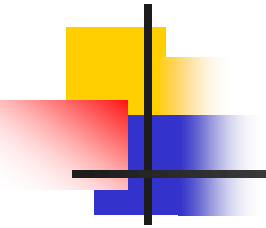
Объектно-ориентированный Анализ и Дизайн

Часть 2

ОО Анализ: Аналитическая модель

Переход от анализа к дизайну

Принципы ОО дизайна



5. OO Анализ

- Цели анализа и дизайна
- Аналитическая модель
- Аналитические классы и отношения
- Реализация use-cases
- Диаграммы деятельностей и состояний
- Диаграммы взаимодействия
- Трансформация анализа в дизайн

Цели анализа и дизайна

Задачи:

- Трансформировать требования, собранные на предыдущем этапе, в дизайн системы
- Проработать архитектуру системы
- Адаптировать дизайн к среде исполнения

Модели:

- Аналитическая модель (Analysis model)
- Дизайн модель (Design model)

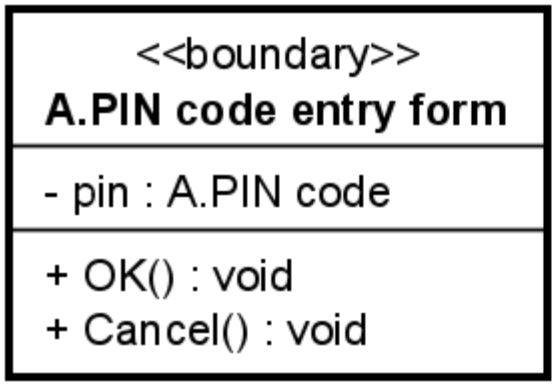
Аналитическая модель

- Абстрактная модель системы, описывающая ее в терминах **use-case realization**. Язык реализации классов не фиксируется. Обычно не сопровождается.

- Элементы аналитической модели:
  - **Use-case realization** – реализация use-case, набор activity, state, collaboration и class диаграмм
  - **Boundary class** – класс, разграничивающий actor-ов и систему
  - **Control** – класс, управляющий другими классами
  - **Entity** – класс, моделирующий информацию, используемую в системе

Boundary class

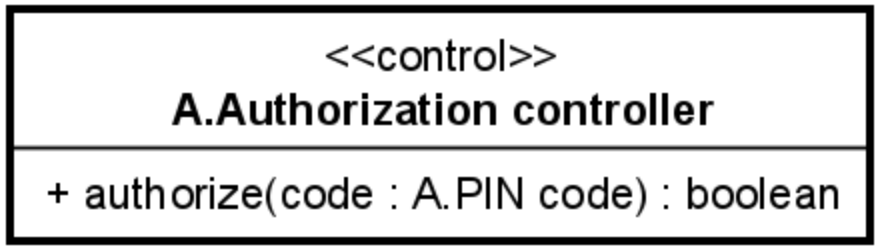
- Класс, разграничивающий (под-)систему и окружение.
- UML: class со стереотипом <<boundary>>
- Примеры: классы пользовательского интерфейса, классы интерфейсов систем и устройств



Представление boundary посредством стереотипа и пиктограммы

Control

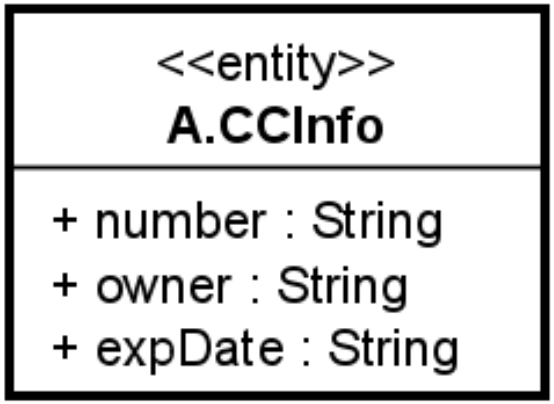
- Класс, управляющий другими классами. Можно сказать, что control “исполняет” use-case.
- UML: class со стереотипом <<control>>



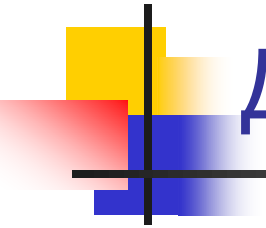
Представление control посредством стереотипа и пиктограммы

Entity

- Класс, моделирующий информацию, используемую в системе.
- UML: class со стереотипом <<entity>>
- Примеры: документы, данные пользователей



Представление entity посредством стереотипа и пиктограммы



Диаграммы взаимодействия

- Последовательностей - Sequence diagrams
- Кооперaций - Collaboration diagrams

- Отражают динамические аспекты поведения объектов
- Семантически эквивалентны
- Содержат:
  - Объекты
  - Связи
  - Сообщения
  - Потоки данных

Авторизация в банкомате

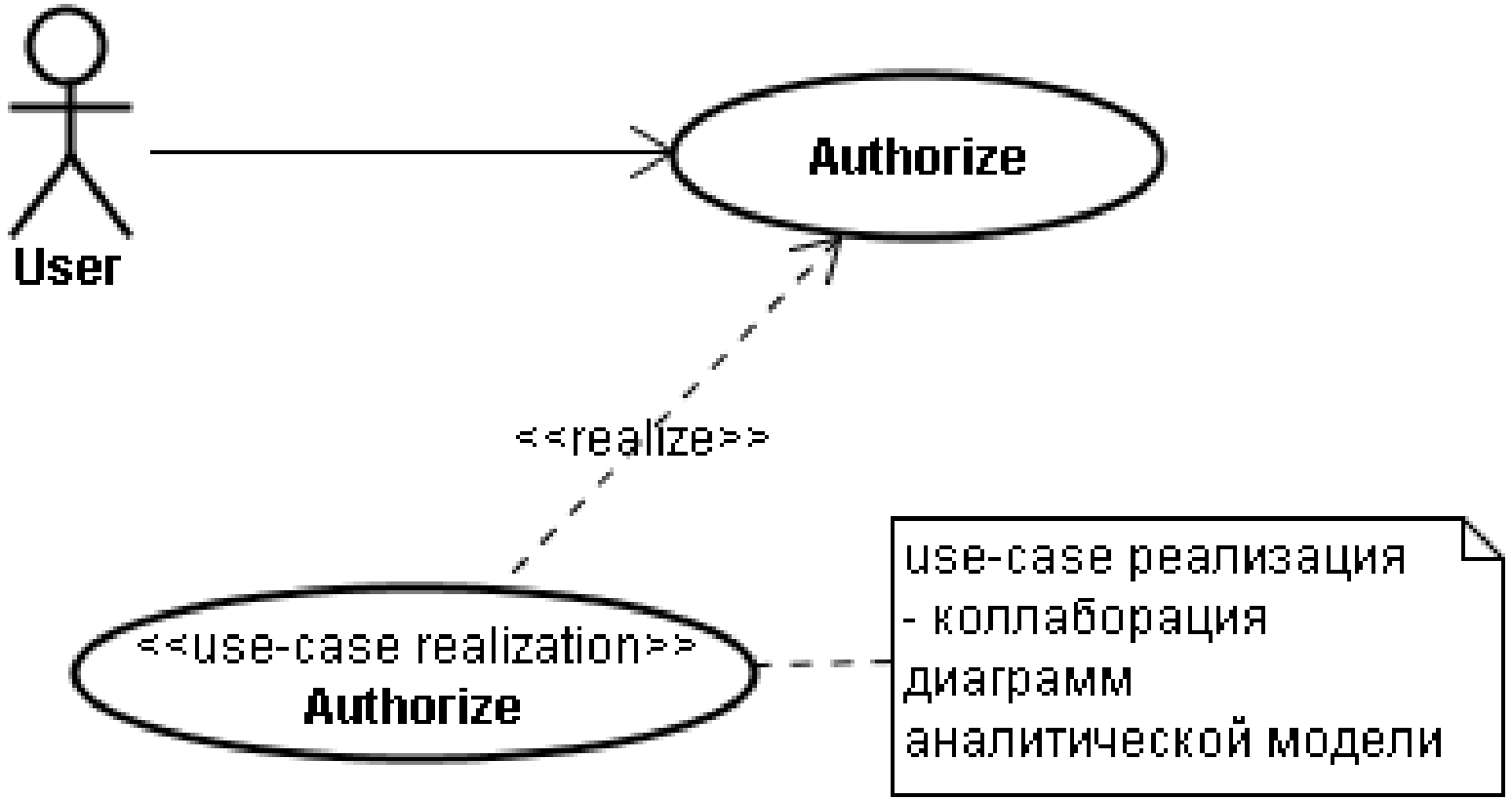


Диаграмма аналитических классов

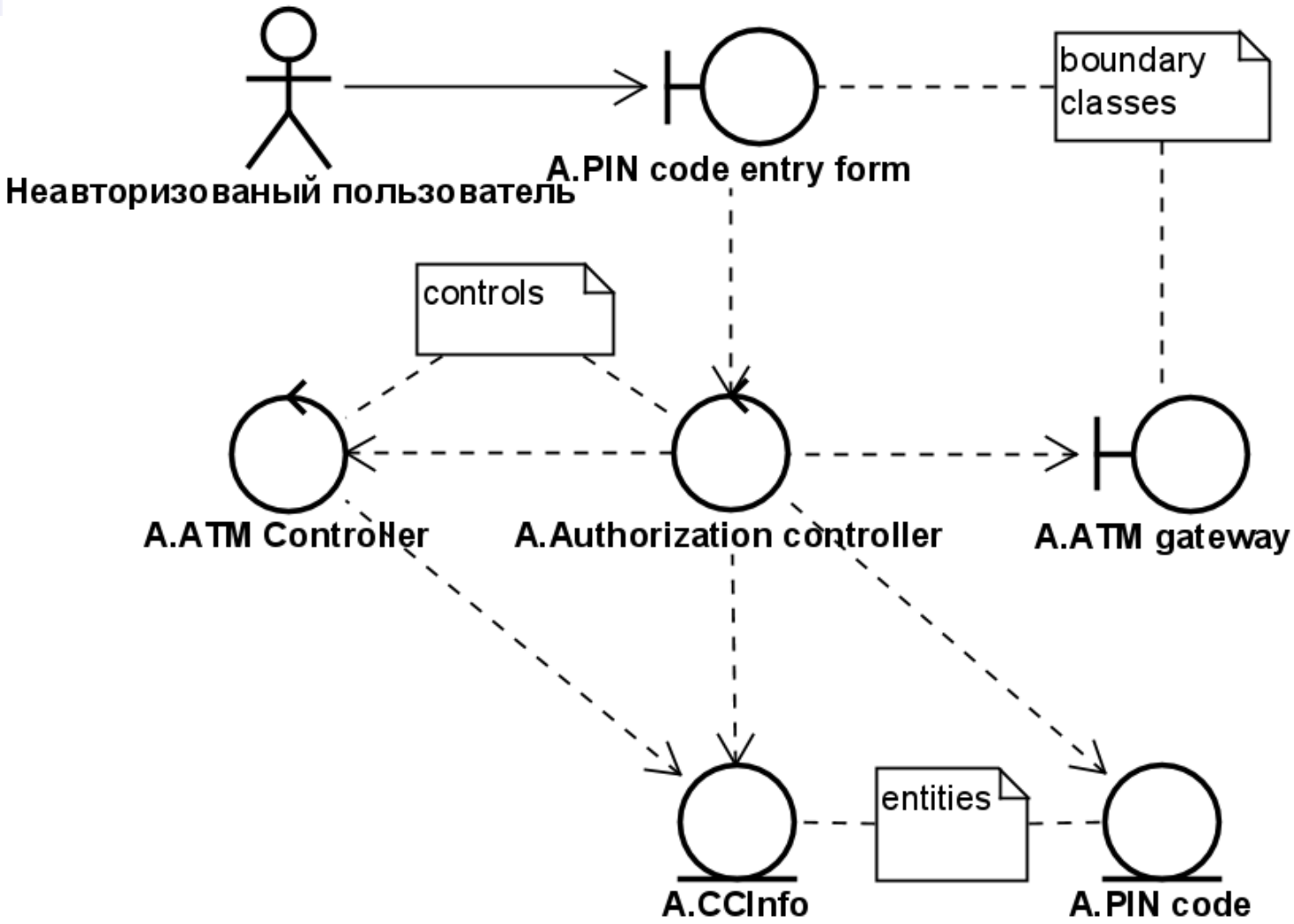


Диаграмма последовательностей

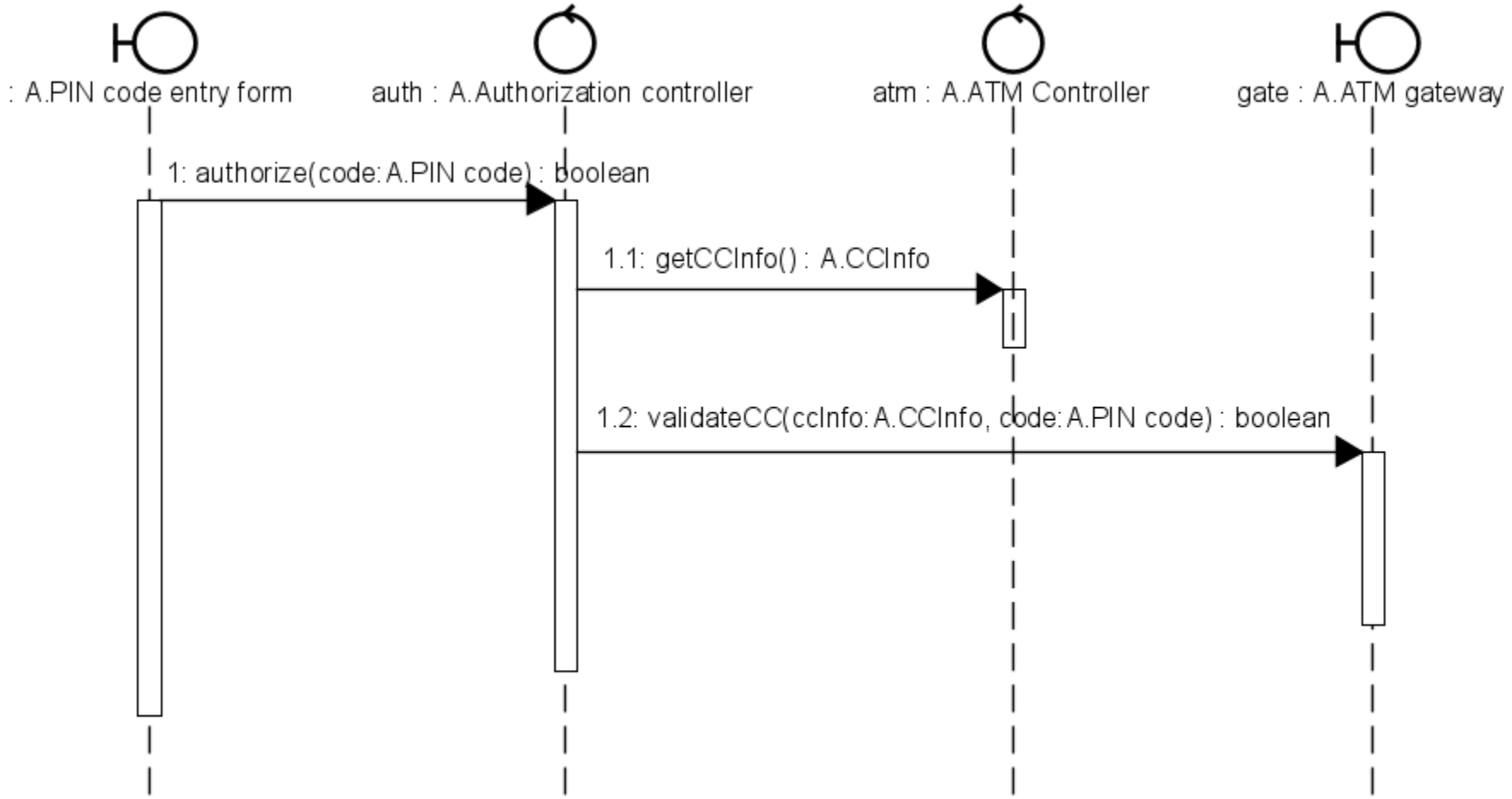
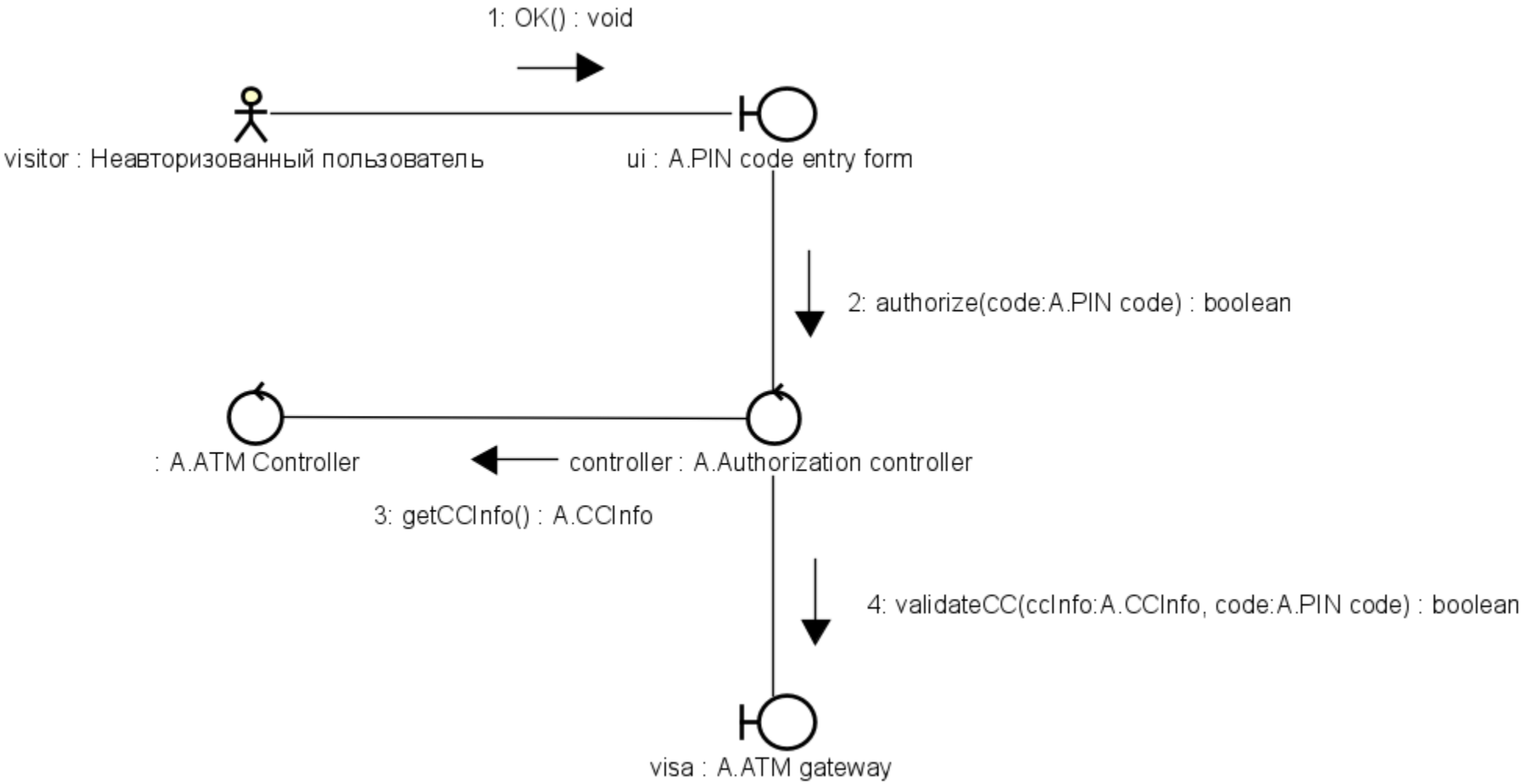


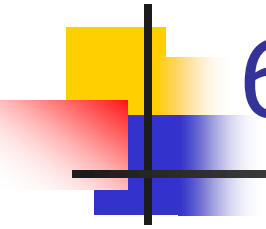
Диаграмма кооперацій



Ограничения на связи

From\To (navigability)	Boundary	Entity	Control
Boundary	yes	yes	yes
Entity	**no***	yes	**no***
Control	yes	yes	yes

* Используйте обратные связи со стереотипом “subscribe-to”



6. ОО-дизайн

- Дизайн классов
- Дизайн пакетов
- Поиск и применение шаблонов

Цели дизайна

- Адаптировать аналитическую модель к конкретным языкам и технологиям, выбранным для реализации системы, т.е:
  - Завершить проработку архитектуры системы (выбор платформ, технологий, библиотек, компонентов, протоколов...)
  - Проработать дизайн (слои, пакеты, межпакетные интерфейсы, ключевые абстракции)
  - При этом обеспечить:
    - Соответствие нефункциональным требованиям
    - Тестопригодность
    - Расширяемость системы (extensibility)
    - Легкость поддержки (maintainability)
    - Создание переиспользуемых компонент (reusability)

Дизайн модель

- Модель реализации системы. Создается на основе аналитической модели. Фиксирует язык реализации классов и используемые API. Сопровождается до конца разработки.

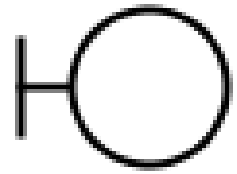
- Элементы дизайн модели:
  - **Layer** - слой (UI, UI logic, Business logic, Data, System)
  - **Subsystem** - подсистема
  - **Package -** пакет
  - **Class** – класс
  - **Use-case realization** – коллекция диаграмм

Переход от анализа к дизайну

- Аналитический класс при переходе к дизайну трансформируется в один или несколько классов дизайн модели, которые реализуются на каком-либо конкретном языке программирования.

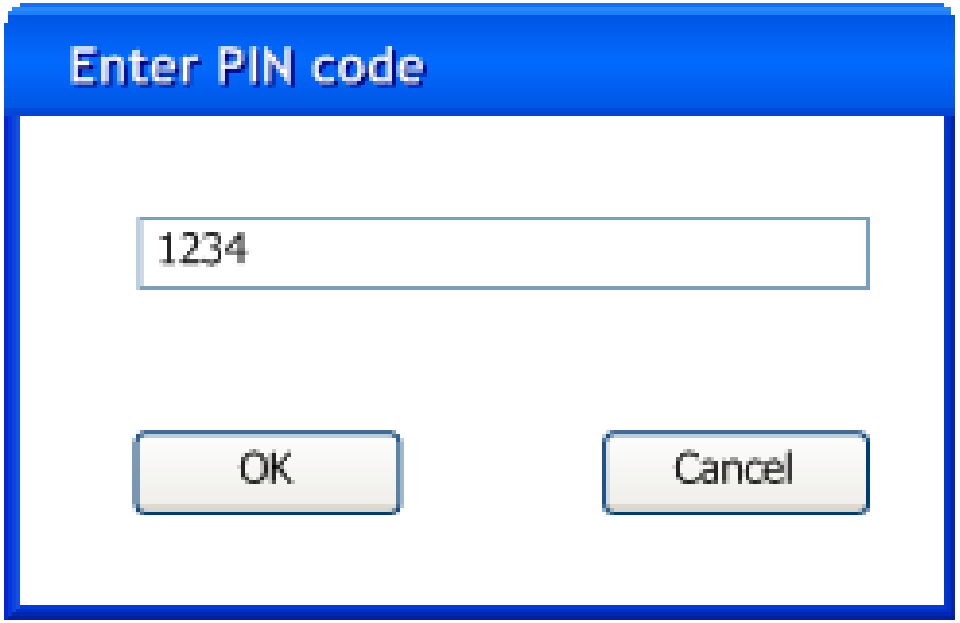
Трансформация boundary

- Классы пользовательского интерфейса



A.PIN code entry form

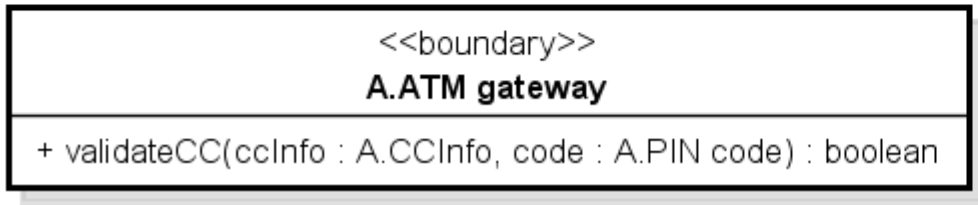
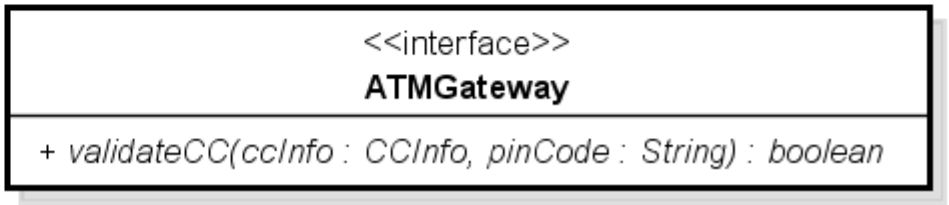
<<boundary>> **A.PIN code entry form**
- pin : String
+ OK() : void + Cancel() : void



- Сколько объектов скольких классов вы можете найти на форме ввода PIN кода справа?

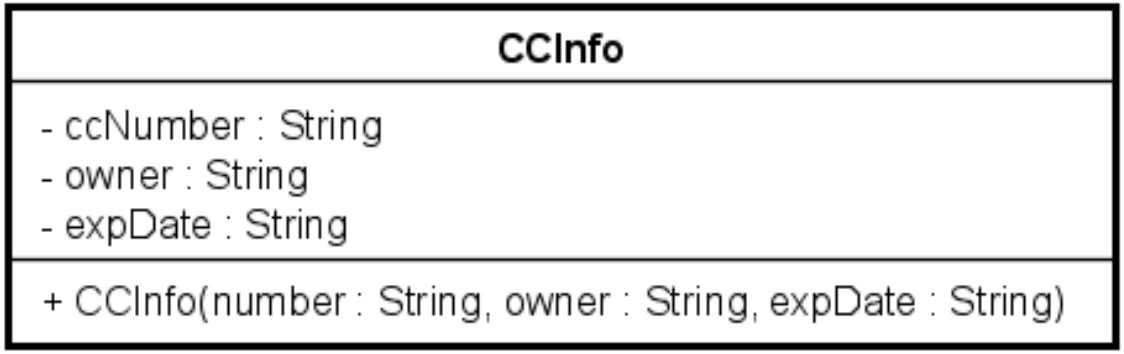
Трансформация boundary

- Интерфейс (для) внешней системы

Аналитическая модель	Дизайн модель
	
	

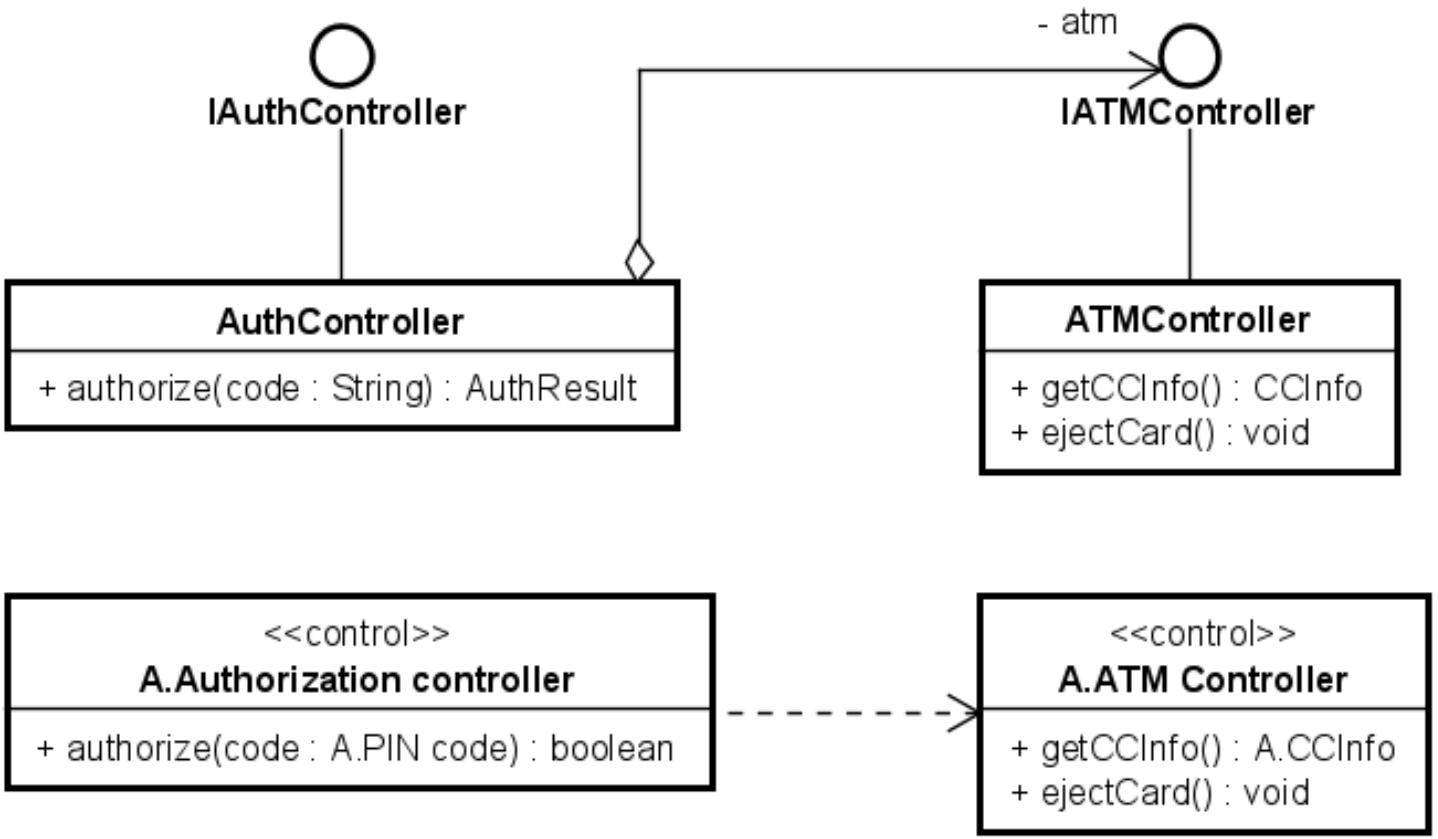
Трансформация entity

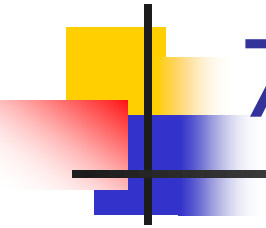
- Класс(ы) типов данных, специфичных для предметной области

Аналитическая модель	Дизайн модель
	

Трансформация control

- Один или несколько интерфейсов, реализованные пакетом или группой пакетов (подсистема).





7. Принципы ОО дизайна

- Вопрос о том, как пишут хорошие программы на C++, похож на вопрос о том, как пишут хорошую английскую прозу.

Б.Страуструп

Принципы дизайна классов

- **SRP** Принцип единственности
  (Single Responsibility Principle)
- **LSP** Принцип подстановки
  (Liskov Substitution Principle)
- **LoD** Закон Деметры
  (Law of Demeter)
- **OCP** Принцип закрытости абстракции
  (Open-Closed Principle)
- **ISP** Принцип разделения интерфейсов
  (Interface Segregation Principle)

SRP* – Принцип единственности ответственности

Класс должен обладать единственной ответственностью,

- ✓ реализуя ее полностью,
- ✓ реализуя ее хорошо,
- ✓ реализуя только ее

A class has a single responsibility:

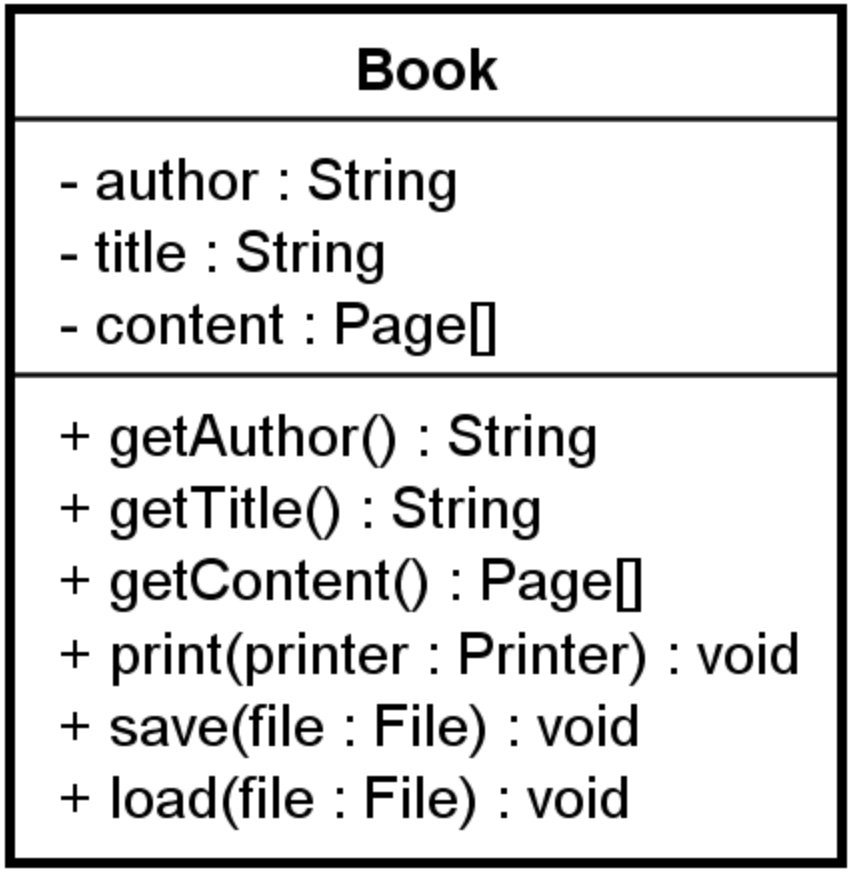
- ✓ it does it all,
- ✓ it does it well,
- ✓ it does it only

A class should have only one reason to change.

- Robert C. Martin

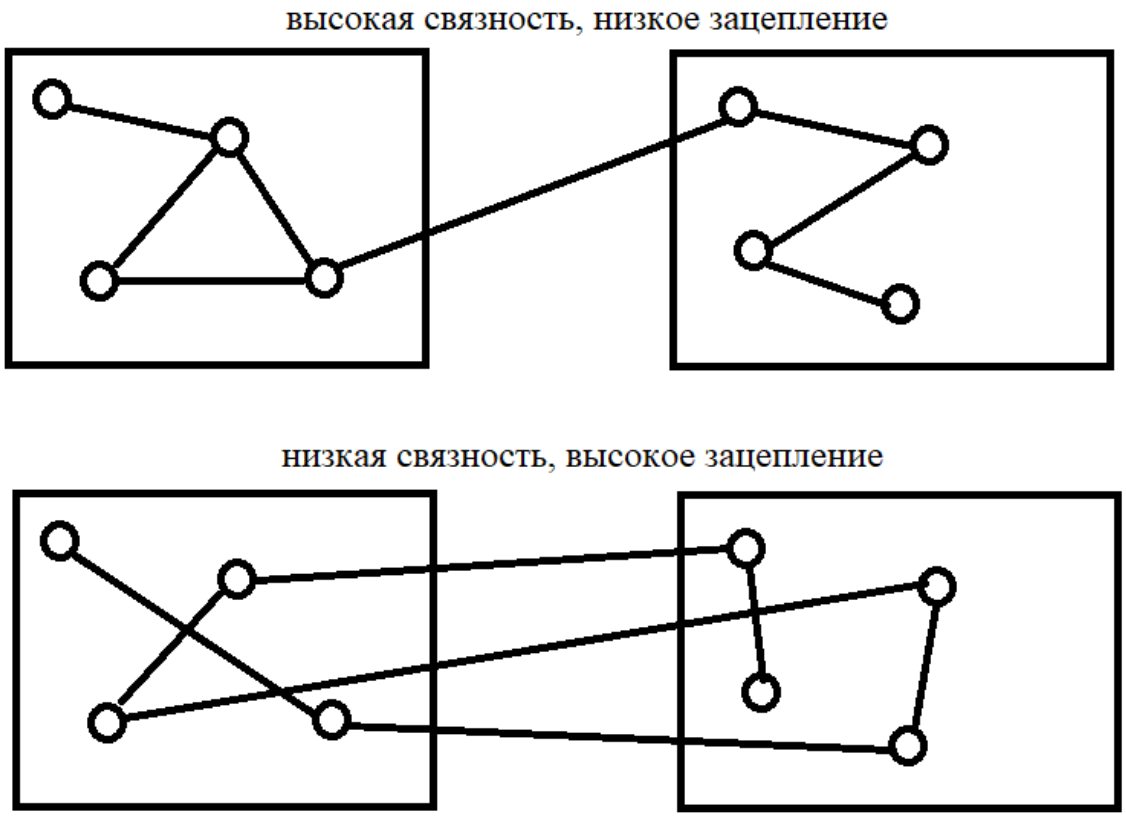
* **SRP** – **S**ingle **R**esponsibility **P**rinciple

Нарушение SRP



ВОПРОС: Чем именно плох этот класс с точки зрения SRP?

SRP: связность и зацепление



- Улучшает связность (*cohesion*)
- Помогает уменьшить зацепление (*coupling*)
- Позволяет избежать антипаттерна God Object

LSP* – Принцип подстановки

Поведение методов, принимающих в качестве параметра указатели и ссылки на объекты базового класса, не должно зависеть от того, к какому классу (базовому или любому из производных) принадлежит переданный объект.

- Robert C. Martin

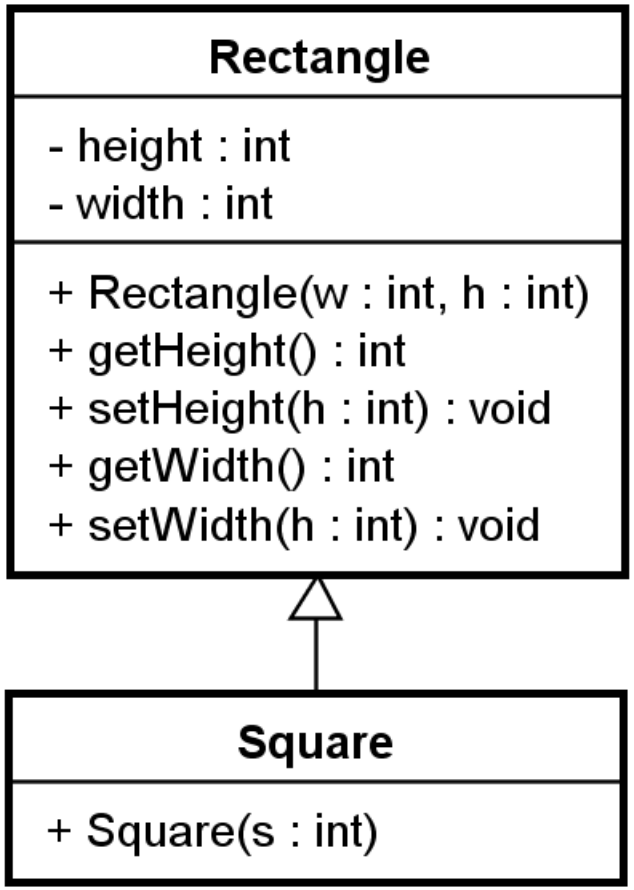
Оригинальная формулировка:

If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T the behavior of P is unchanged when o1 is substituted by o2 then S is a subtype of T.

- Barbara Liskov

* **LSP** – **L**iskov **S**ubstitution **P**rinciple

Нарушение LSP: Фигуры

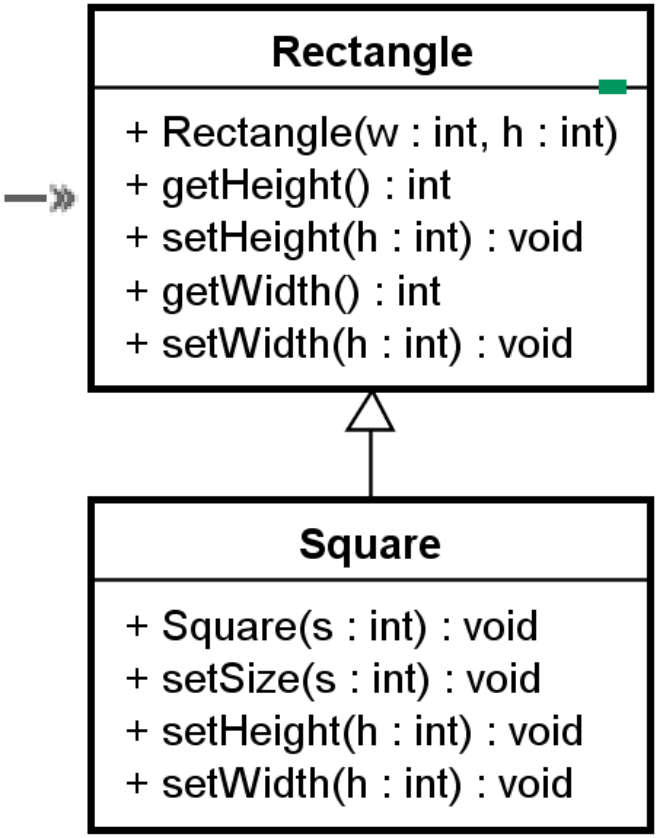



```
class Rectangle
{
  private int h;
  private int w;
  public Rectangle( int w, int h )
    { this.h = h; this.w = w; }
  public void setHeight( int h ) { this.h = h; }
  public int    getHeight()        { return h; }
}
class Square extends Rectangle
{
  public Square( int s ) { super( s, s ); }
}
```

Проблема:

```
Square s = new Square(5);
s.setHeight(6); // Объект s перестал быть квадратом
```

Попробуем исправить проблему



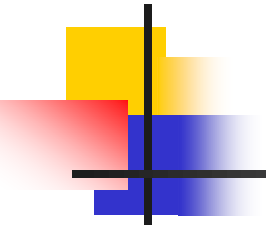

```
class Square extends Rectangle
{
  public Square( int s ) { super( s, s ); }
  public void setSize( int s ) {
    super.setHeight(s);
    super.setWidth(s);
  }
  public void setHeight( int h ) { setSize(h); }
  public void setWidth( int w ) { setSize(w); }
}
```

А что если кто-то напишет функцию f, а затем в нее передадут Square?

```
void f( Rectangle r ) throws Exception {
  r.setHeight(4);
  r.setWidth(5);
  if( r.getHight() * r.getWidth() != 20 ) throw new Exception( “Bug!” );
}
```

LSP: в чем проблема?

- С точки зрения ОО подхода наш квадрат просто **не является** прямоугольником, который задан классом Rectangle, потому что:
- Класс – абстракция данных *и поведения*
- При этом, *поведение* квадрата существенно отличается от поведения *такого* прямоугольника
- Генерализация – отношение «частное-общее», но *общность* нужно искать в *поведении*, а не в структуре данных или форме объектов.

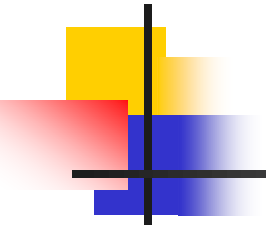


LSP: все ответы здесь

- Do not try to bend the spoon. That's impossible. Instead... only try to realize the truth.
- What truth?
- There is no spoon.

THE MATRIX

В абстракциях предметной области нет никакого «наследования», есть генерализация и реализация интерфейса.



LoD – Law of Demeter

Метод должен обладать ограниченым знанием об объектной модели приложения.

- D. Rumbaugh

Оригинальная формулировка:

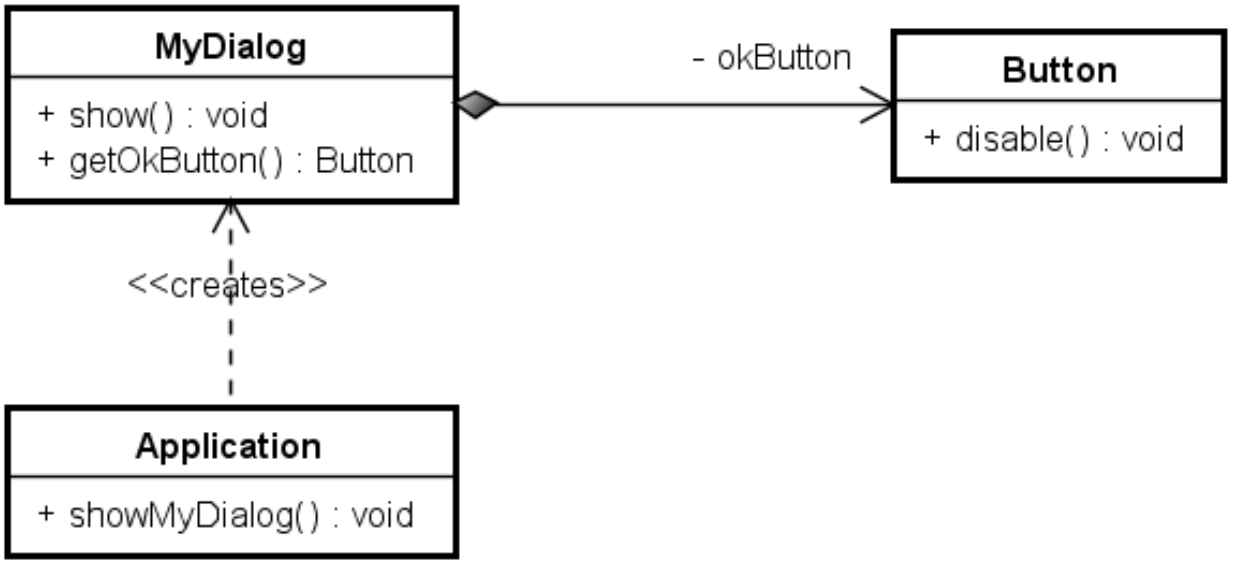
Only talk to your immediate friends. Never talk to strangers.

- Ian Holland

Друзья (friends) метода **f** :

- методы класса **f** и классы параметров метода **f**
- методы классов - полей класса **f**
- методы классов объектов, создаваемых в **f** .

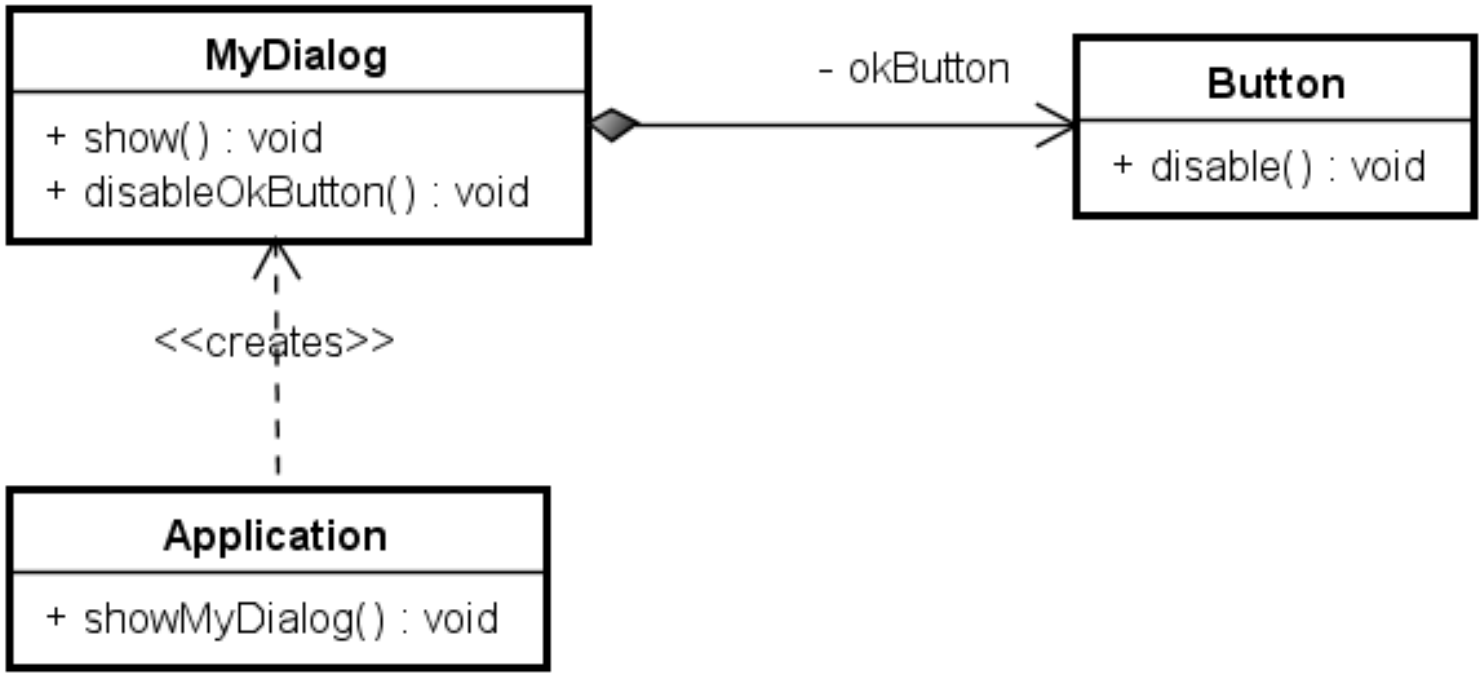
Нарушение LoD



Проблема:
```
public void showMyDialog() {
  MyDialog dialog;
  dialog.getOkButton().disable();
}
```

1. связь Application - > Button мы не планировали.
2. Замена класса Button на другой интерфейсный класс может потребовать изменений в Application (если disable() в новом классе называется по другому)

LoD-совместимый дизайн



Решение: void showMyDialog() {
…. // создание MyDialog
dialog.disableOkButton(); // секрет класса MyDialog останется секретом
}

OCP – Принцип закрытости абстракции

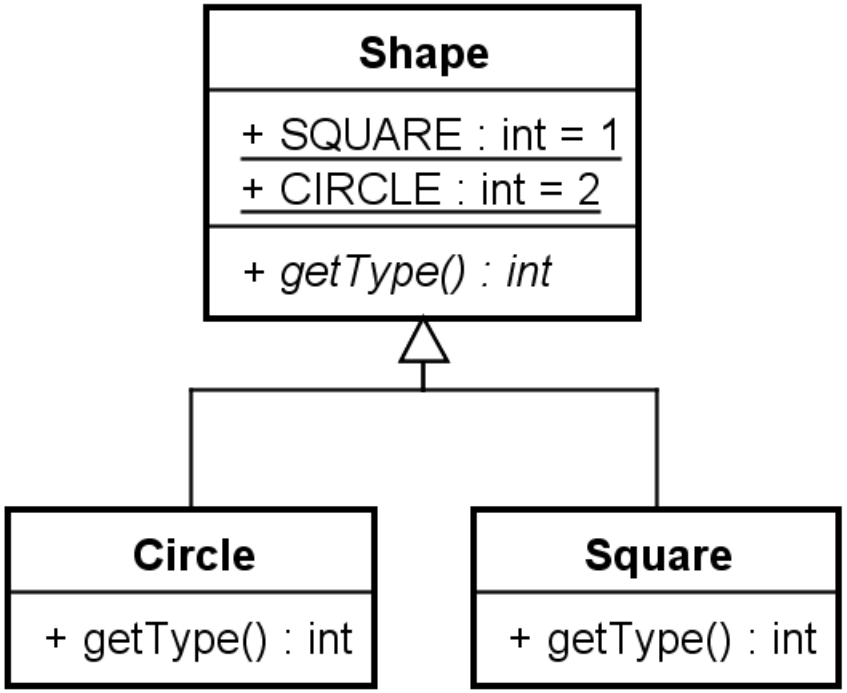
✓ Компоненты программной системы (классы, модули, методы) должны быть открыты для расширения, но закрыты от модификации.

- B. Meyer

✓ Не давая каких-либо общих рецептов, этот принцип заставляет нас думать о возможных *изменениях* кода под воздействием изменяющихся требований

✓ Смысл здесь в том, что добавление новой функциональности должно скорее приводить к **добавлению** нового кода, нежели к модификации существующего. В идеале – только к написанию новых классов.

Нарушение OCP: Фигуры

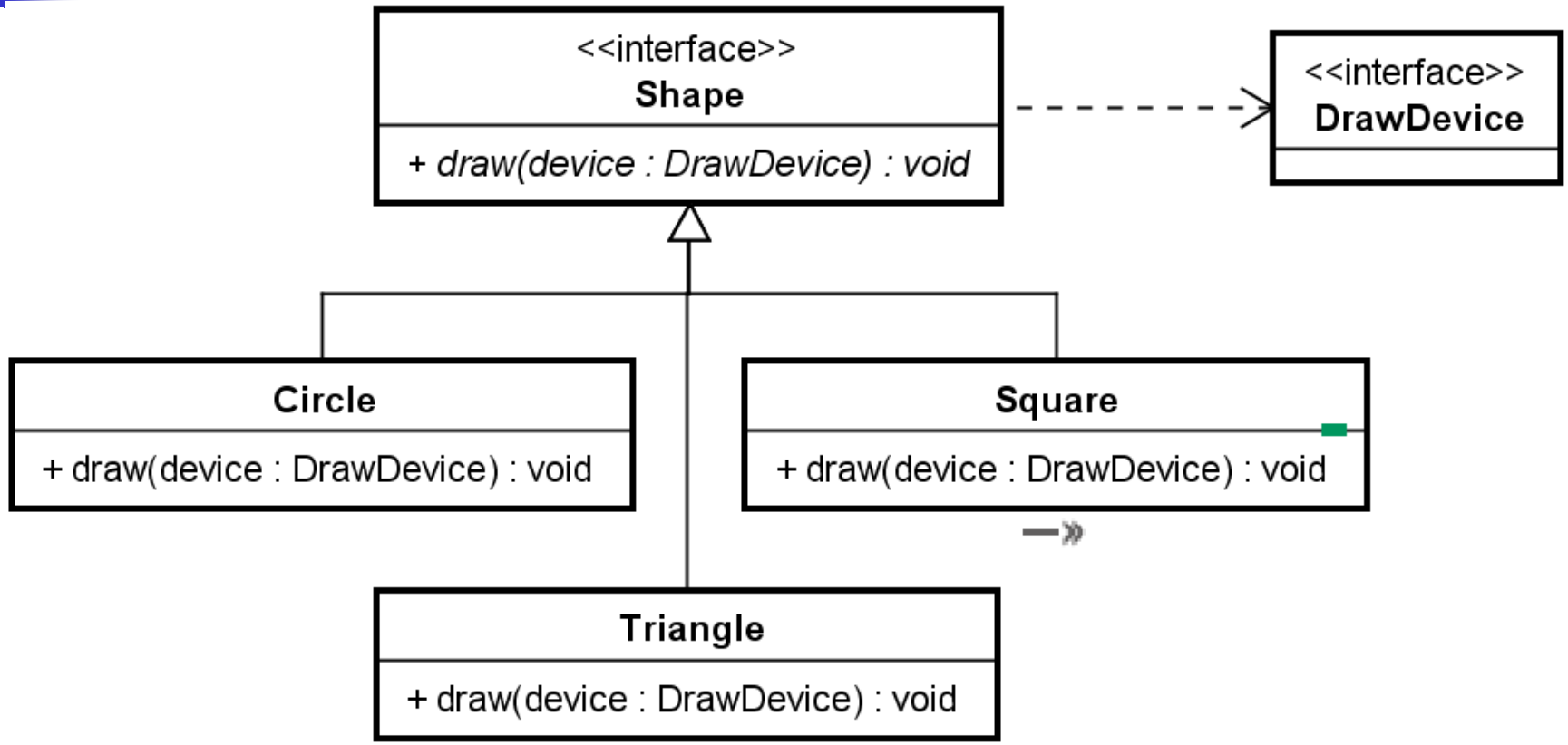



```
void drawShapes( Shape[] shapes )
{
  for( int i = 0; i < shapes.length; ++i )
  {
    if( shape[i].getType == Shape.SQUARE )
    {
      drawSquare( (Square)shape[i] );
    }
    else drawCircle( (Circle)shape[i] );
  }
}
```

Проблема:

Нельзя добавить в систему новый тип фигур, не изменив класса Shape и метода drawShapes().

OCP-совместимое решение




```
void drawShapes( Shape[] shapes ) {
  for( Shape shape : shapes ) shape.draw( device );
}
```

ISP – Разделение интерфейсов

Клиентов нельзя заставлять платить за сервисы, которых они не используют.

- Robert C. Martin

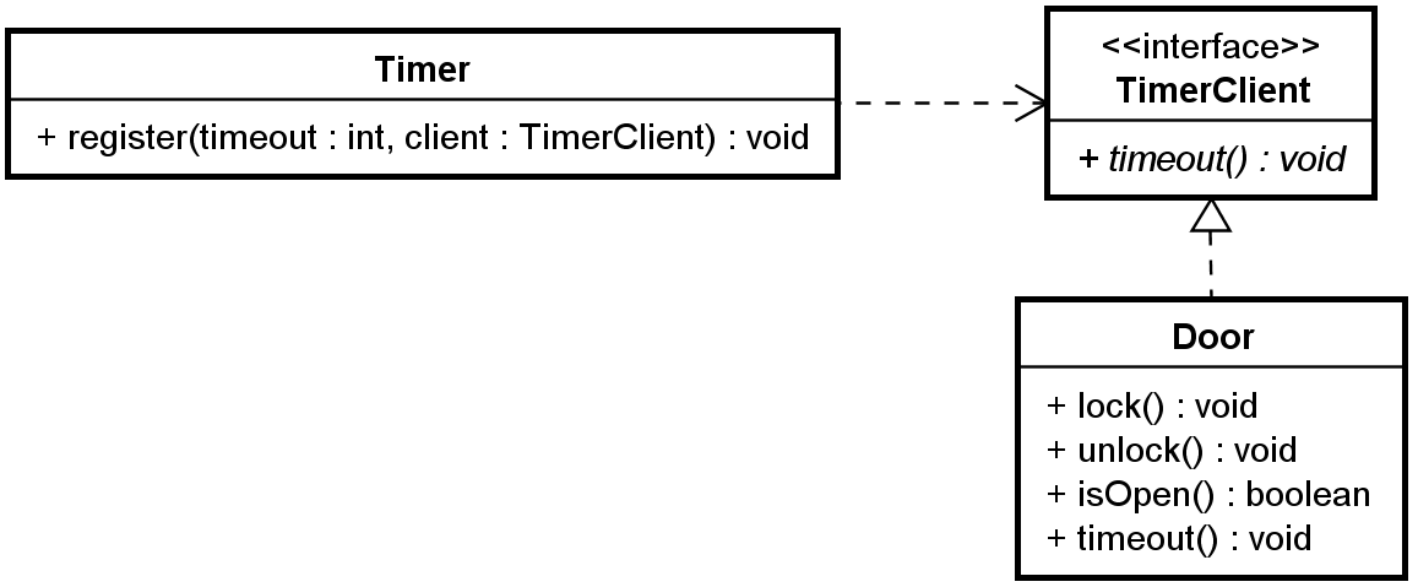
Hints:

- ✓ Избегайте «толстых» интерфейсов
- ✓ Разные клиенты – разные интерфейсы

Цена нарушения:

- ✓ Потеря гибкости

Нарушение ISP: Security Door

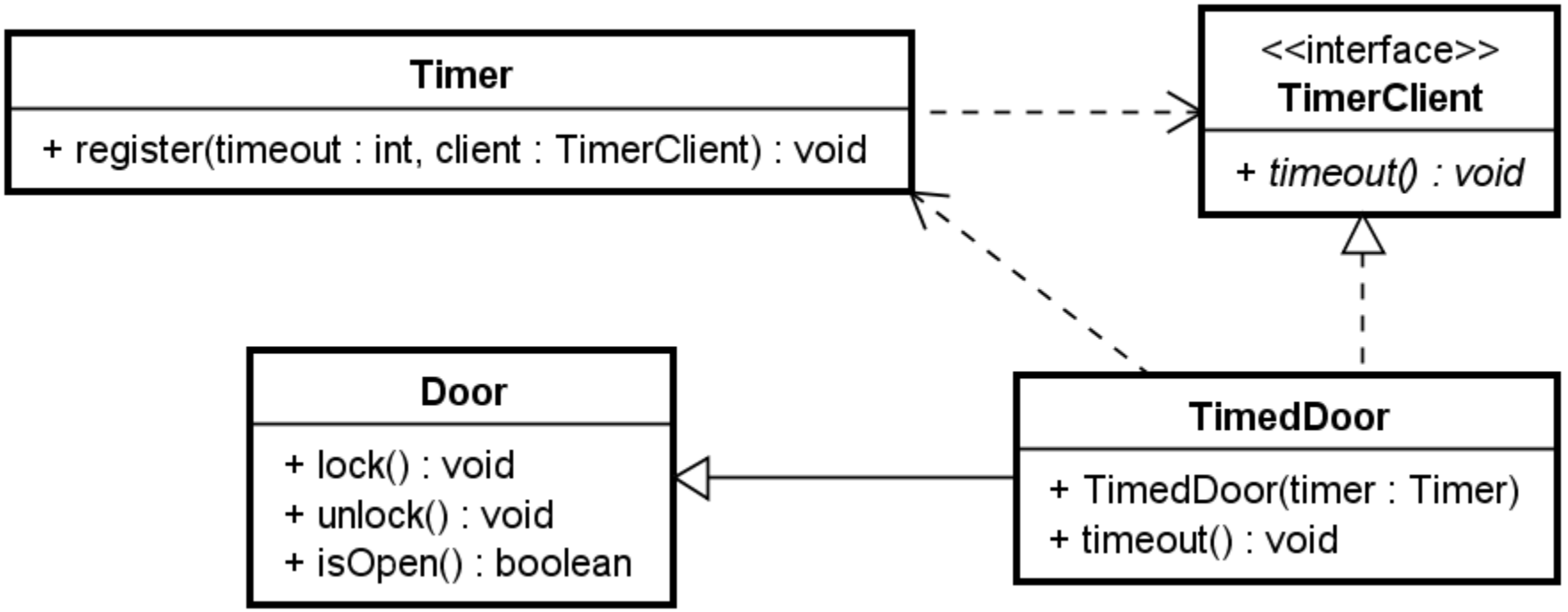


Дверь издает звук если открыта слишком долго.

Проблемы:

- Метод *timeout()* обязан быть public.
- Некоторые клиенты класса Door не используют и не должны использовать *timeout()*.
- *Может приводить к ошибкам.*

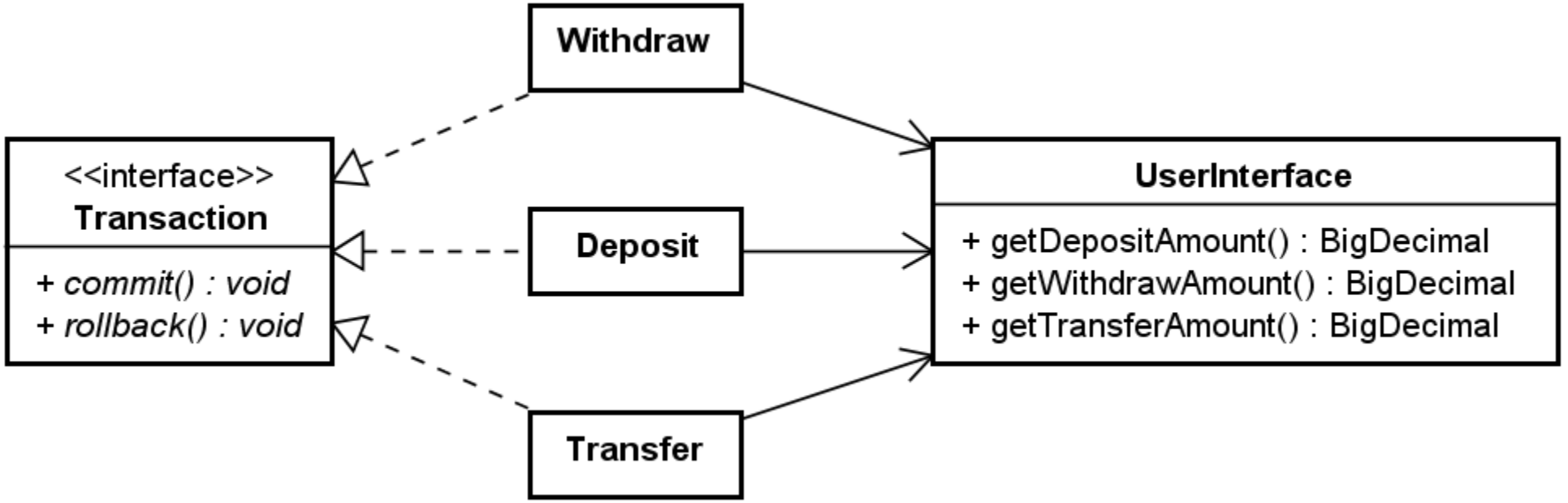
ISP-совместимая Security Door



✓ Клиенты Door могут использовать TimedDoor

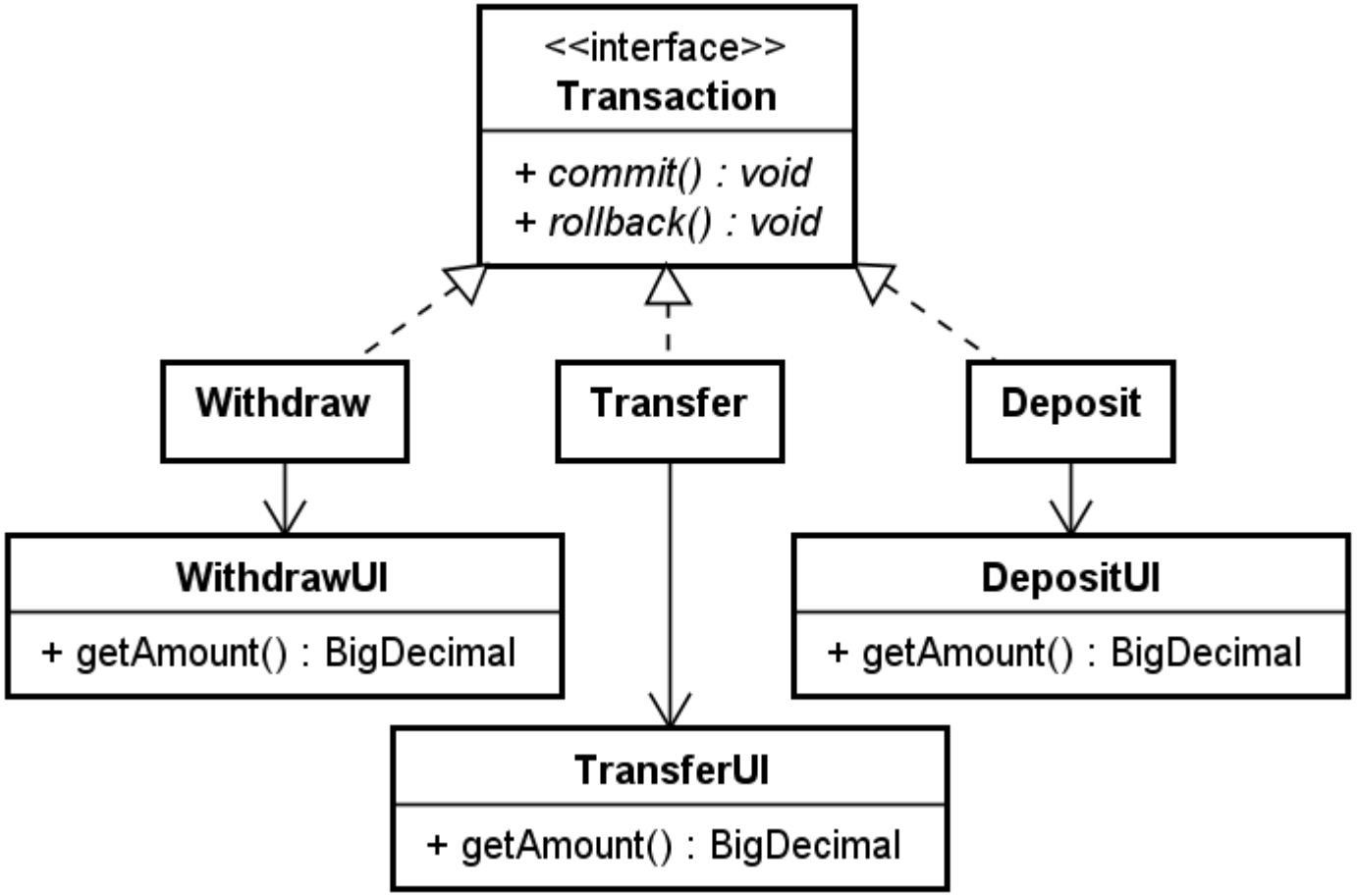
✓ Клиенты Door не будут зависеть от изменений в Timer, TimerClient или TimedDoor

Нарушение ISP: Банкомат

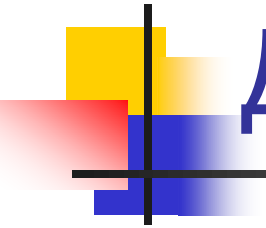


- ✓ Добавление новой транзакции потенциально затрагивает все остальные
- ✓ Если любая транзакция потребует изменений в UserInterface, остальные придется перетестировать

ISP-совместимое решение



Чем это лучше в контексте ISP?



Дизайн связей

- **DIP** Dependency Inversion Principle (инверсия зависимостей)
- **ADP** Acyclic Dependencies Principle (ацикличность зависимостей)

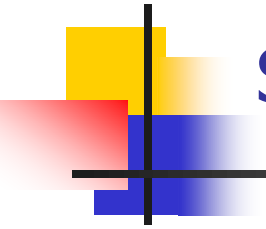
DIP* – Инверсия зависимостей

✓ Модули «высокого» уровня не должны зависеть от модулей «низкого» уровня. И те и другие должны зависеть от абстракций.

✓ Абстракции не должны зависеть от деталей реализации, напротив, детали реализации могут зависеть от абстракций.

- Robert C. Martin

* **DIP** - **D**ependency **I**nversion **P**rinciple



SLAP* – Принцип абстрагирования

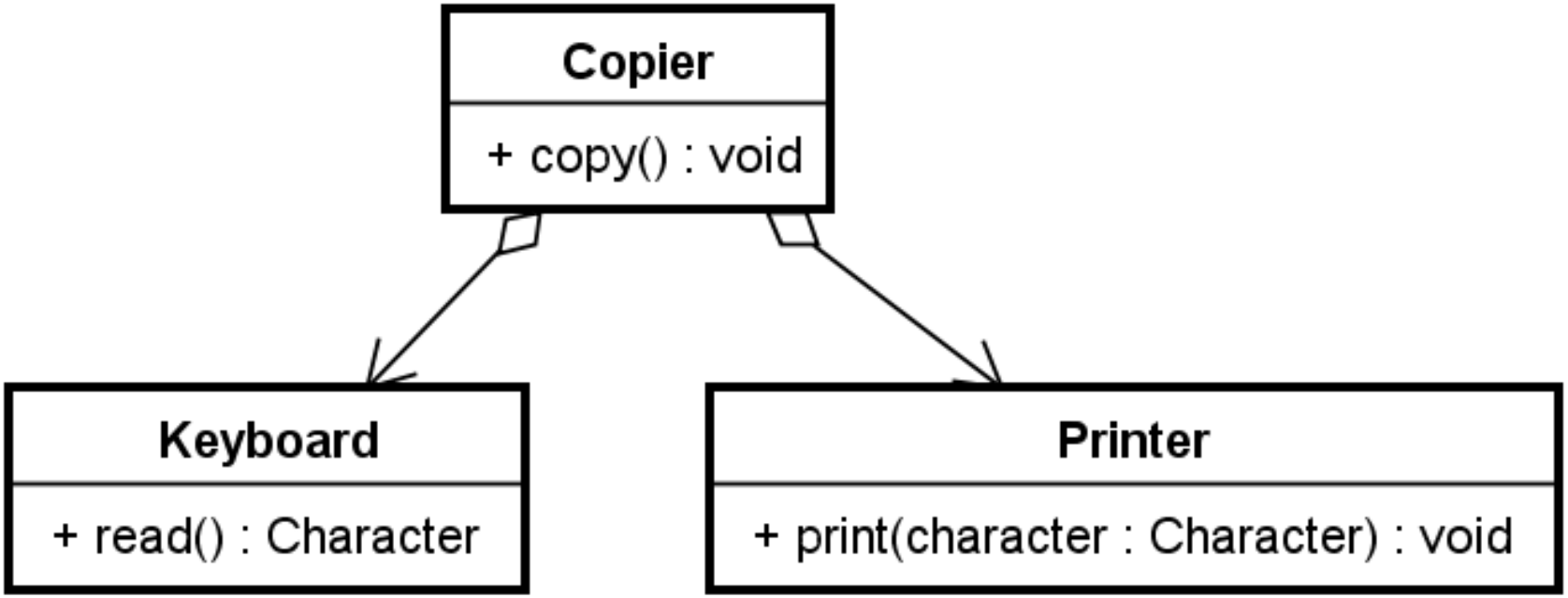
Keep lines of code within a method at the same level of abstraction via aggressive refactoring.

\- Neil Ford
The Productive Programmer, p.164

Все операторы метода должны находиться на одном уровне абстракции.

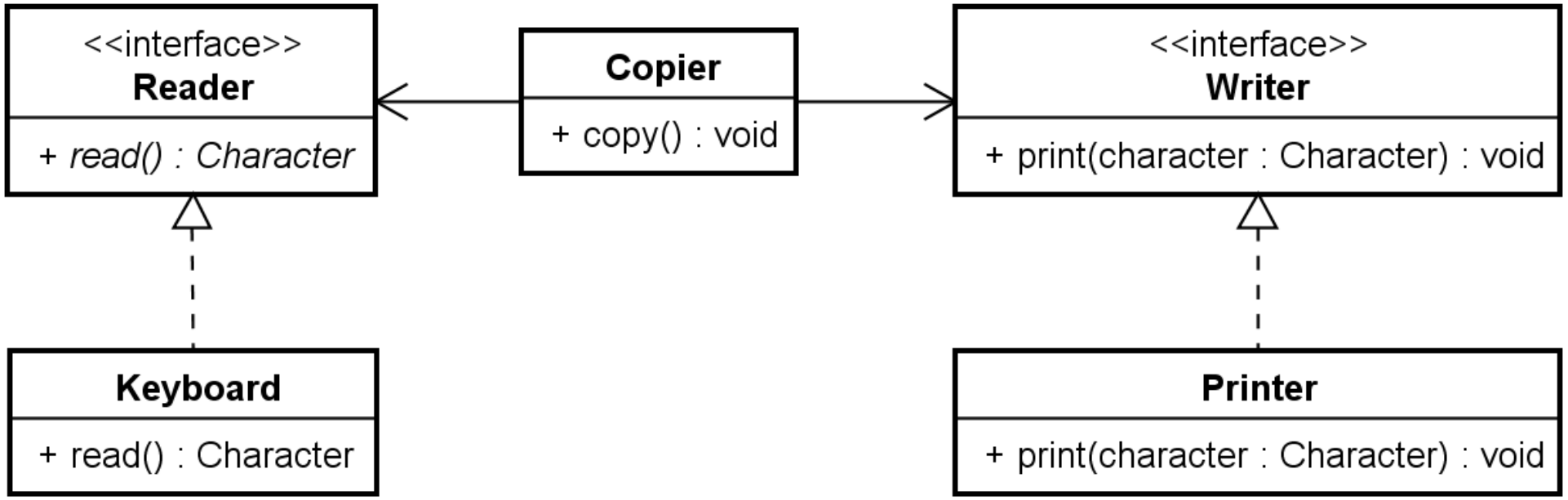
*** SLAP – S**ingle **L**evel of **A**bstraction **P**rinciple

Нарушение DIP: Copier

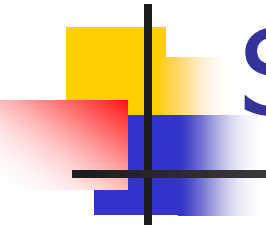


- Что делать, если требуется добавить еще одно устройство печати?
- Или другое устройство ввода?

DIP-совместимое решение



- ✓ Теперь легко добавляем новые устройства
- ✓ Никакие изменения в Keyboard или Printer не влияют на Copier



S.O.L.I.D.

S	**SRP**	Single Responsibility Principle
O	**OCP**	Open/Closed Principle
L	**LSP**	Liskov Substitution Principle
I	**ISP**	Interface Segregation Principle
D	**DIP**	Dependency Inversion Principle

ADP – Ациклические связи

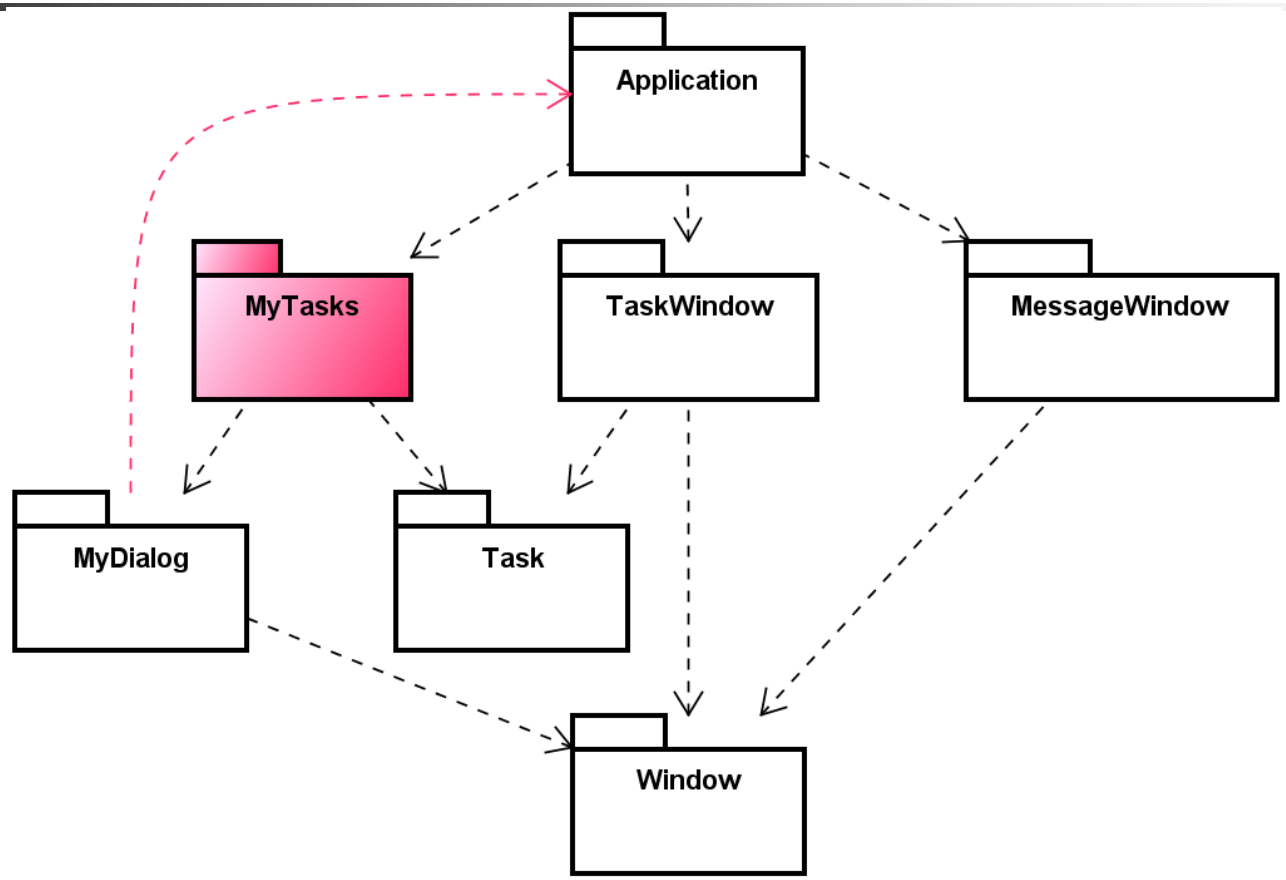
- Граф зависимостей между компонентами ПО (классы, пакеты, методы) должен быть ациклическим.

- Robert C. Martin

❖ Две сущности, которые не могут существовать друг без друга, не могут быть (пере-)использованы иначе как вместе. В таком случае, теряется смысл в их разделении на различные классы (пакеты, методы).

✓ Упрощает поддержку (maintainability)

Пример: циклическая зависимость



✓ Из-за связи из MyDialog в Application, пакет MyTasks переиспользовать вообще нельзя – он зависит от ВСЕЙ системы.

✓ Такую проблему называют «блюдо спагетти»

Принципы дизайна пакетов

- **CRP** – Common Reuse Principle

  Общий принцип переиспользования

- **CCP** - Common Closure Principle

  Принцип локализации изменений

- **SDP** - Stable Dependencies Principle

  Принцип стабильности зависимостей

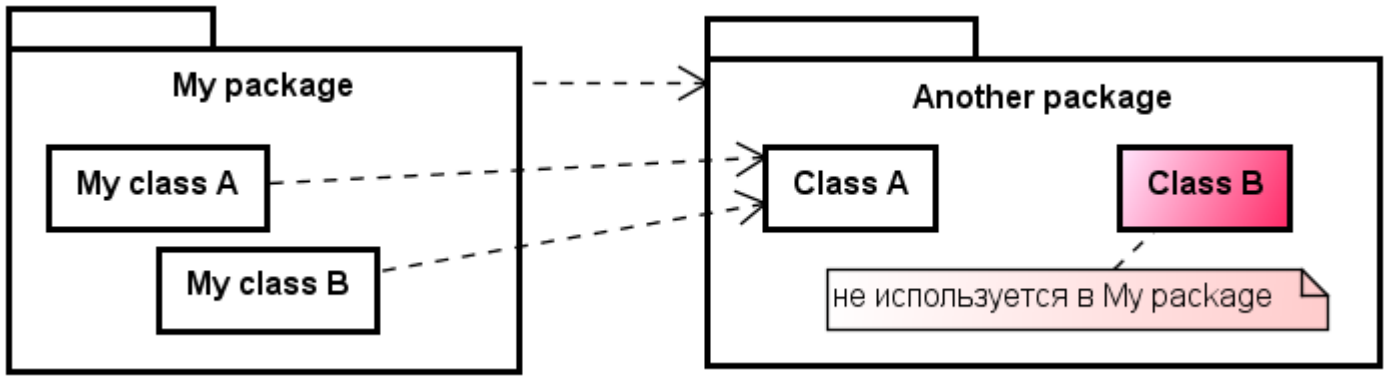
- **SAP** - Stable Abstractions Principle

  Принцип стабильности абстракций

CRP – Common Reuse Principle

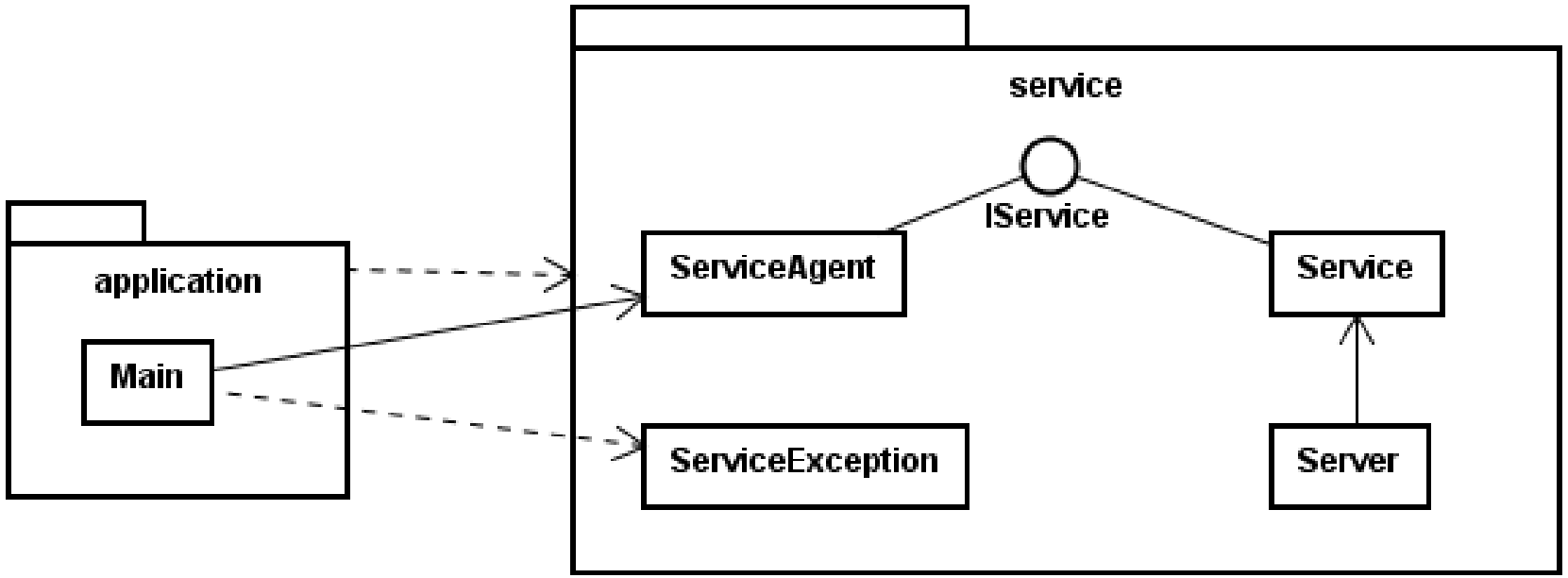
✓ Классы из пакета должны переиспользоваться вместе. Пользователи должны зависеть от пакета в целом, а не от его части.

- Robert C. Martin



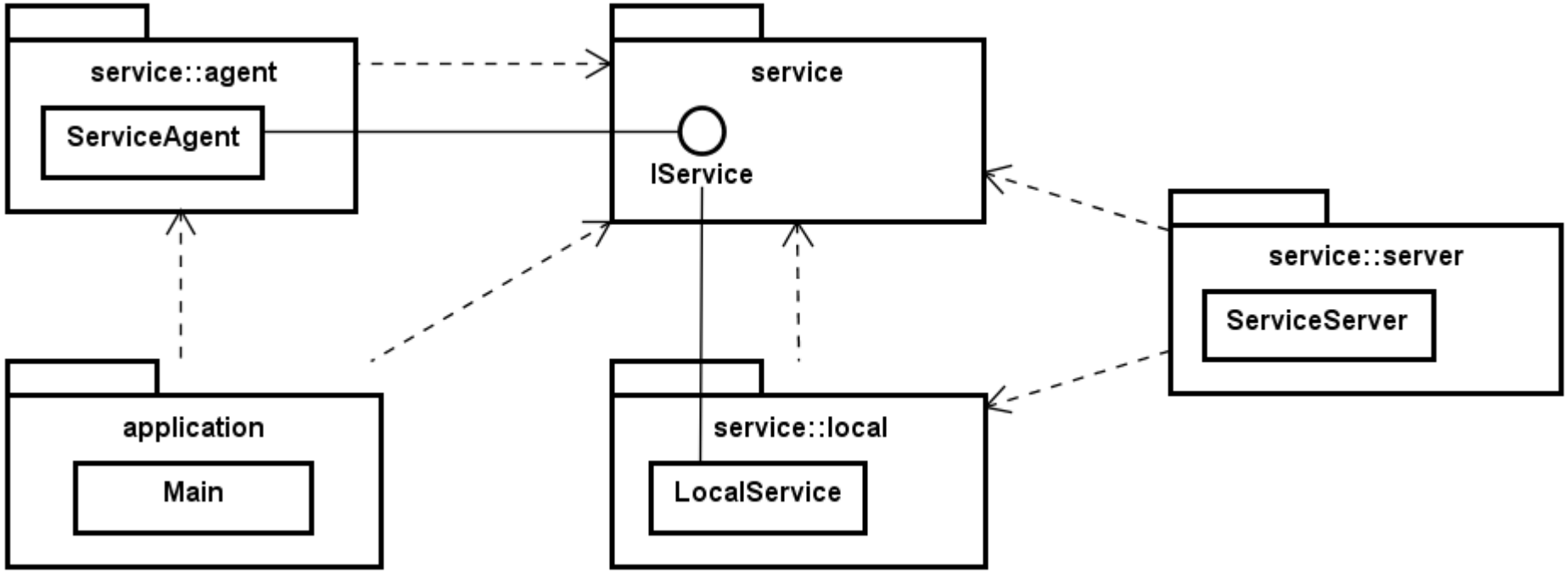
- ISP, адаптированный к пакетам
- Облегчает поддержку (maintenance)

Нарушение CRP: remote service



Проблема: Всякий раз когда выходит новая версия пакета service, клиенты ServiceAgent должны ожидать, что их код может перестать работать, даже если изменения реально не затронули ServiceAgent.

CRP совместимое решение



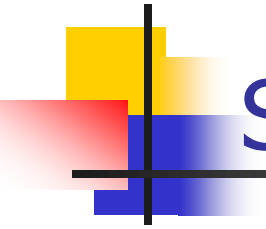
- Пакет application зависит только от того, что реально использует
- Изменения в пакетах local и server никак не затрагивают application

CCP: Локализация изменений

✓ Классы в пакете должны быть подвержены одному и тому же типу изменений – либо открыты для данного вида модификаций, либо закрыты от него.

- Robert C. Martin

- Локализует изменения и снижает число версий.
- Есть некоторое сходство с OCP для классов



SDP – Стабильность зависимостей

✓ Пакет должен зависеть только от пакетов, более стабильных, чем он сам.

✓ нестабильность пакета – мера вероятности появления в нем изменений вследствие изменений других пакетов.

- Robert C. Martin

Нестабильность пакета

Нестабильность пакета:

I = Ce / (Ca + Ce)

Где:

Ca* = число входящих связей (число классов вовне пакета, **которые** зависят от классов внутри пакета). = насколько пакет «важен» для других пакетов

Ce* = число исходящих связей (число классов внутри пакета, **которые** зависят от классов вовне). = насколько пакет «зависим» от других пакетов

I = 0 – абсолютно стабильный пакет
I = 1 – очень нестабильный пакет

* Здесь даны определения Ca, Ce согласно R. Martin «Agile Principles, Patterns, and Practices in C#», 2006г. В более поздних публикациях, например «Clean Architecture» за 2018г., R.Martin использует вместо Ca, Ce обозначения Fan-In и Fan-Out соответственно.

SAP – Стабильность абстракций

✓ Абстрактность пакета должна быть пропорциональна его стабильности.

- Robert C. Martin

- Если все пакеты абсолютно стабильны, то система просто немодифицируема, т.е бесполезна.
- => некоторые пакеты просто обязаны быть нестабильными.
- Вопрос: какие это пакеты?

- Упрощает поддержку (maintainability)

Абстрактность пакета

Абстрактность пакета

A = Na / N

где

Na = число абстрактных классов (интерфейсов)

N = полное число классов в пакете

Генеральная линия

- ✓ Строим график I(A) - нестабильность(абстрактность)
- ✓ Пакеты вдоль линии (0,1) to (1,0) имеют хороший баланс
- ✓ Отклонение от генеральной линии:

$$D = | A + I - 1 |$$

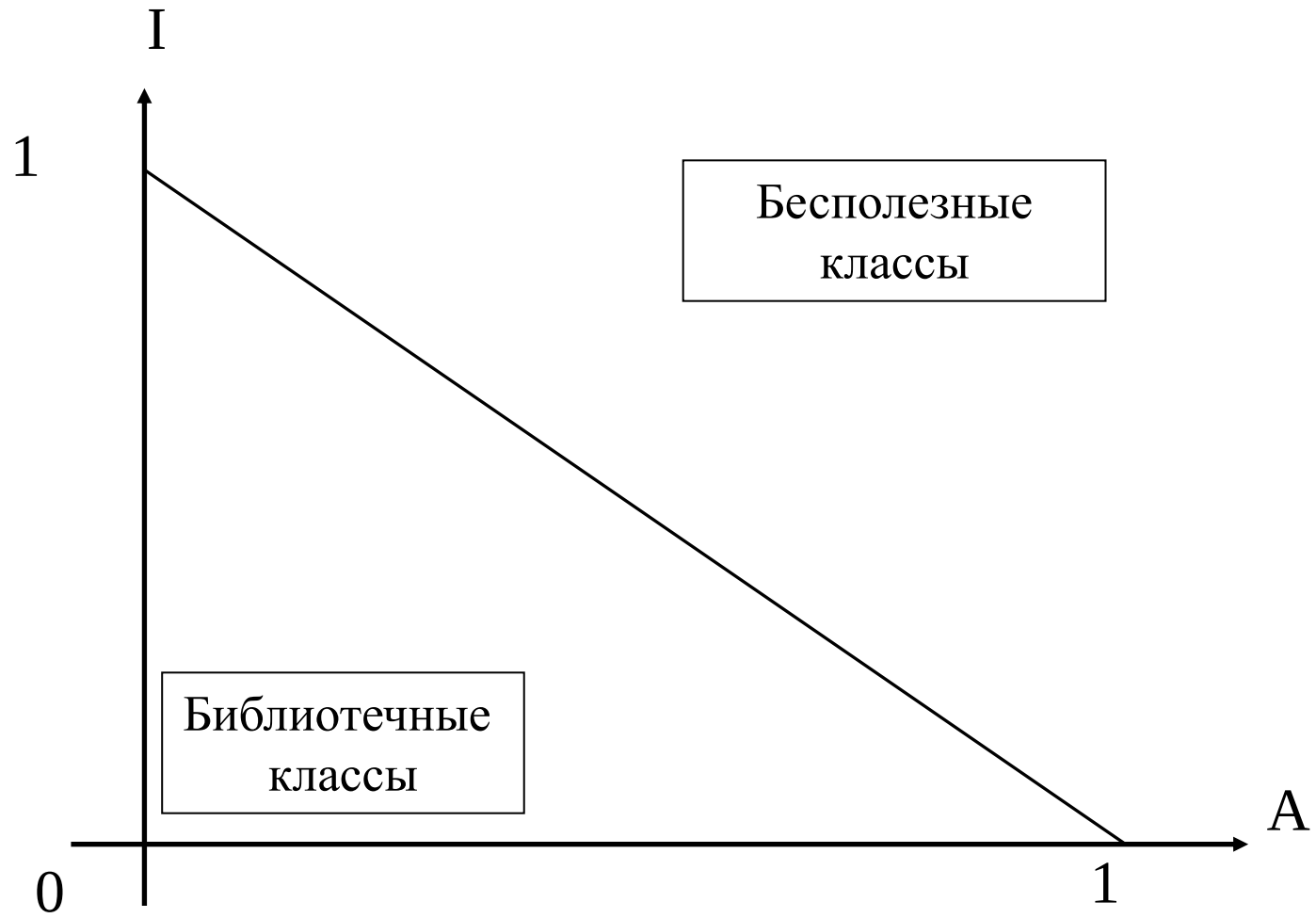
указывает на потенциальные проблемы в дизайне пакета.

-Robert C. Martin

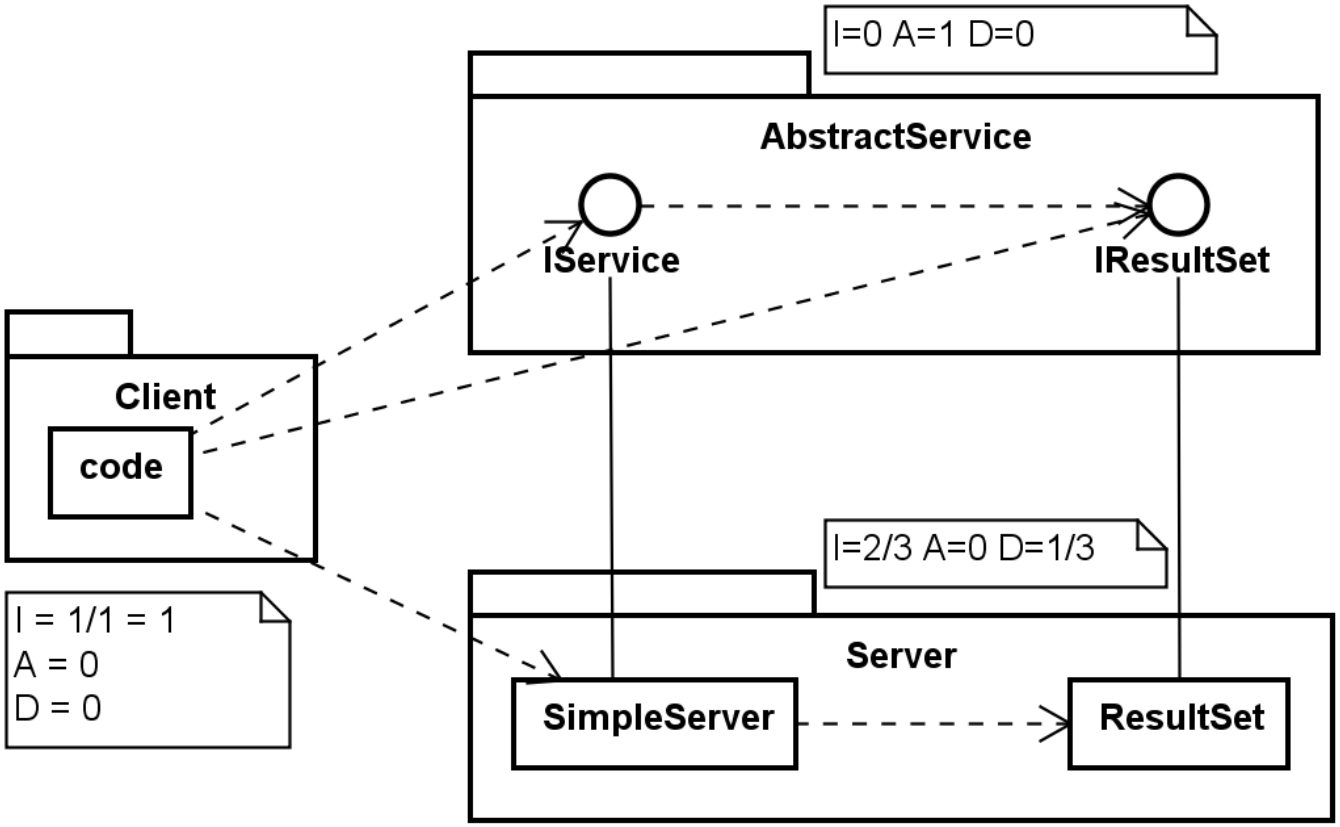
Java: Рассчет **I**, **A** и **D** реализован в пакете **JDepend** (open source)

.NET - **NDepend** (коммерческое ПО)

Main sequence

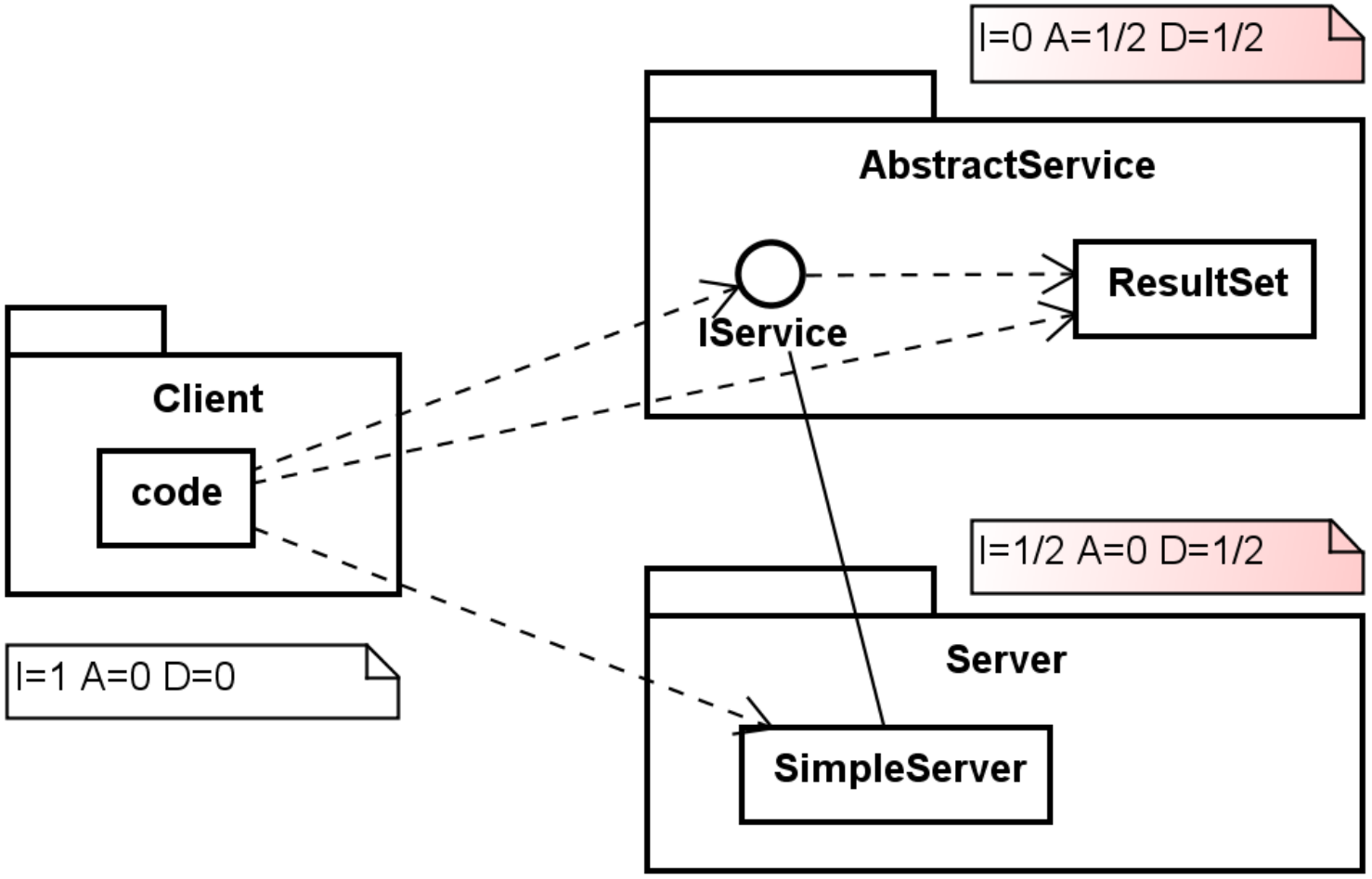


Пример: client-server



- Теперь внесем заведомо плохое изменение, сделав IResultSet не interface. (Вопрос: какой принцип мы нарушим?) Если наше правило работает – метрики должны измениться в худшую сторону

Пример: client-server



- Увеличилась дистанция у обоих пакетов - Server и AbstractService