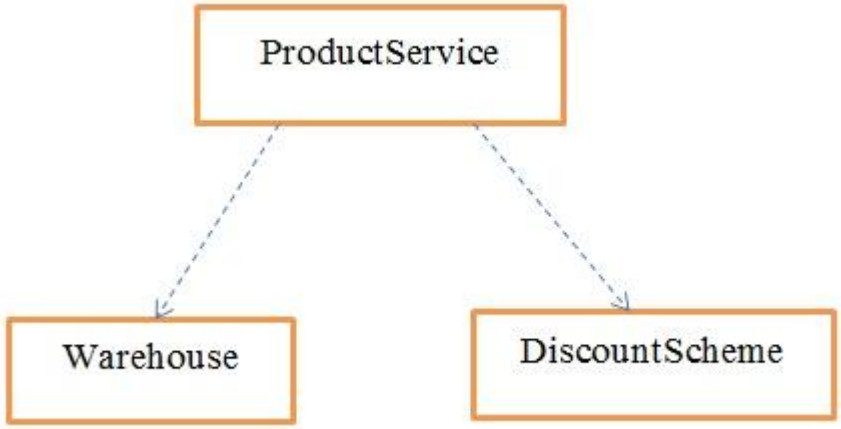
Пример нарушения принципа разделения интерфейсов

```
public interface Animal {
     void fly();
     void run();
     void bark();
}

public class Bird implements Animal
{
     public void bark() { /* do nothing */ }
     public void run()  {     // write code about running of the bird}
     public void fly()  {     // write code about flying of the bird }
}

public class Cat implements Animal
{
     public void fly()  {throw new Exception("Undefined cat property"); }
     public void bark() {throw new Exception("Undefined cat property"); }
     public void run()  {// write code about running of the cat }
}
```

Принцип инверсии зависимостей (*Dependency Inversion Principle*)

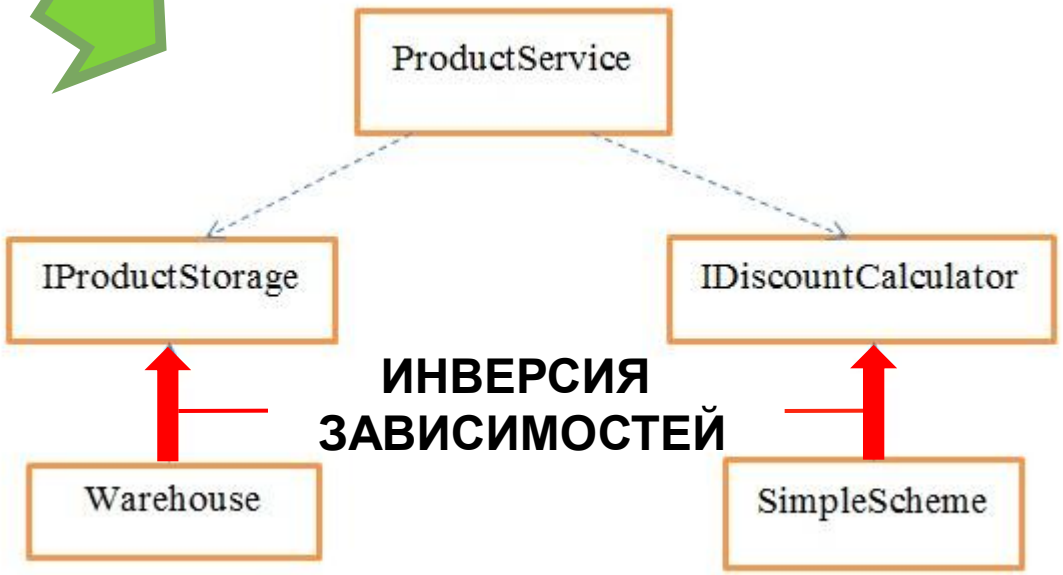
- Принцип обеспечения слабой связанности разрабатываемых модулей:
  - Высокоуровневые модули не должны зависеть напрямую от низкоуровневых модулей. И те, и другие должны зависеть от абстракций
  - Абстракции не должны зависеть от детализации. Детализация должна зависеть от абстракций.

Пример: расчет скидок для товаров на складе



Нужно выделить использование реализаций Warehouse и Discount Scheme из ProductService при помощи абстракций.

Мы не можем без изменения ProductService рассчитать скидку на товары, которые могут быть не только на складе. Так же нет возможности подсчитать скидку по другой карте скидок (с другим Disctount Scheme).



Проблемы, решаемые применением принципа инверсии зависимостей

1. **Жесткость ПО:** если изменение одного модуля приводит к изменению других модулей.
2. **Хрупкость ПО:** если изменения в одном модуле приводят к неконтролируемым ошибкам в других частях программы.
3. **Неподвижность ПО:** если модуль сложно отделить от остальной части приложения для повторного использования.

http://worldofapple.com

DON'T REPEAT YOURSELF

Repetition is the root of all software evil

Keep it DRY

- **Don't Repeat Yourself** *(Не повторяйся)*
  - Каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы

> Энди Хант, Дэйв Томас
> **The Pragmatic Programmer**

  - Применяется к:
    - Исходному коду реализации методов, классов, модулей;
    - Схемам баз данных;
    - Планам тестирования;
    - Документации и т.д.
  - Если DRY применяется успешно, то изменение единственного элемента системы не требует внесения изменений в другие, логически не связанные элементы.