Объектно-ориентированное программирование

Object-oriented programming

IX. Принципы SOLID

The SOLID principles

“Грис [David Gries] в интервью сказал, "теперь, после того, как все эти **ужасные языки** (как C и C++) исчезли, мы можем учить **правильно программировать**. Используя язык Java мы учим людей наследованию, делегации и приведению типов **перед тем, как учим их писать циклы**".”

A. Степанов

Принципы дизайна классов

- Единственной ответственности
  - **S**ingle Responsibility Principle (SRP)
- Открытости/закрытости
  - **O**pen/Closed Principle (OCP)
- Подстановки
  - **L**iskov Substitution Principle (LSP)
- Разделения интерфейса
  - **I**nterface Segregation Principle (ISP)
- Инверсии зависимостей
  - **D**ependency Inversion Principle (DIP)

“The **SOLID principles are not rules**. They are not laws. They are not perfect truths. The are statements on the order of "An apple a day keeps the doctor away." This is **a good principle**, it is good advice, **but it’s not a pure truth**, nor is it a rule.”

R. Martin

http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

Симптомы

- **Косность** (rigidity) – стойкость к изменениям
- **Хрупкость** (fragility) – неустойчивость при малейших изменениях
- **Неподвижность** (immobility) – непереносимость кода из проекта в проект
- **Вязкость** (viscosity) – сложность в применении “правильных” практик к коду

Основные причины возникновения симптомов

Часто **изменяющиеся требования** к дизайну порождают сложные цепочки зависимостей, **добавление новых зависимостей** с каждым изменением.

Способ борьбы с деградацией дизайна приложения – **управление зависимостями** с помощью блокирования путей проникновения зависимостей в дизайн (dependency firewall).

Open Closed Principle (OCP)

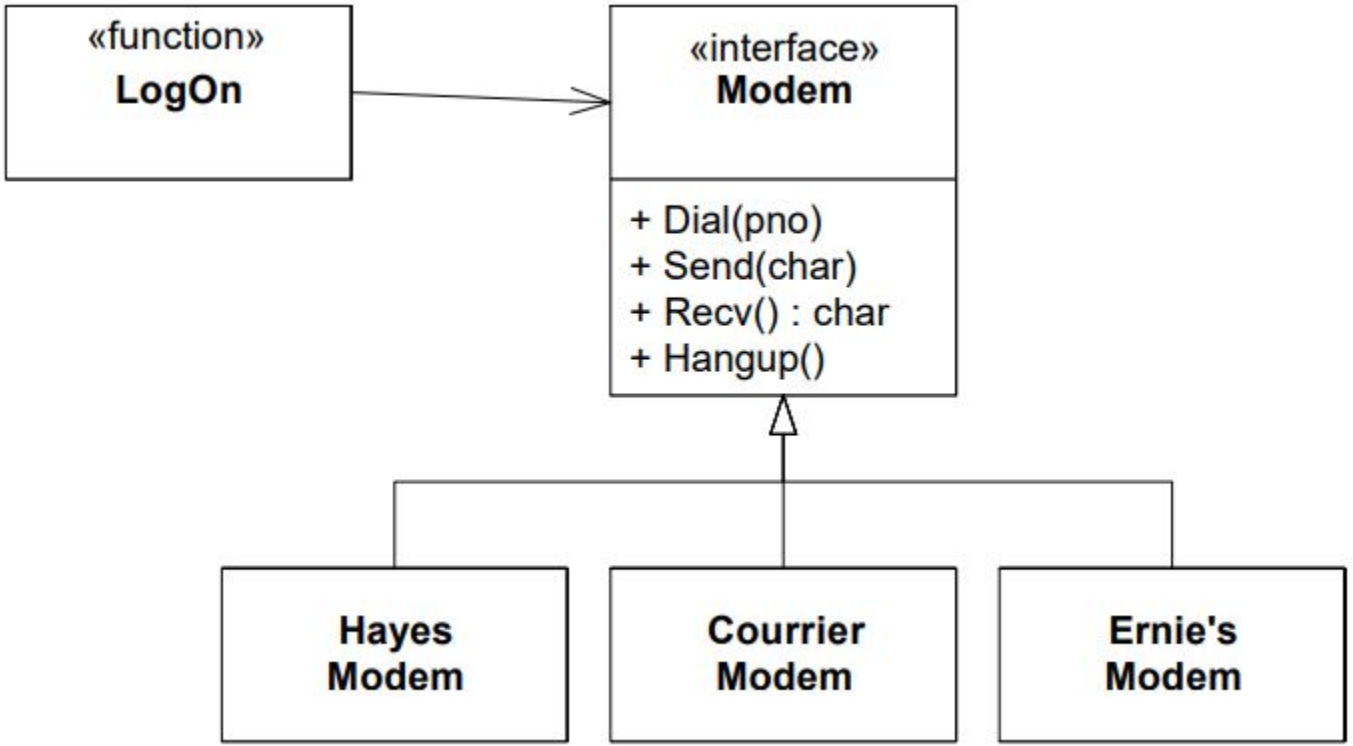
"A module should be **open for extension** but **closed for modification**".

Необходимо изменить поведение модуля, не меняя его исходный код. Ключ к достижению такого эффекта – в **абстракции**.

Структурное решение (OCP)

```
struct Modem {
enum Type {
    hayes, courrier, ernie} type;
};
struct Hayes {
    Modem::Type type;
    // ..
};
struct Courrier {
    Modem::Type type;
    // ..
};
struct Ernie {
    Modem::Type type;
    // ..
};
```

```
void LogOn(
    Modem &m,
    string& pno,
    string& user,
    string& pw)
{
    if(m.type == Modem::hayes)
        DialHayes((Hayes&)m, pno);
    if(m.type == Modem::courrier)
        DialCourrier((Courrier&)m, pno);
    if(m.type == Modem::ernie)
        DialErnie((Ernie&)m, pno);
    // ..
}
```


«function»
LogOn
«interface»
Modem
+ Dial(pno)
+ Send(char)
+ Recv() : char
+ Hangup()
Hayes Modem
Courrier Modem
Ernie's Modem

Динамический полиморфизм (OCP)

```
struct Modem {
    virtual void Dial(const string &pno) = 0;
};

struct Hayes : Modem {
    void Dial(const string &pno) override;
    // ..
};

void LogOn(Modem &m, string &pno, string &user, string &pw){
    m.Dial(pno);
    // ..
}
```

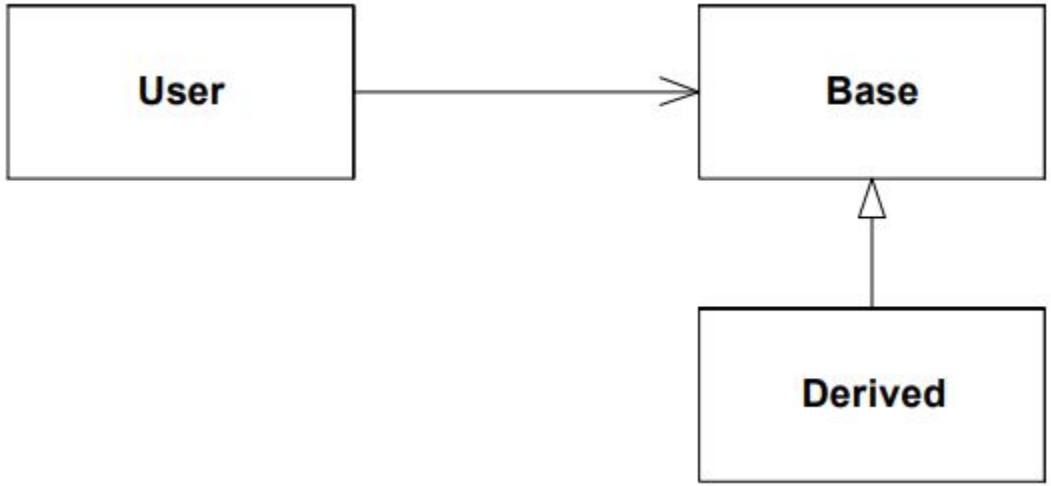
Статический полиморфизм (OCP)

```
template <typename _Modem>
void LogOn(_Modem &m, string &pno, string &user, string &pw)
{
    m.Dial(pno);
    // ..
}
```

Liskov Substitution Principle (LSP)

"Subclasses should be **substitutable for their base** classes".

Клиент базового класса не должен терять доступ к функциональности если происходит **подстановка подкласса**.




```
void User(Base &b) {
    // ...
}

Derived d;  // объект подкласса
User(d);    // контракт не должен нарушаться
```

Дилемма “овала/круга” (LSP)

“Все круги – это частные случаи овала (с совпадающими фокусами)”.

```
class Ellipse {
    Point focusA;
    Point focusB;
    double axis;
public:
    void SetF(Point, Point);
    void SetA(double);
    double Area();
    // ...
};
```

```
struct Circle : Ellipse {
    void SetF(Point a, Point b) {
        focusA = a; focusB = a;
    }
    double Area();
    // ...
};

void User(Ellipse &e) {
    Point a, b;
    e.SetF(a, b);
    assert(e.GetFoci() == {a, b});
}
```

Предусловие (LSP)

```
void User(Ellipse &e) {
    if(typeid(e) != typeid(Ellipse))
        return;
    Point a, b;
    e.SetF(a, b);
    assert(e.GetFoci() == {a, b});
}
```

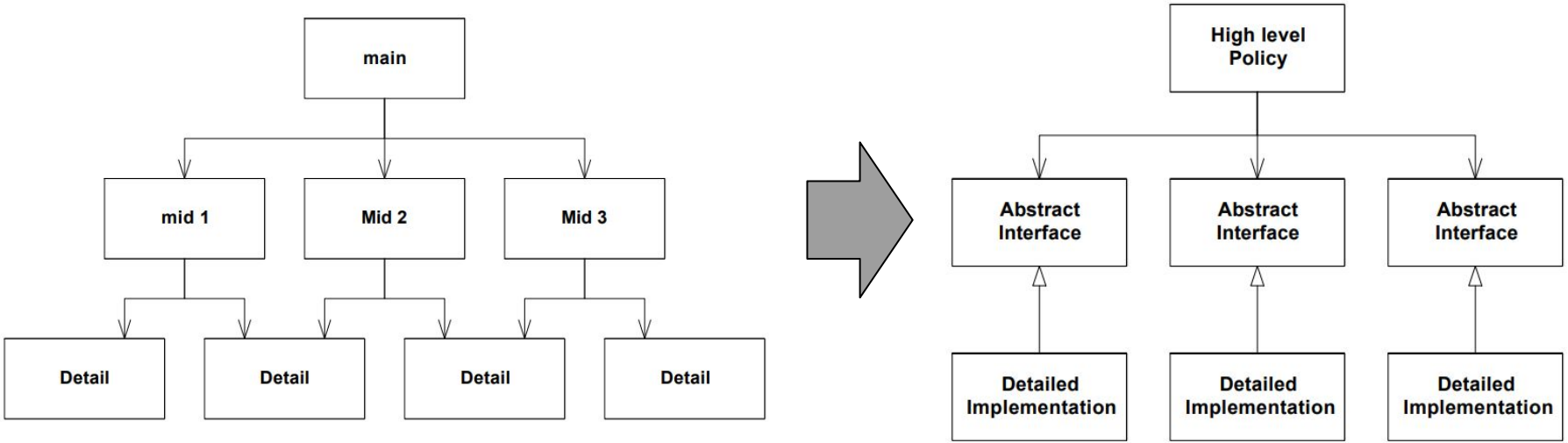
Нарушения LSP одновременно указывают на нарушение OCP.

Dependency Inversion Principle (DIP)

"**Depend upon abstractions**, not upon concretions".

Это основной механизм достижения **OCP** – он постулирует необходимость **направлять все зависимости на абстрактные классы** или абстрактные функции.

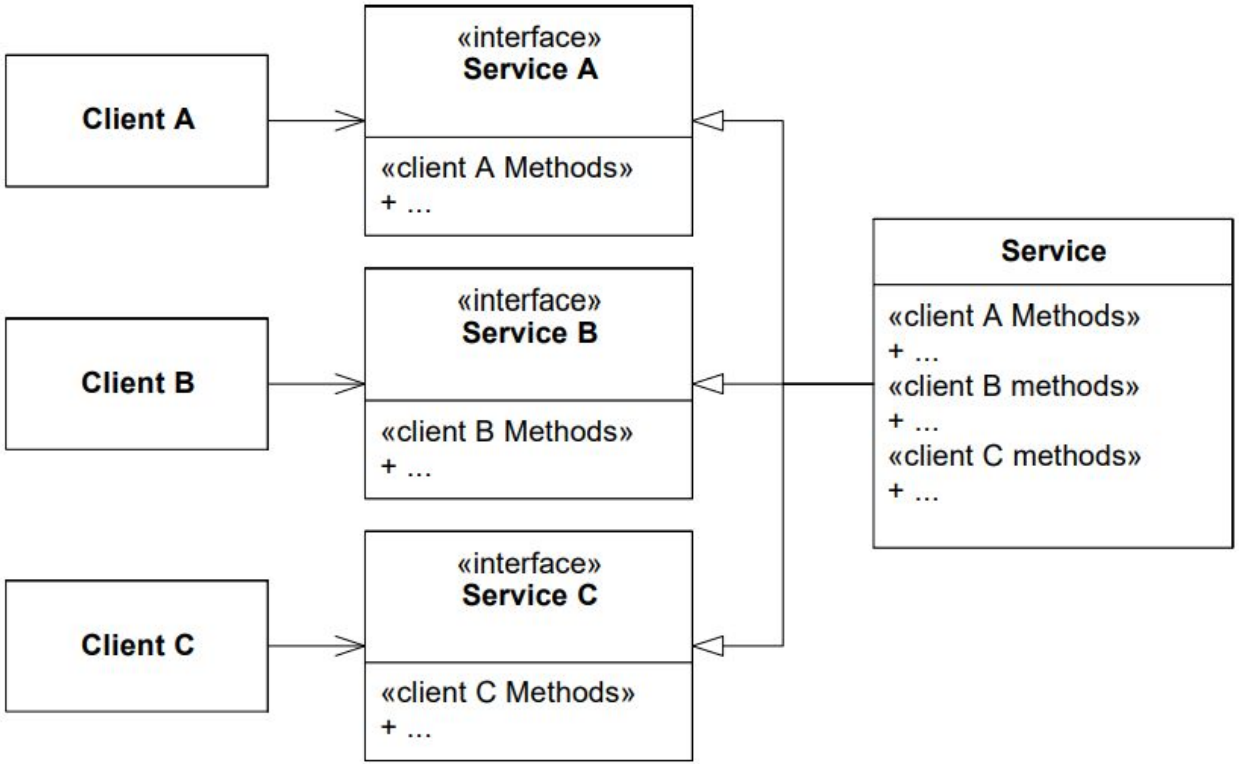
Реализации тоже зависят от абстракций (DIP)



Interface Segregation Principle (ISP)

"Many **client-specific interfaces are better** than one general purpose interface".

Множественное наследование позволяет клиентам **зависеть от нескольких интерфейсов**, которые реализуются в одном модуле/классе.


Client A
«interface»
Service A
«client A Methods»
+ ...
Client B
«interface»
Service B
«client B Methods»
+ ...
Client C
«interface»
Service C
«client C Methods»
+ ...
Service
«client A Methods»
+ ...
«client B methods»
+ ...
«client C methods»
+ ...

Single Responsibility Principle (SRP)

"Each module should have **only one reason to change**".

Декомпозиция системы должна начинаться не со связей между модулями, а с дизайнерских решений, которые имеют **большую вероятность измениться** в будущем. Тогда дизайн отдельного модуля выстраивается вокруг сокрытия такого решения (см. "separation of concerns").

https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html

“Заказчик” (stakeholder) решает (SRP)

“Сущности, которые **изменяются по одной** и той же **причине**, должны быть **сгруппированы вместе**. Сущности, которые **изменяются по разным причинам**, должны быть **разделены**.”

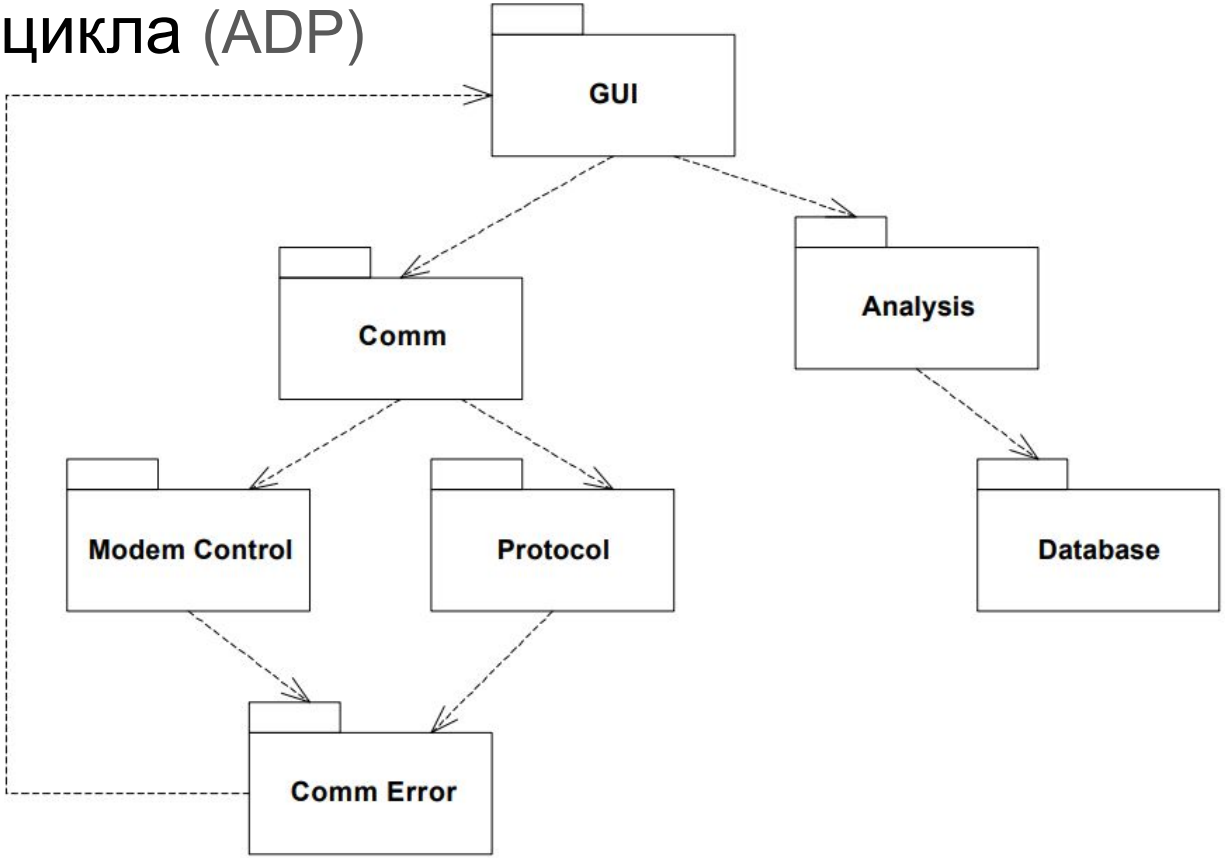
“Package cohesion” principles

- Release Reuse Equivalency Principle (REP)
  - “The granule of reuse is the **granule of release**.”
- Common Closure Principle (CCP)
  - “Classes that **change together**, belong together.”
- Common Reuse Principle (CRP)
  - “Classes that **aren’t reused together** should not be grouped together.”

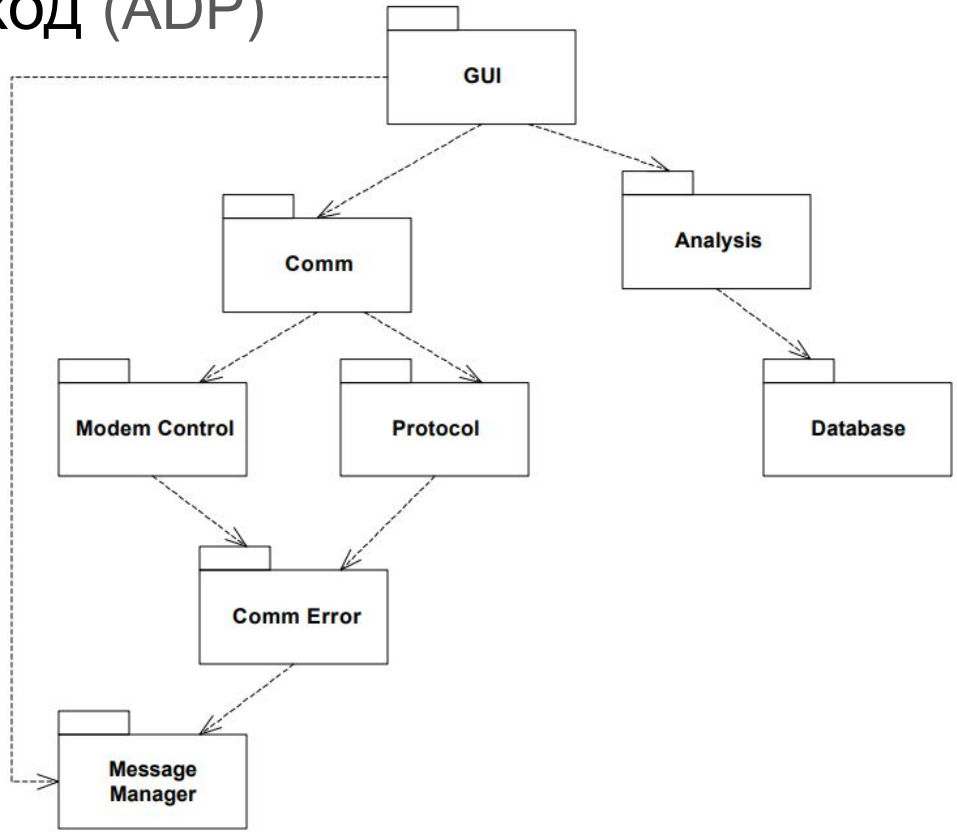
“Package coupling” principles

- Acyclic Dependency Principle (ADP)
  - “The **dependencies** between packages **must not form cycles**.”
- Stable Dependencies Principle (SDP)
  - “Depend in the direction of **stability**.”
- Stable Abstractions Principle (SAP)
  - “Stable packages should be **abstract** packages.”

Пример цикла (ADP)



Первый подход (ADP)



Второй подход (ADP)

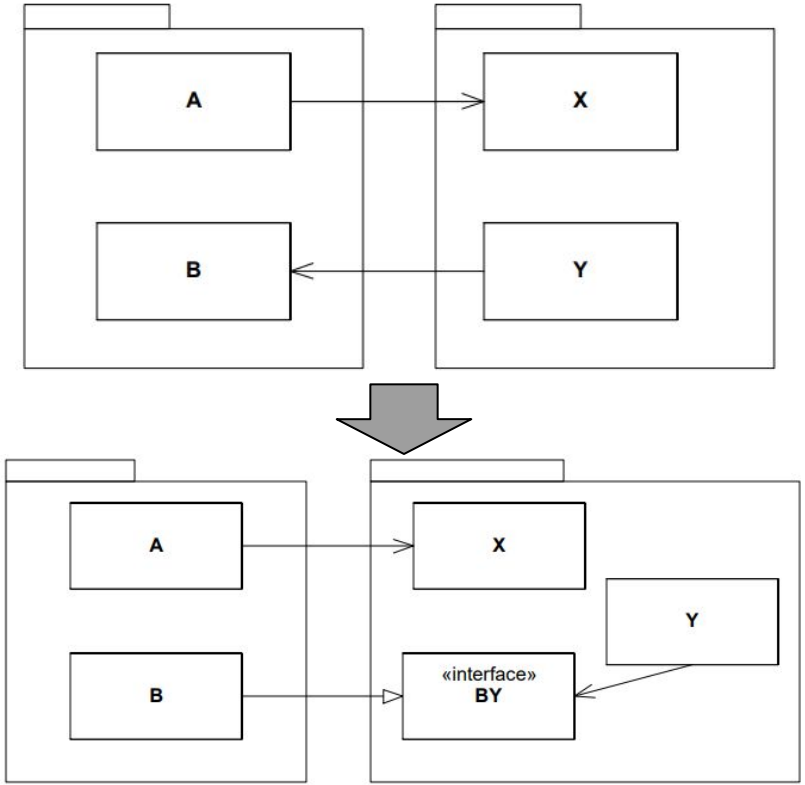
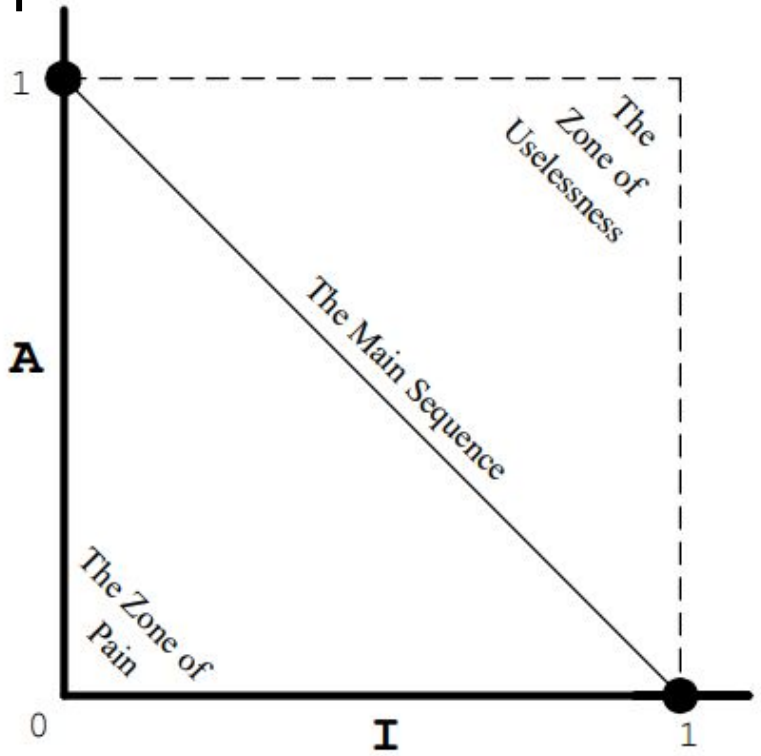


Диаграмма абстрактности/нестабильности



Другие “принципы”

- “Чем хуже, тем лучше”
- KISS (Keep It Simple Stupid)
- DRY (Don’t Repeat Yourself)
- WET (Write Everything Twice)
- YAGNI (You Aren’t Gonna Need It)
- HATEOAS (Hypermedia Is The Engine Of App State)

и др.

https://youtu.be/7YpFGkG-u1w?t=1413

"Object-oriented programming is **an exceptionally bad idea** which could only have originated in California."

E. Dijkstra

https://www.reddit.com/r/compsci/comments/ajx7t/askcompsci_why_is_according_to_edsger_dijkstra/