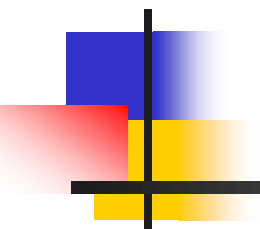


Объектно-ориентированный Анализ и Дизайн



Часть 2

ОО Анализ: Аналитическая модель
Переход от анализа к дизайну
Принципы ОО дизайна



5. OO Анализ

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



Цели фаз 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)
- Design model



Аналитическая модель

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

➤ Элементы analysis model:

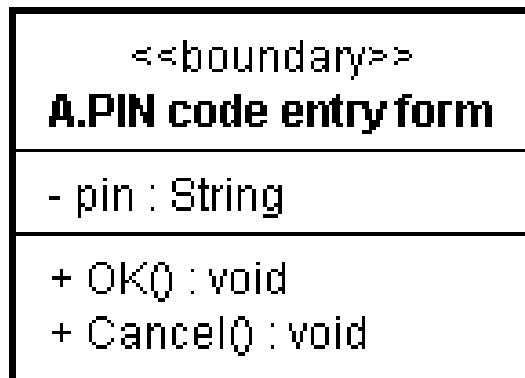
- **Use-case realization** – реализация use-case, набор activity, state, collaboration и class диаграмм
- **Boundary class** – класс, разграничивающий actor-ов и систему
- **Control** – класс, управляющий другими классами
- **Entity** – класс, моделирующий информацию, используемую в системе

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>>



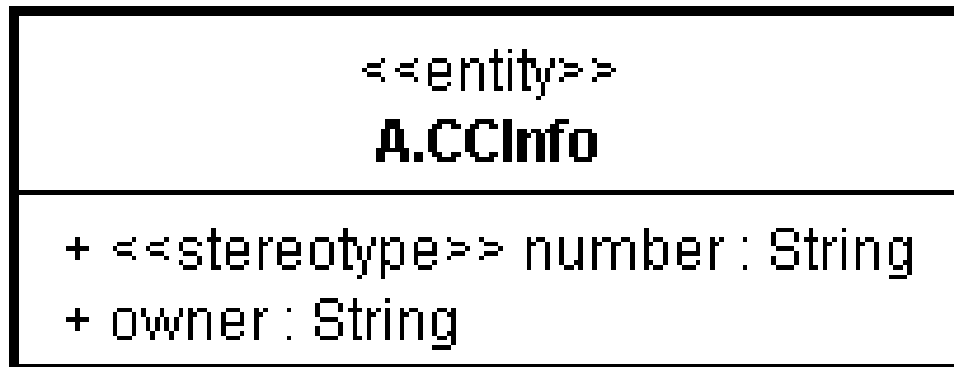
Представление control посредством стереотипа и пиктограммы

Entity

-Класс, моделирующий информацию, используемую в системе

используемую в

UML: class со стереотипом <<entity>>



Представление entity посредством стереотипа и пиктограммы



Диаграммы взаимодействия

- Последовательностей - Sequence diagrams
 - Коопераций - Collaboration diagrams
- Отражают динамические аспекты поведения объектов
 - Семантически эквивалентны
 - Содержат:
 - Объекты
 - Связи
 - Сообщения
 - Поток данных

Авторизация в банкомате

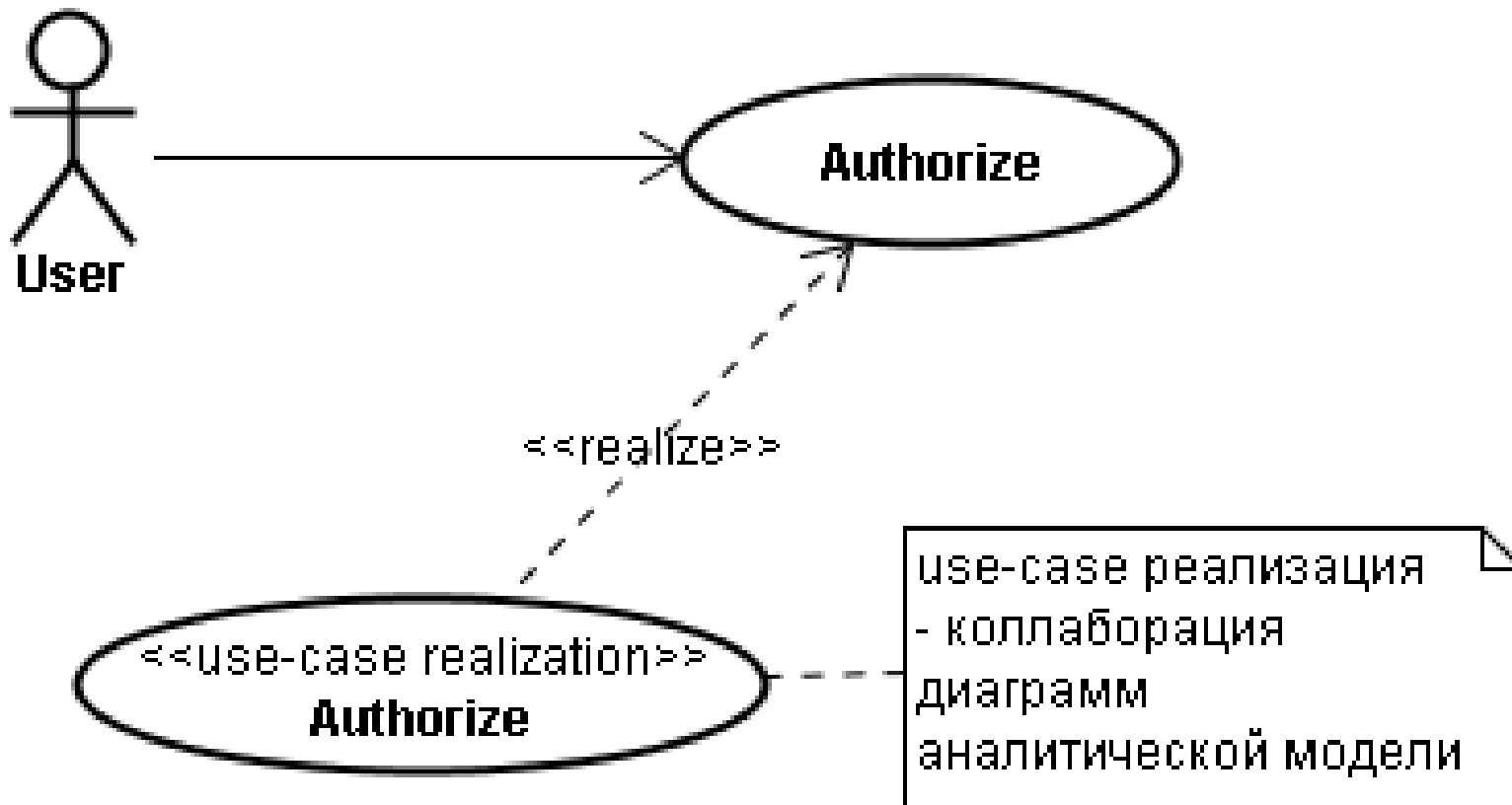


Диаграмма аналитических классов

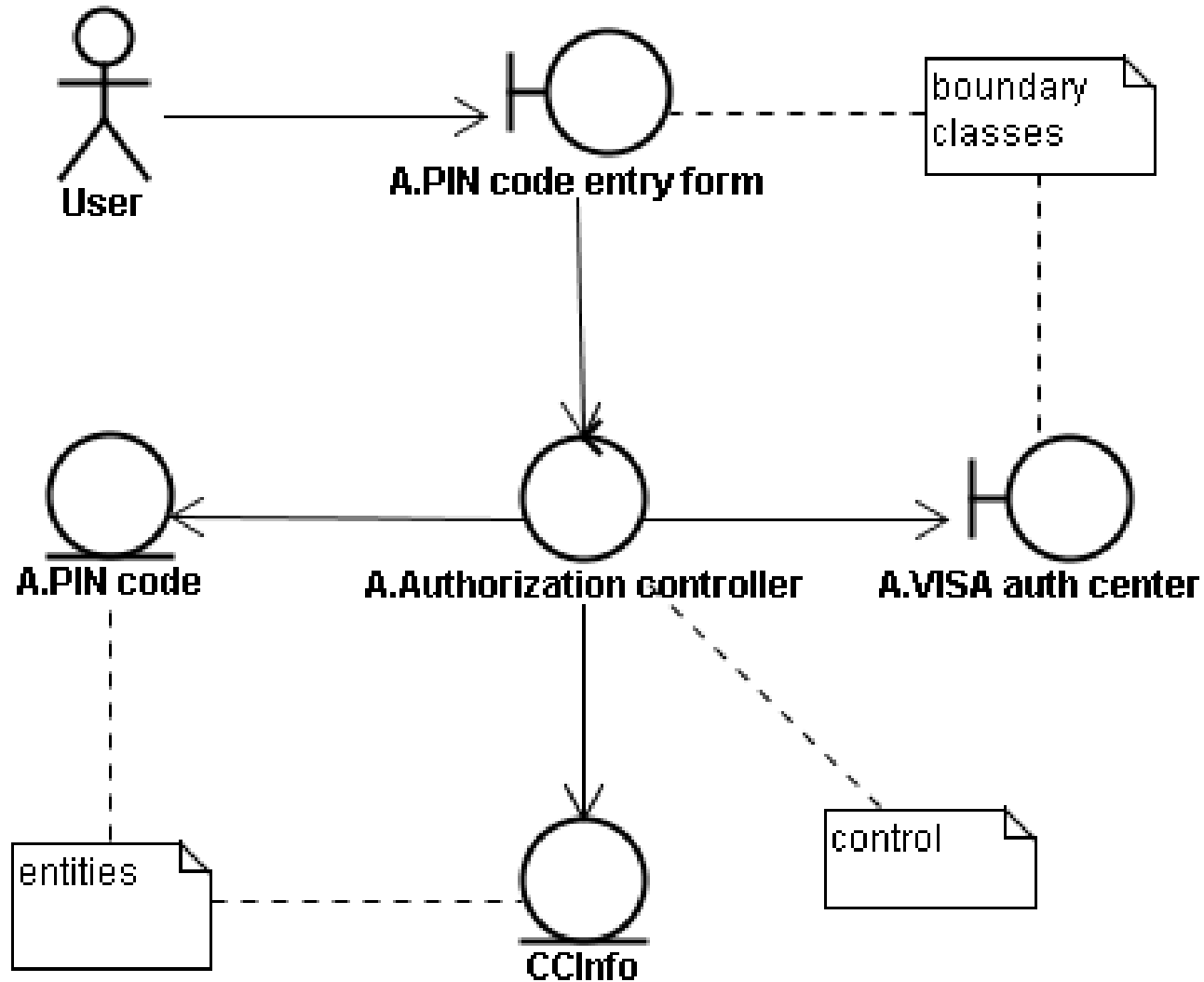


Диаграмма последовательностей

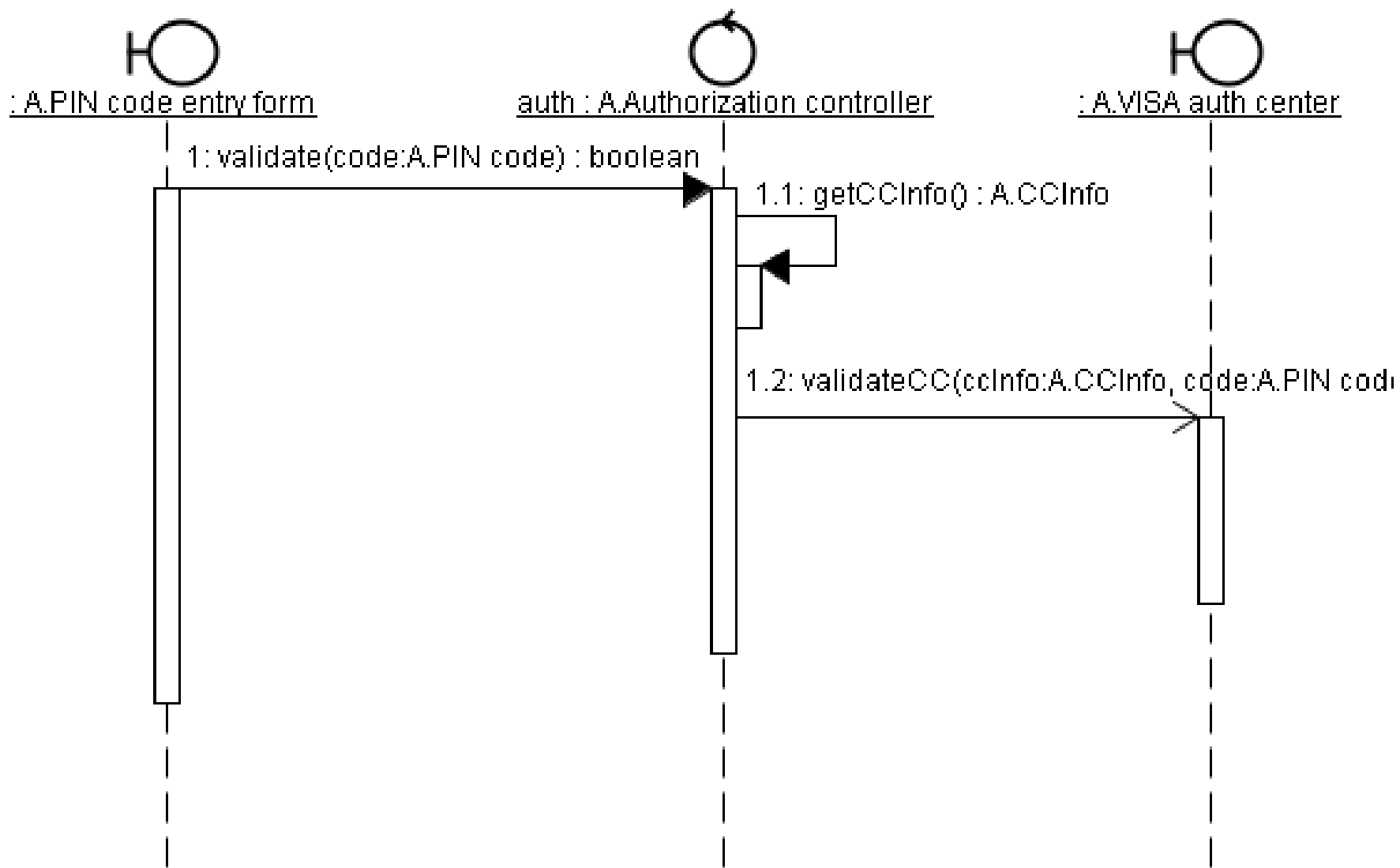
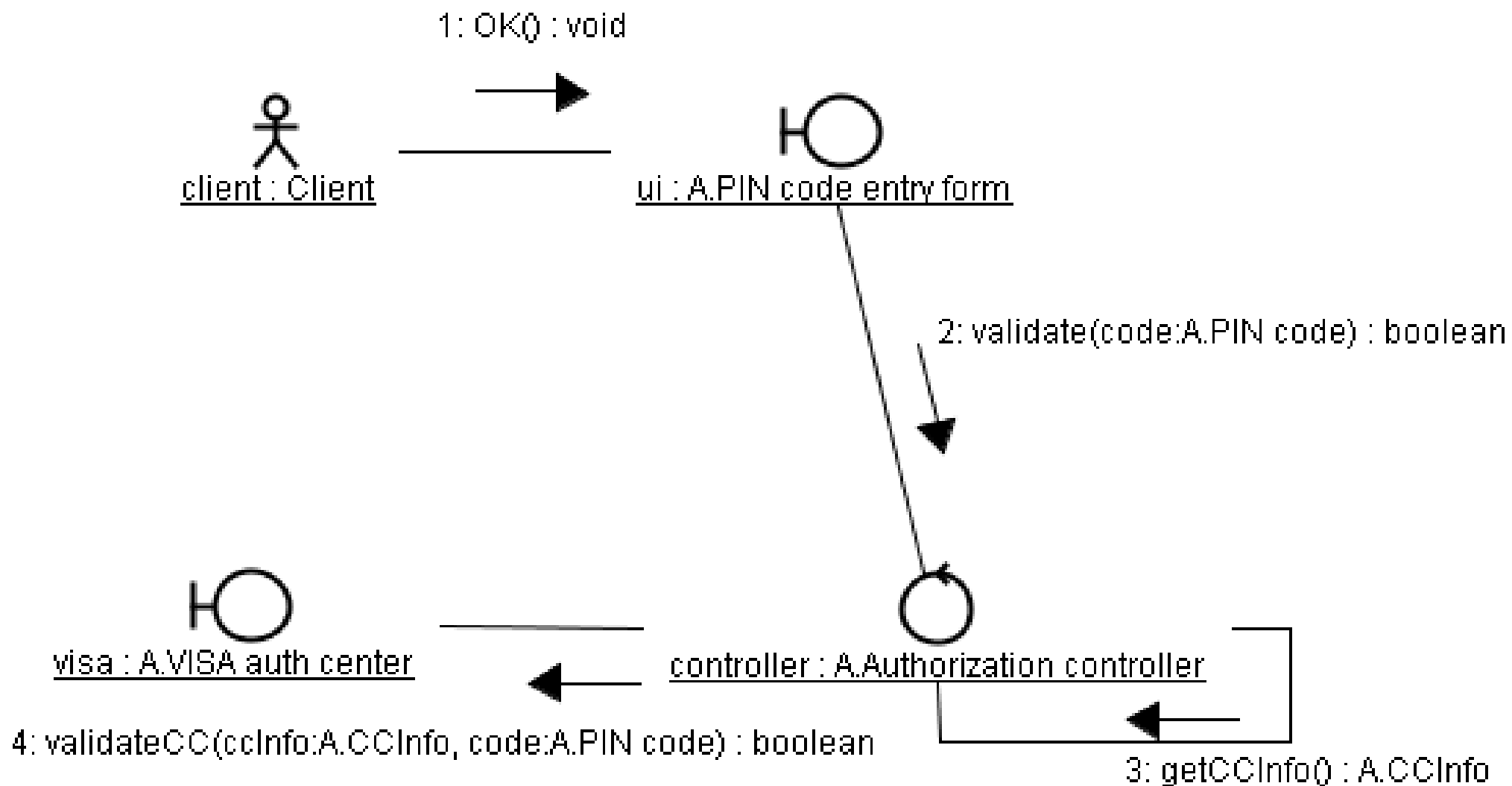


Диаграмма коопераций





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. OO-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)



Design model

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

■ Элементы design model:

- **Layer** - слой (UI, UI logic, Business logic, Data, system)
- **Subsystem** - подсистема
- **Package** - пакет
- **Class** – класс
- **Use-case realization** – коллекция диаграмм

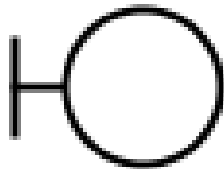


Переход от анализа к дизайну

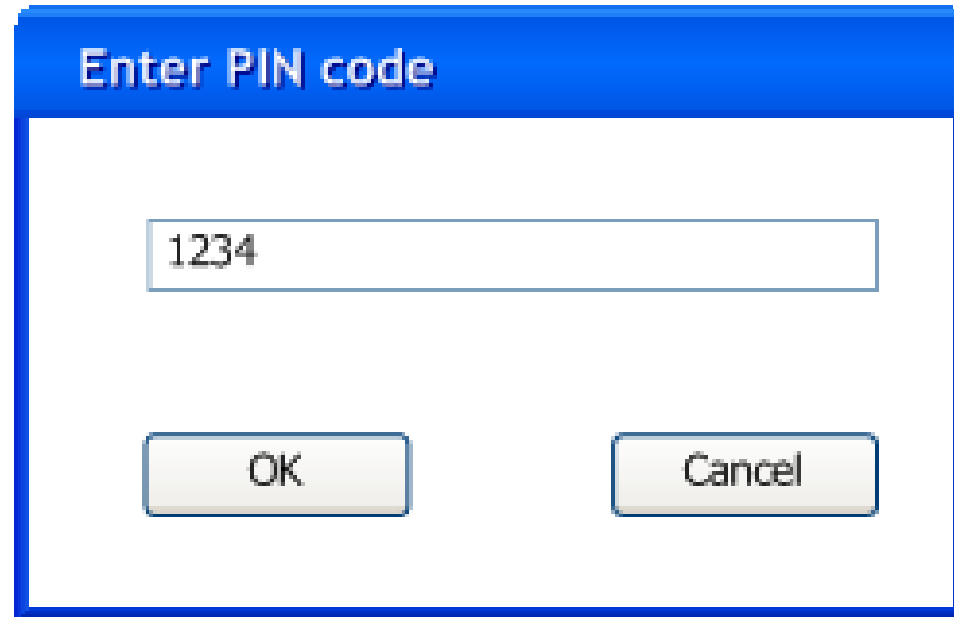
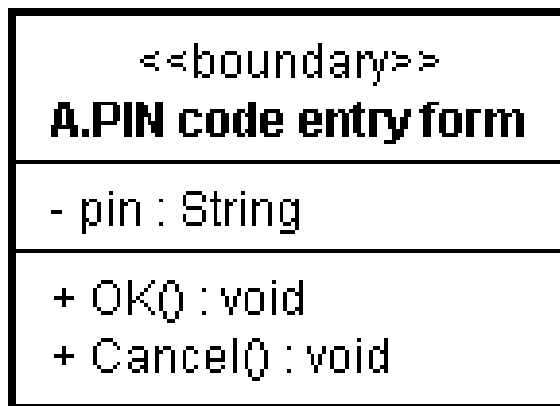
- Аналитический класс при переходе к дизайну трансформируется в один или несколько классов дизайна, которые реализуются на каком-либо конкретном языке программирования

Трансформация boundary

- Классы пользовательского интерфейса



A.PIN code entry form



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

Трансформация boundary

- Интерфейс (для) внешней системы

Аналитическая модель

Дизайн модель


A.VISA auth center


VisaAuthority

**<<boundary>>
A.VISA auth center**

+ validateCC(ccInfo : CCInfo, code : A.PIN code) : boolean

**<<interface>>
VisaAuthority**

+ validateCC(ccInfo : CCInfo, pinCode : String) : boolean

Трансформация entity

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

Аналитическая модель

Дизайн модель



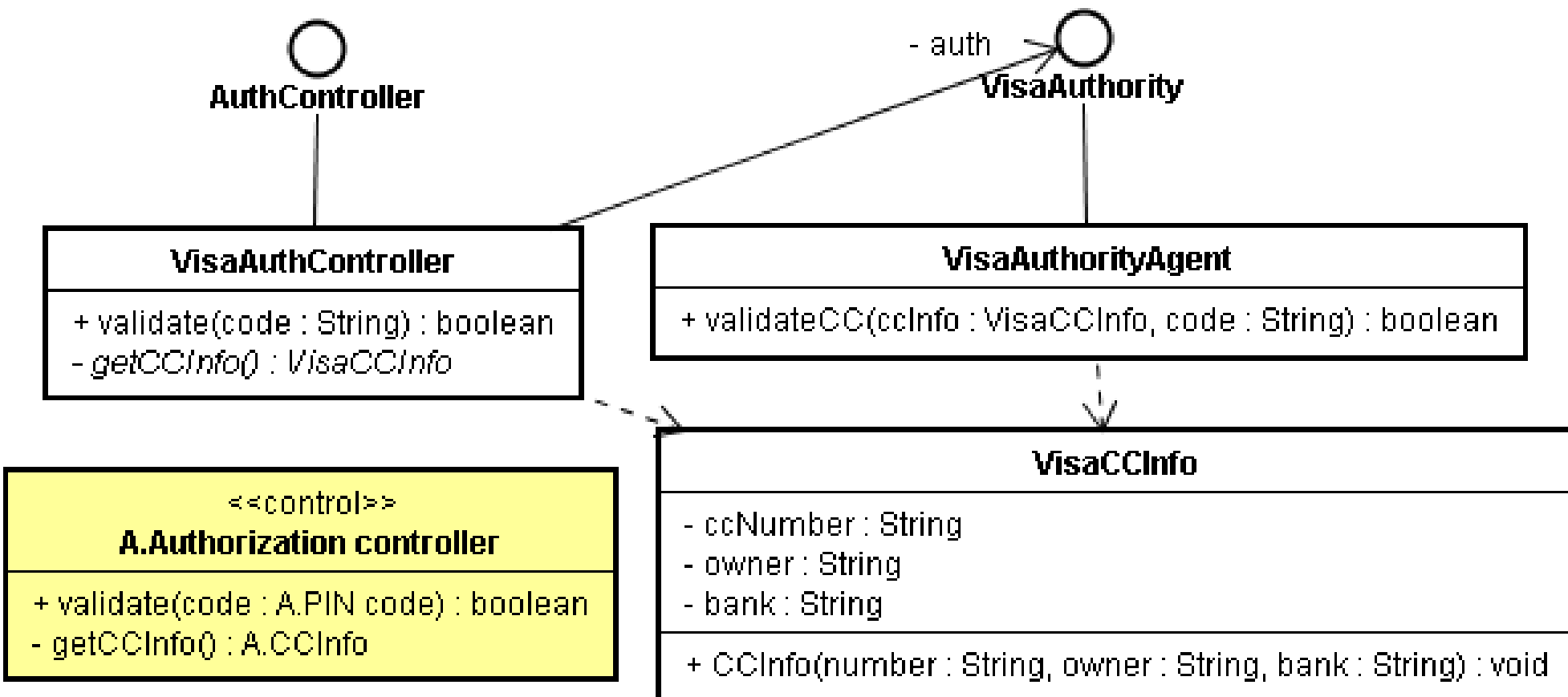
VisaCCInfo

- ccNumber : long
- owner : String
- bank : String

+ CCInfo(number : int, owner : String, bank : String) : void

Трансформация control

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





7. Принципы ОО дизайна

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

Б.Страуструп



Принципы дизайна классов

- **ORR** - Принцип целостности абстракции
(One Responsibility Rule)
- **LSP** - Принцип подстановки
(Liskov Substitution Principle)
- **LoD** - Закон Деметры
(Law of Demeter)
- **OCP** - Принцип закрытости абстракции
(Open-Closed Principle)
- **ISP** - Разделение интерфейсов
(Interface Segregation Principle)



ORR – Правило целостности абстракции

Кот с улыбкой - и то редкость, но уж улыбка
без Кота - это я прямо не знаю что такое!

Алиса в стране чудес

Класс должен обладать единственной ответственностью,

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

- R. Martin

A class has a single responsibility:

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

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.

- R.Martin, 1996

✓ 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, 1988

Нарушение LSP: Фигуры

Rectangle

- ◆ Rectangle(w : int, h : int)
- ◆ setHeight(h : int) : void
- ◆ getHeight() : int
- ◆ setWidth(w : int) : void
- ◆ getWidth() : int



Square

- ◆ Square(s : int)

```
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
```

Пробуем исправить дело:

Rectangle

- ◆ Rectangle(w : int, h : int)
- ◆ setHeight(h : int) : void
- ◆ getHeight() : int
- ◆ setWidth(w : int) : void
- ◆ getWidth() : int



Square

- ◆ Square(s : int)
- ◆ setSize(s : int) : void
- ◆ setHeight(h : int) : void
- ◆ setWidth(w : int) : void

```
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); }
}
```

Не помогает:

```
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 квадрат просто **не является** прямоугольником, потому что:
- Класс – абстракция данных *и поведения*
- При этом *поведение* квадрата существенно отличается от поведения прямоугольника
- Наследование – отношение «частное-общее», но общность надо искать в поведении, а не в структуре



LoD – Закон Дементры

✓Метод должен обладать ограниченным знанием об объектной модели приложения.

- D. Rumbaugh

✓Оригинальная формулировка:

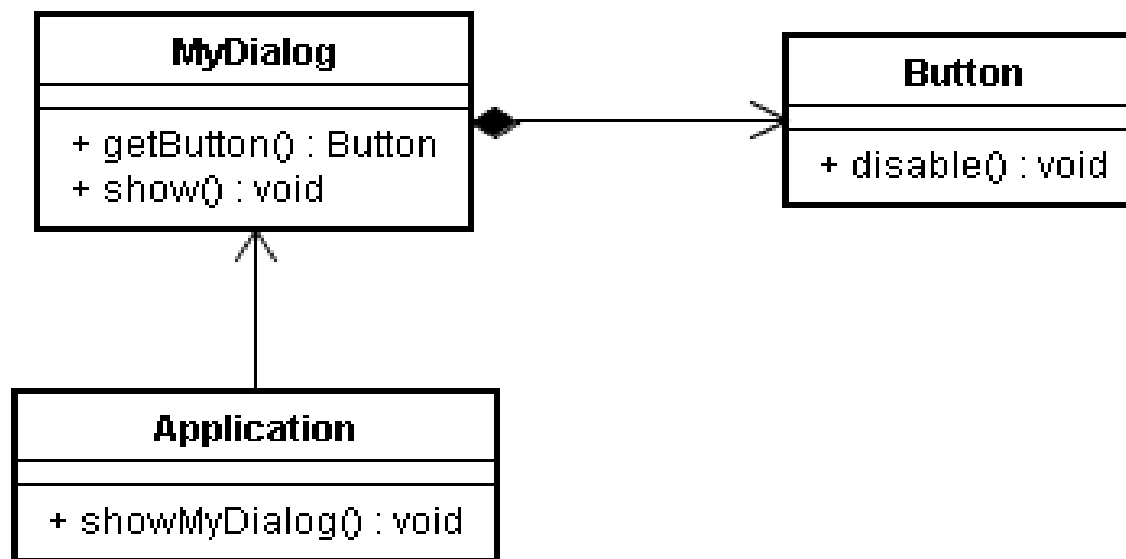
Only talk to your immediate friends. Never talk to strangers.

- Ian Holland, 1987

Друзья метода **f** :

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

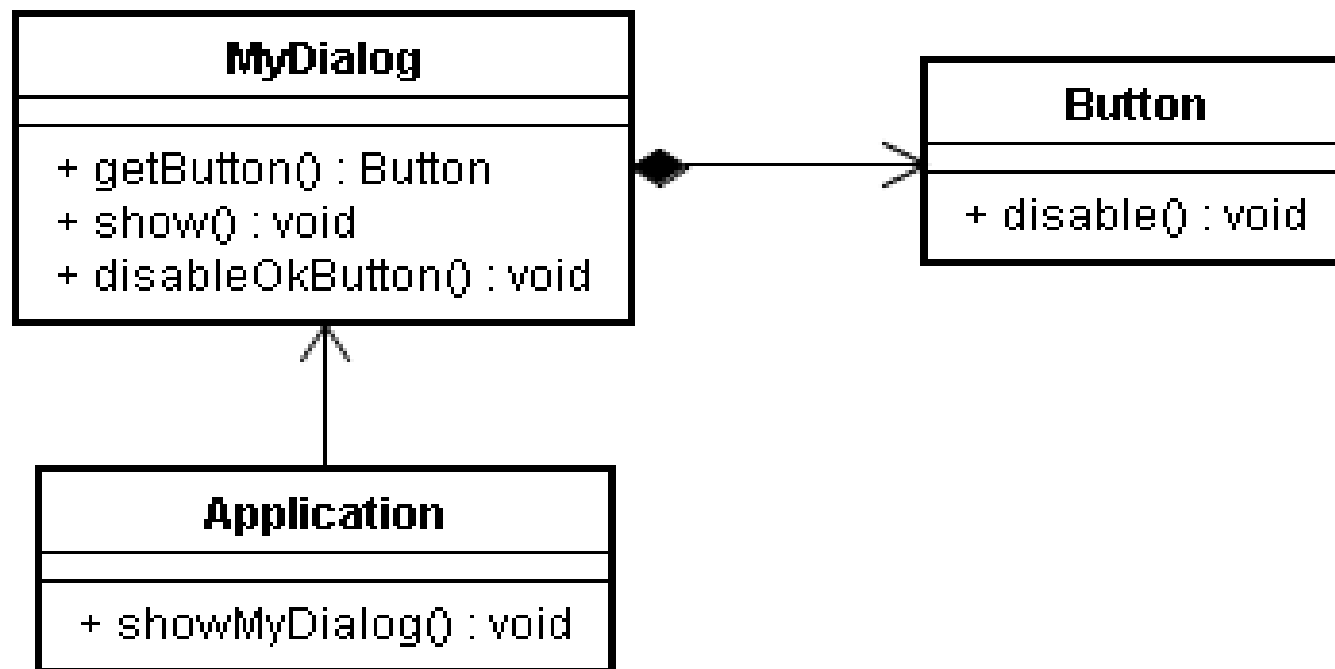
Нарушение LoD



Проблема: `public void showMyDialog() {`
`.... // создание MyDialog`
`myDialog.getButton().disable();`
`}`

1. связь `Application -> Button` мы не планировали.
2. Замена класса `Button` на другой интерфейсный класс ведет к перекомпиляции и перетестированию `Application`

LoD-совместимый дизайн



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



ОСР – Принцип закрытости абстракции

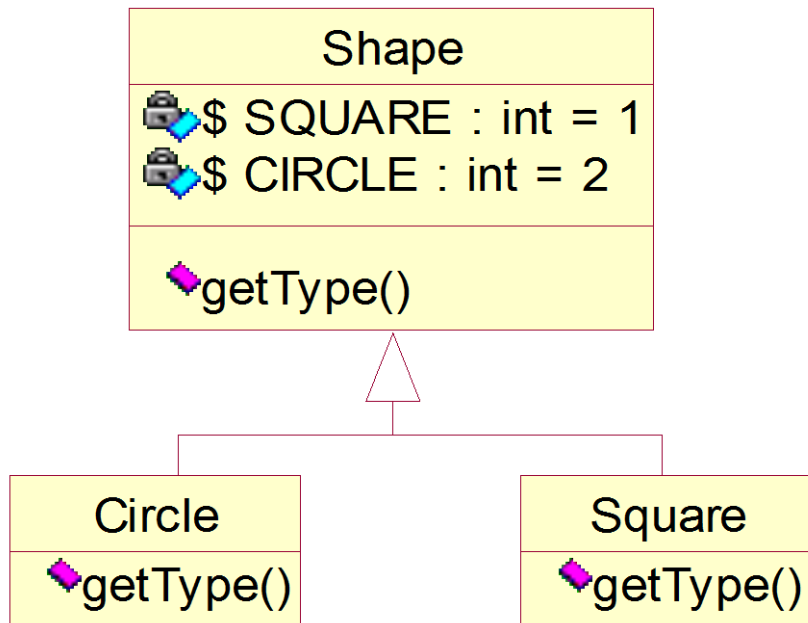
✓ Компоненты программной системы (классы, модули, методы) должны быть открыты для расширения, но закрыты от модификации.

- В. Meyer, 1988

✓ 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 ОСР: Фигуры

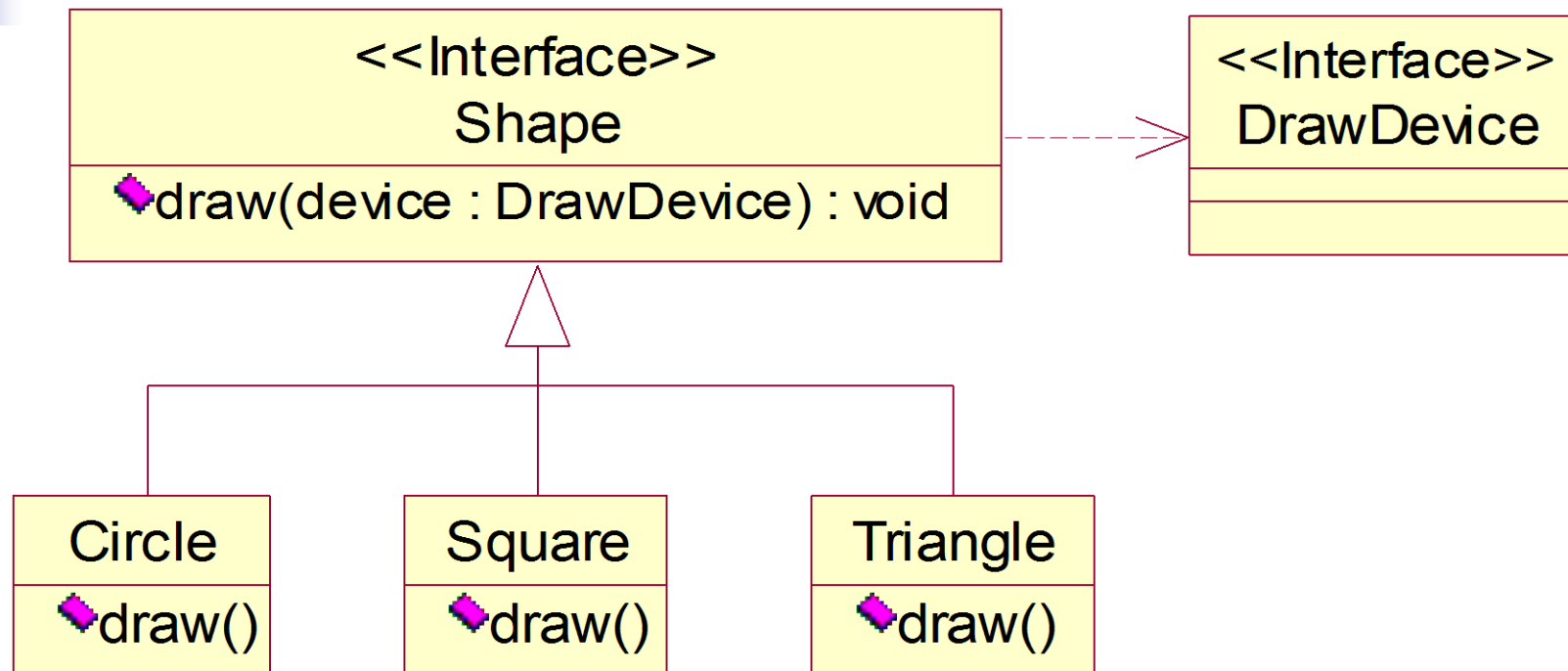


```
void drawShapes( Shape[] shapes )
{
    for( int i = 0; i < shapes.length; ++i )
    {
        if( shape[i].getType == Shape.SQUARE )
        {
            drawSquare( (Square)shape[i] );
        }
        else drawCircle( (Circle)shape[i] );
    }
}
```

Проблема:

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

ОСР-совместимое решение



```
void drawShapes( Shape[] shapes ) {
    for( int i=0; i < shapes.length; ++i )
    {
        shape[i].draw( device );
    }
}
```



ISP – Разделение интерфейсов

✓ Клинтов нельзя заставлять платить за сервисы, которых они не используют.

- R.Martin, 1996

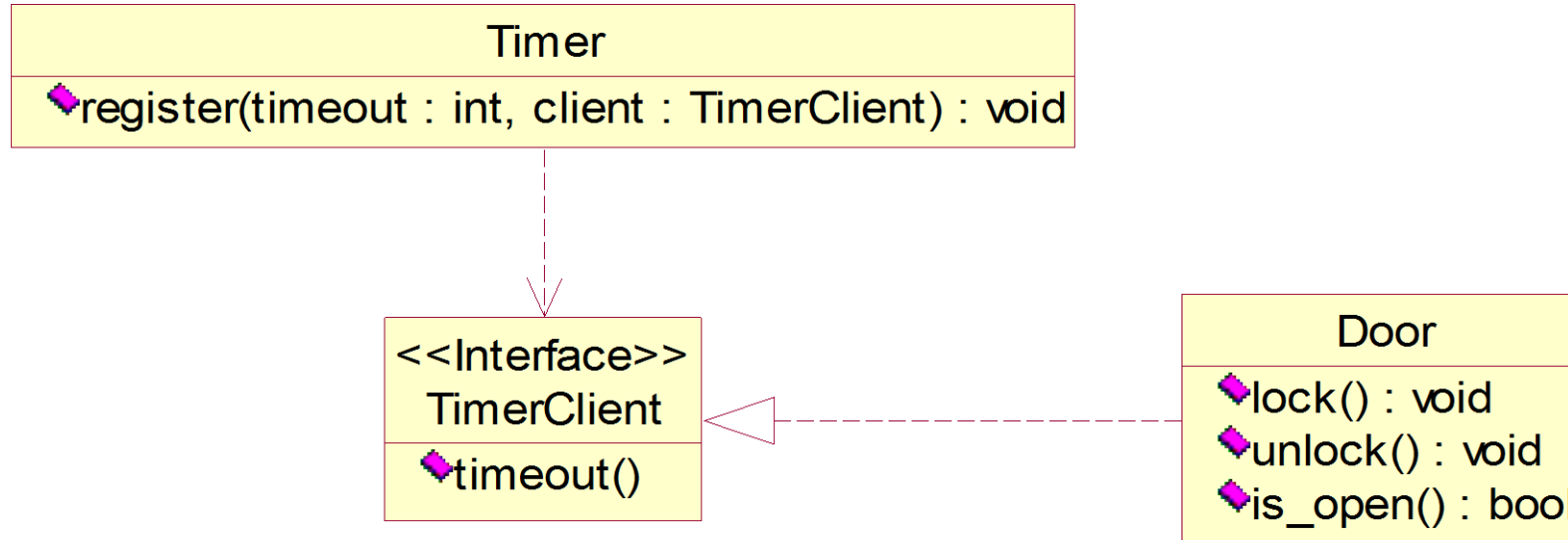
Hints:

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

Цена нарушения:

- ❖ Потеря гибкости

Нарушение ISP: Security Door

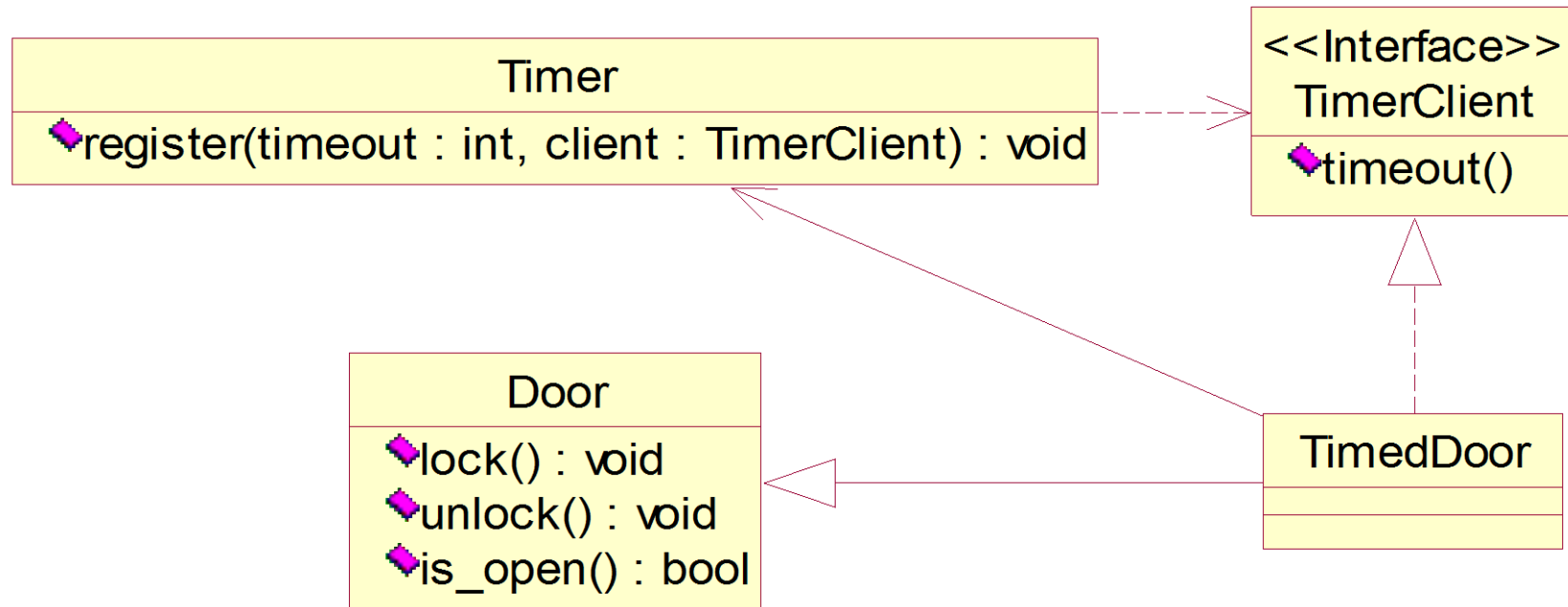


Дверь издает звук если открыта слишком долго.

Проблемы:

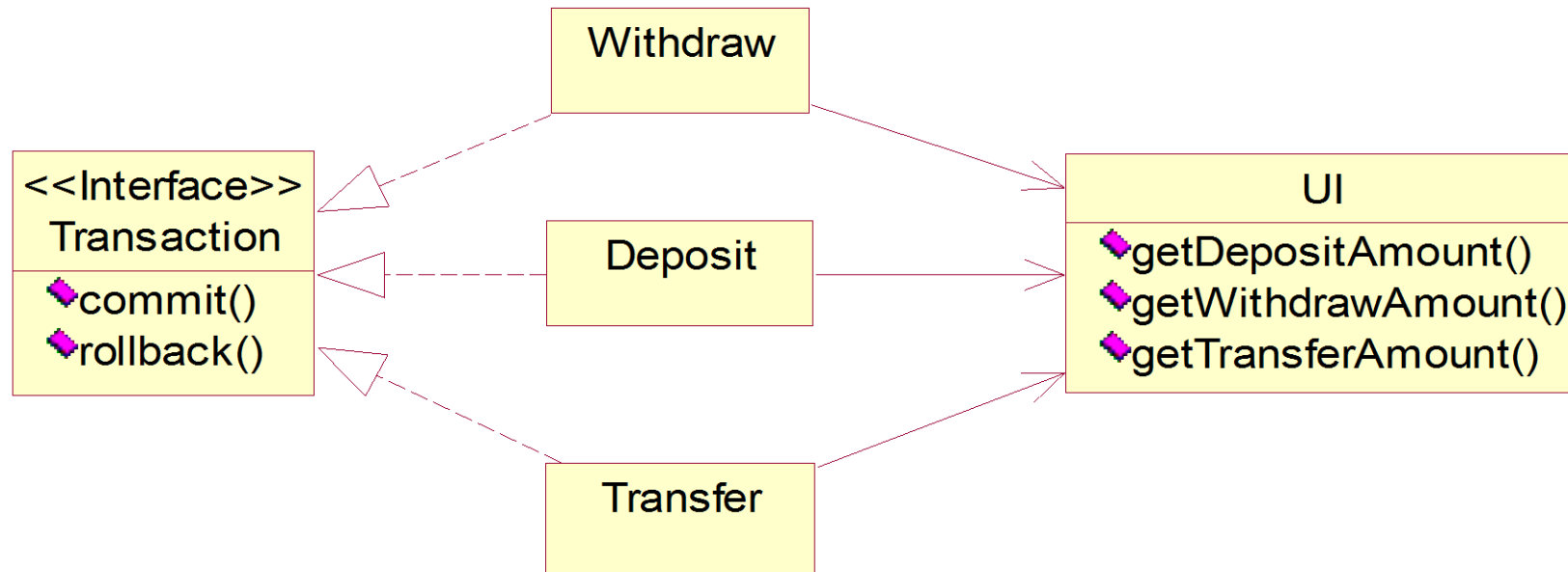
- Метод `timeout()` обязан быть `public`.
- Некоторые клиенты класса **Door** не используют и не должны использовать `timeout()`.
- *Может приводить к ошибкам.*

ISP-совместимая Security Door



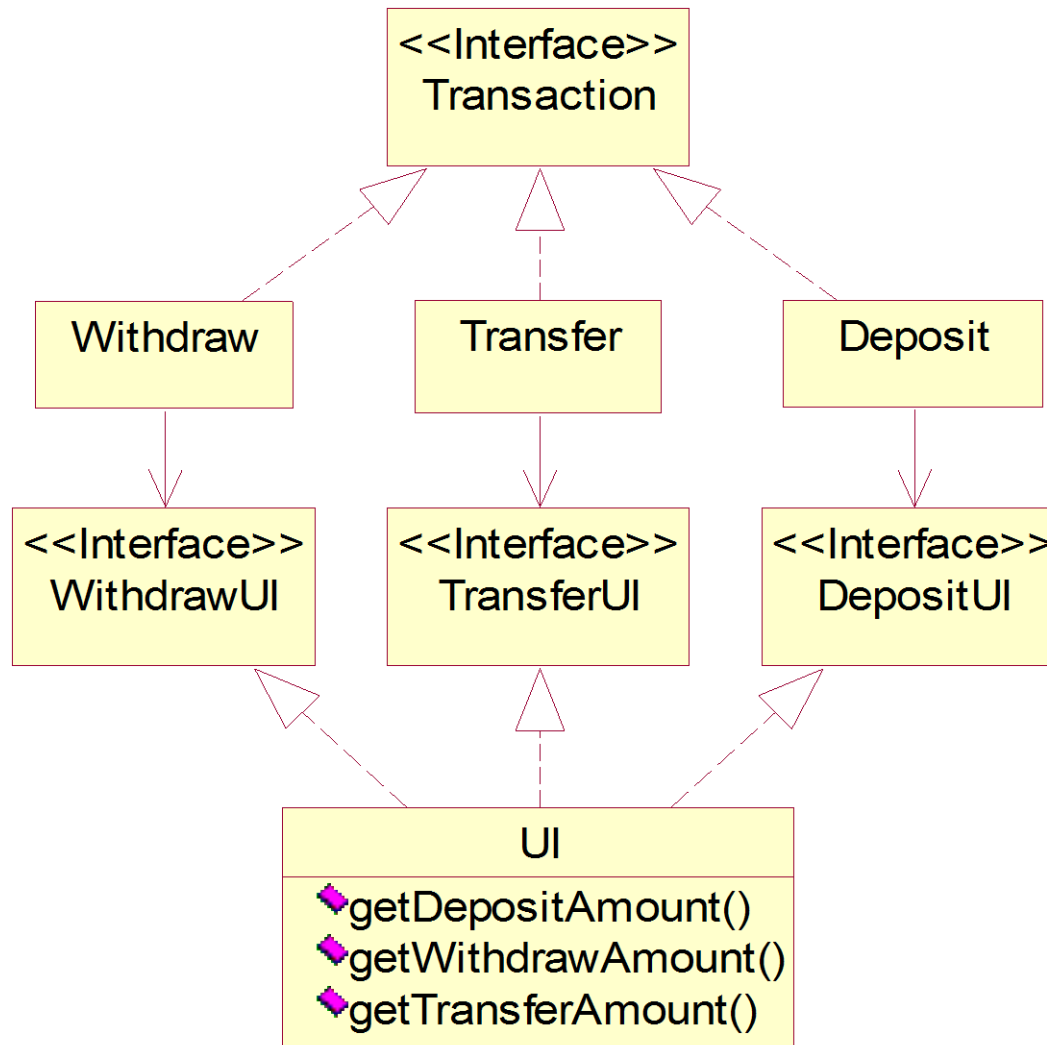
- ✓ Клиенты Door могут использовать TimedDoor
- ✓ Клиенты Door не будут зависеть от изменений в Timer, TimerClient или TimedDoor

Нарушение ISP: Банкомат



- ❖ Добавление новой транзакции требует перекомпилировать (=перетестировать) все остальные
- ❖ Если любая транзакция требует изменений в UI, остальные также придется перекомпилировать

ISP-совместимое решение





Дизайн связей

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



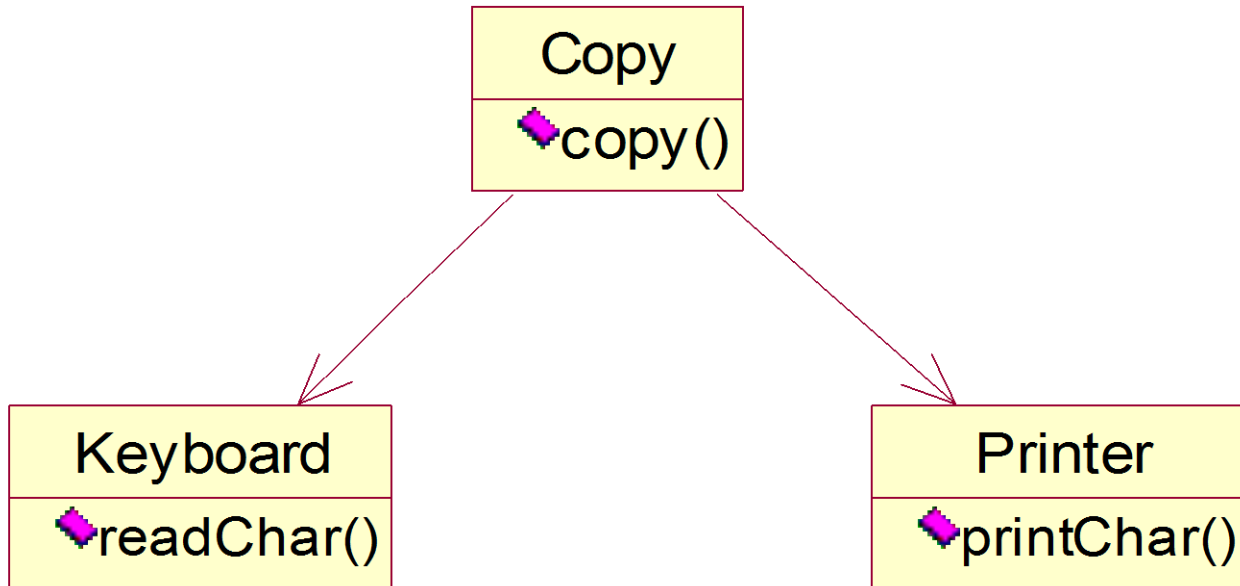
DIP – Инверсия зависимостей

✓ 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.

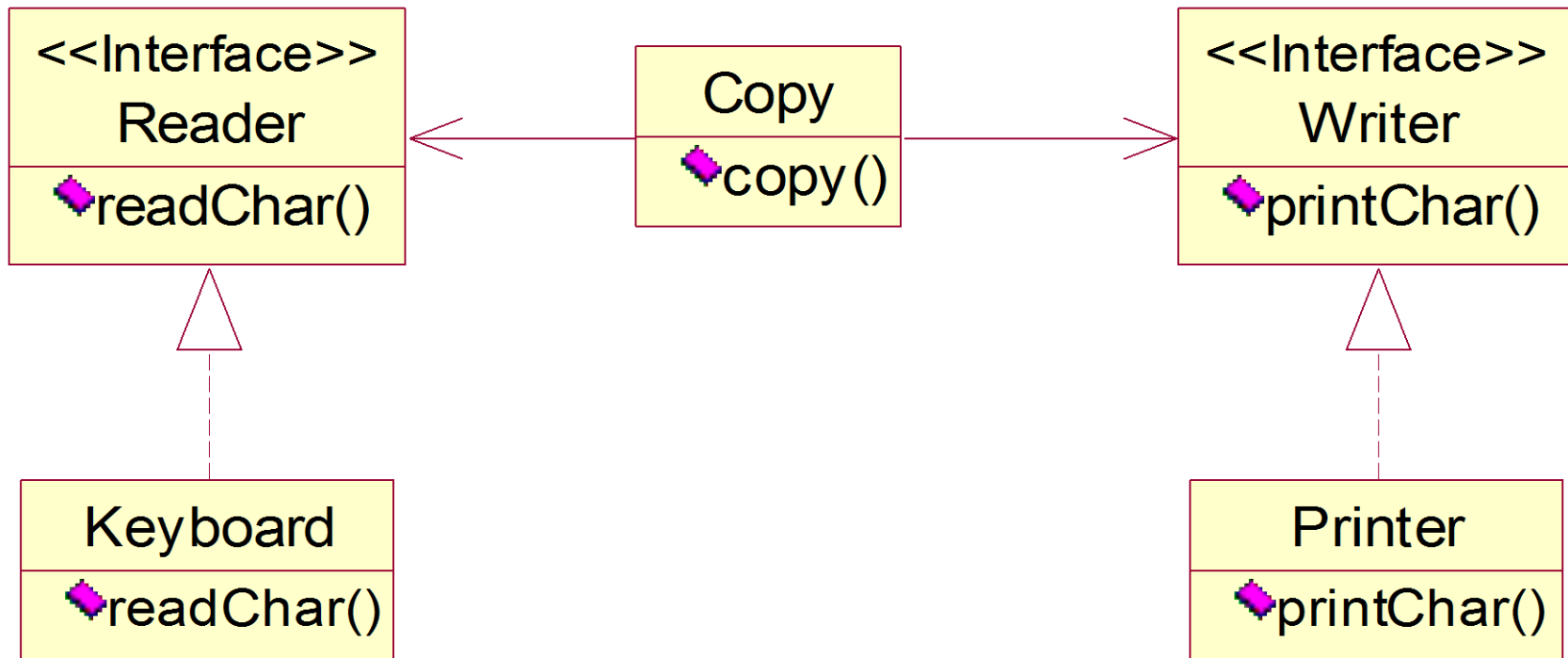
- R.Martin, 1996

Нарушение DIP: Copier



Что делать, если требуется добавить еще одно устройство печати?

DIP-совместимое решение



✓Теперь легко добавляем новые устройства



ADP – Ациклические связи

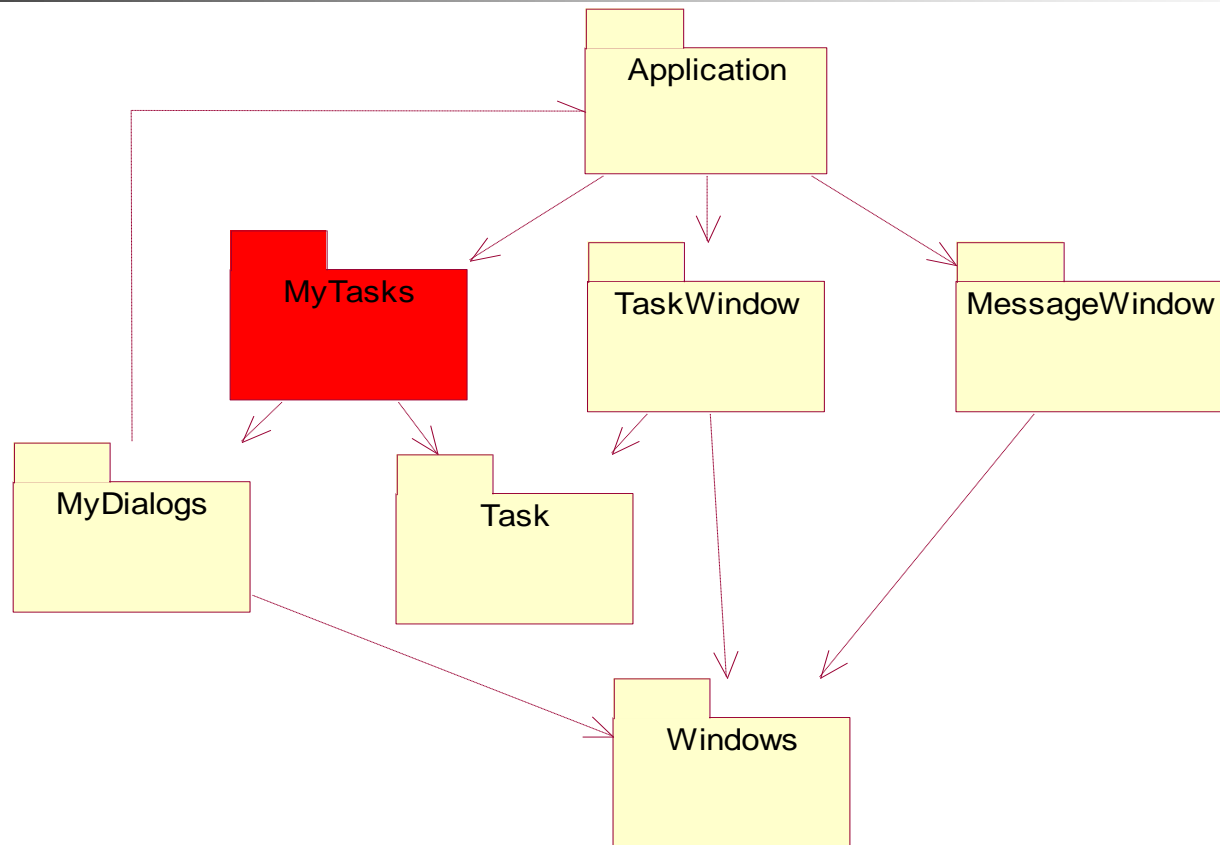
✓ Граф зависимостей между компонентами ПО (классы, пакеты, методы) должен быть ациклическим.

- R.Martin, 1996

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

□ Упрощает поддержку (maintainability)

Пример: циклическая зависимость



- ✓ Из-за связи из MyDialogs в Application, пакет MyTasks переиспользовать вообще нельзя – он зависит от ВСЕЙ системы.
- ✓ Был ОО-дизайн – получилось «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

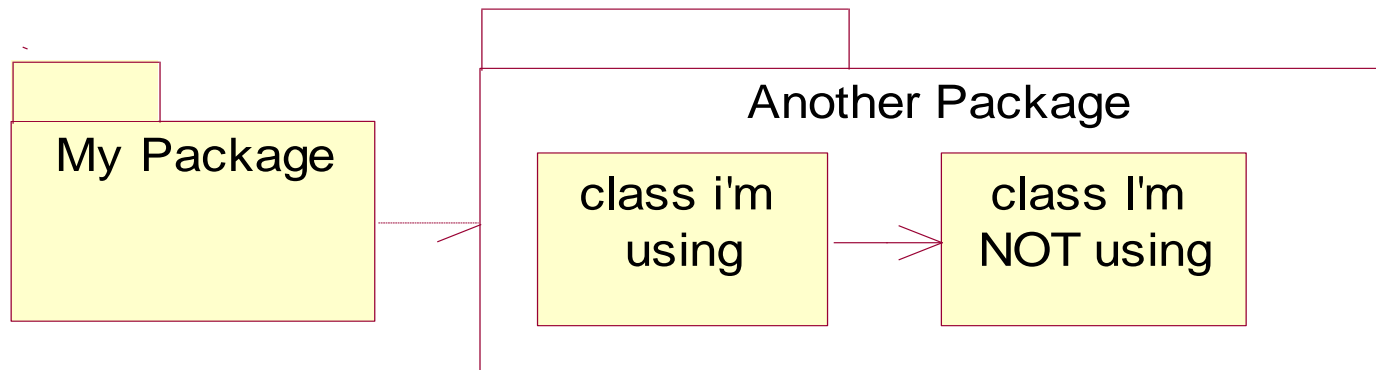
CRP – Common Reuse Principle

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

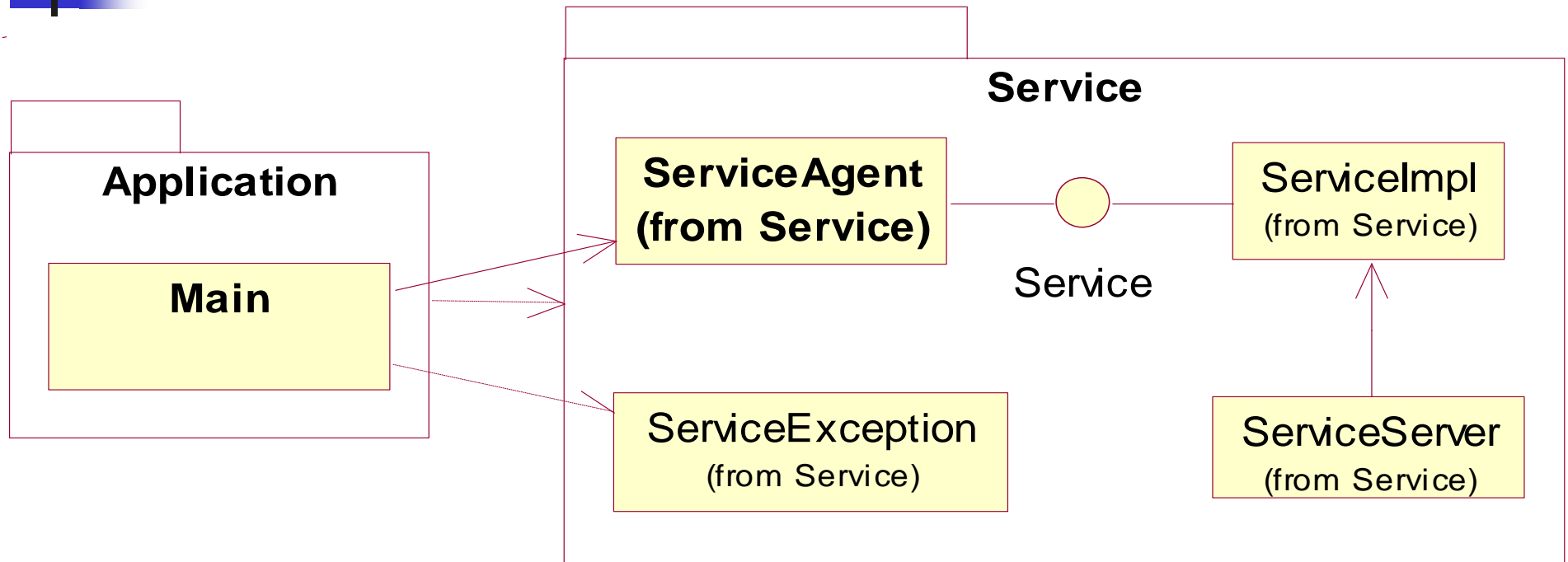
– R.Martin, 1996

❖ ISP, адаптированный к пакетам

❑ Облегчает поддержку (maintenance)

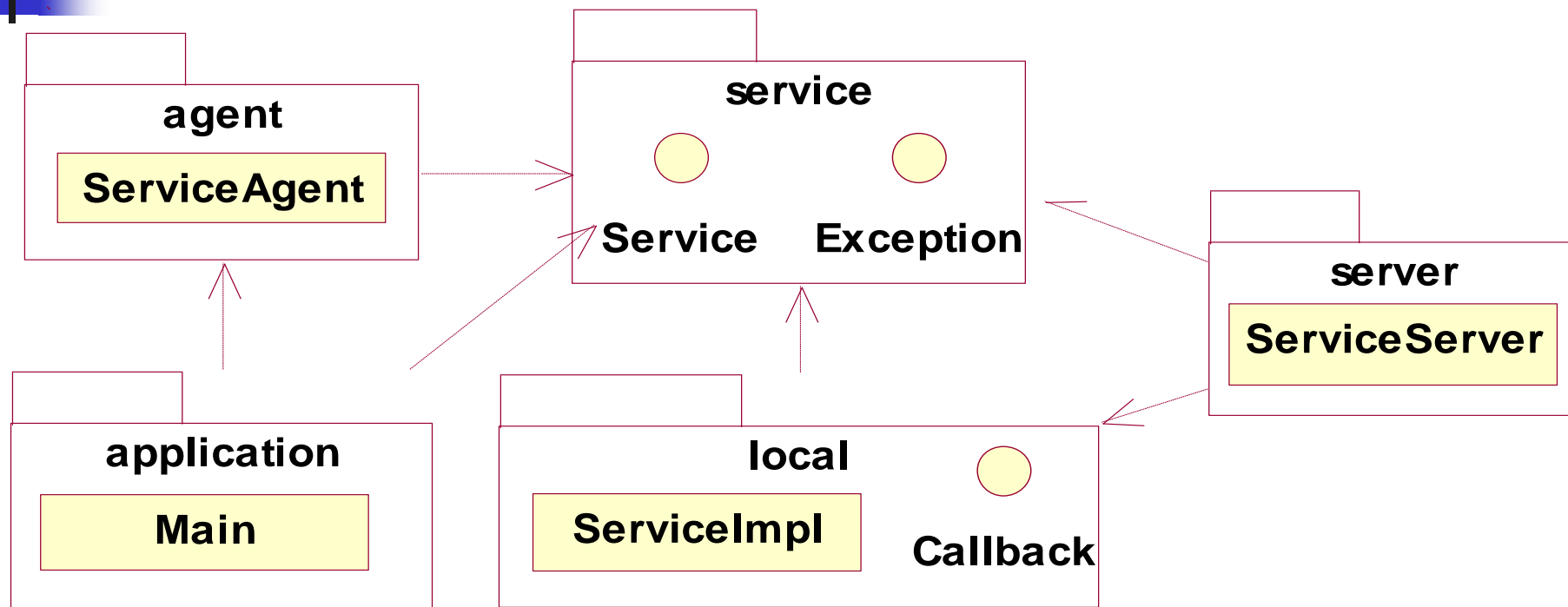


Нарушение CRP: remote service



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

CRP совместимое решение



Клиенты ServiceAgent зависят только от того что реально используют.

Выгоды: изменения в пакетах local и server не затрагивают клиентов



ССР: Локализация изменений

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

- R.Martin, 1996

□ Локализует изменения и снижает число версий.



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.

- R.Martin, 1996



Нестабильность пакета

Нестабильность пакета:

$$I = C_e / (C_a + C_e)$$

Где:

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

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

$I = 0$ – абсолютно стабильный пакет

$I = 1$ – очень нестабильный пакет



SAP – Стабильность абстракций

✓ Абстрактность пакета должна быть пропорциональна его стабильности.

- R.Martin, 1996

- ❖ если все пакеты стабильны – система немодифицируема.
- ❖ => некоторые пакеты просто обязаны быть нестабильными.
- ❖ вопрос: какие это пакеты?

❑ Упрощает поддержку (maintainability)



Абстрактность пакета

Абстрактность пакета

$$A = N_a / N$$

где

N_a = 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:

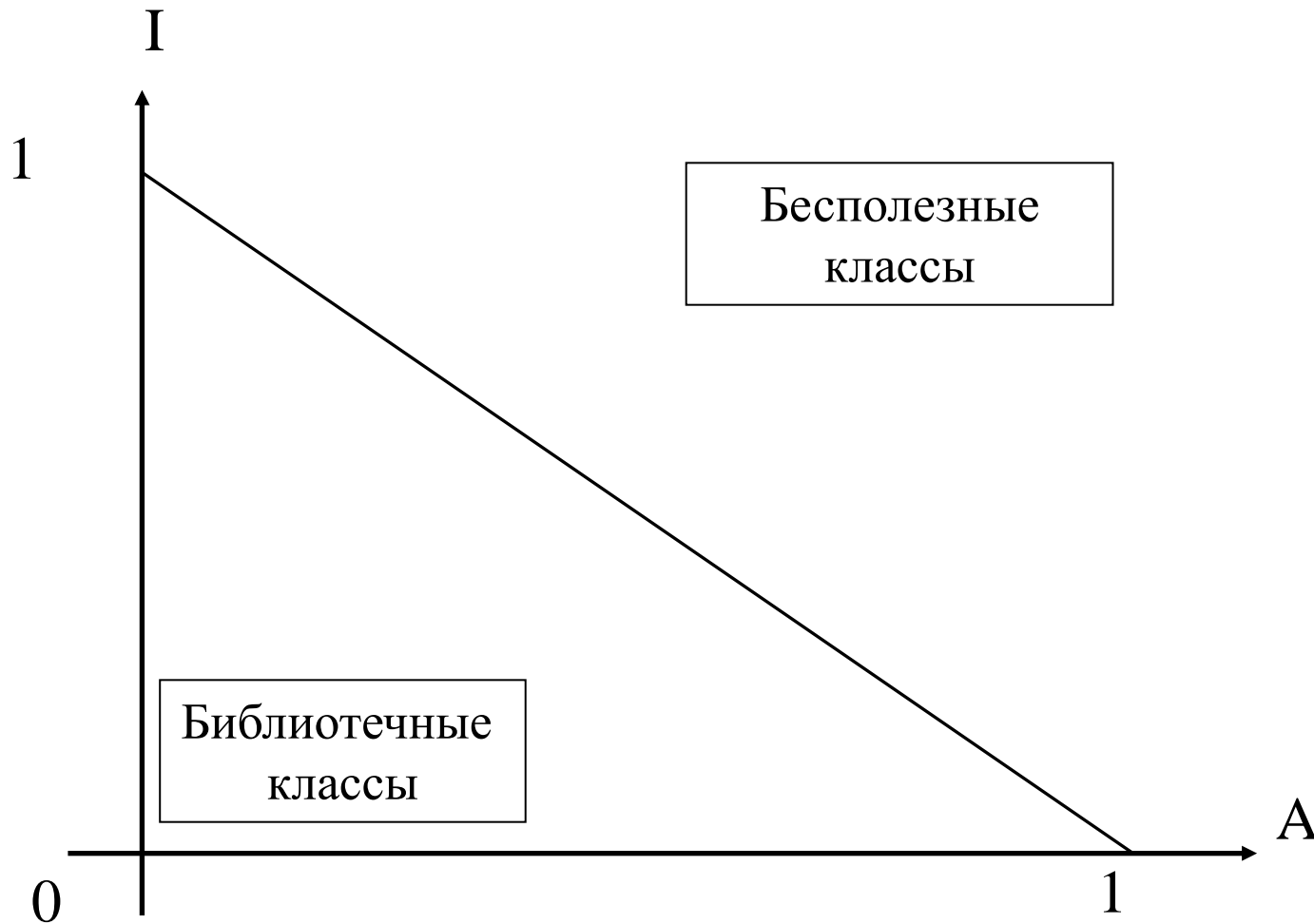
$$D = | A + I - 1 |$$

указывает на потенциальные проблемы в дизайне пакета.

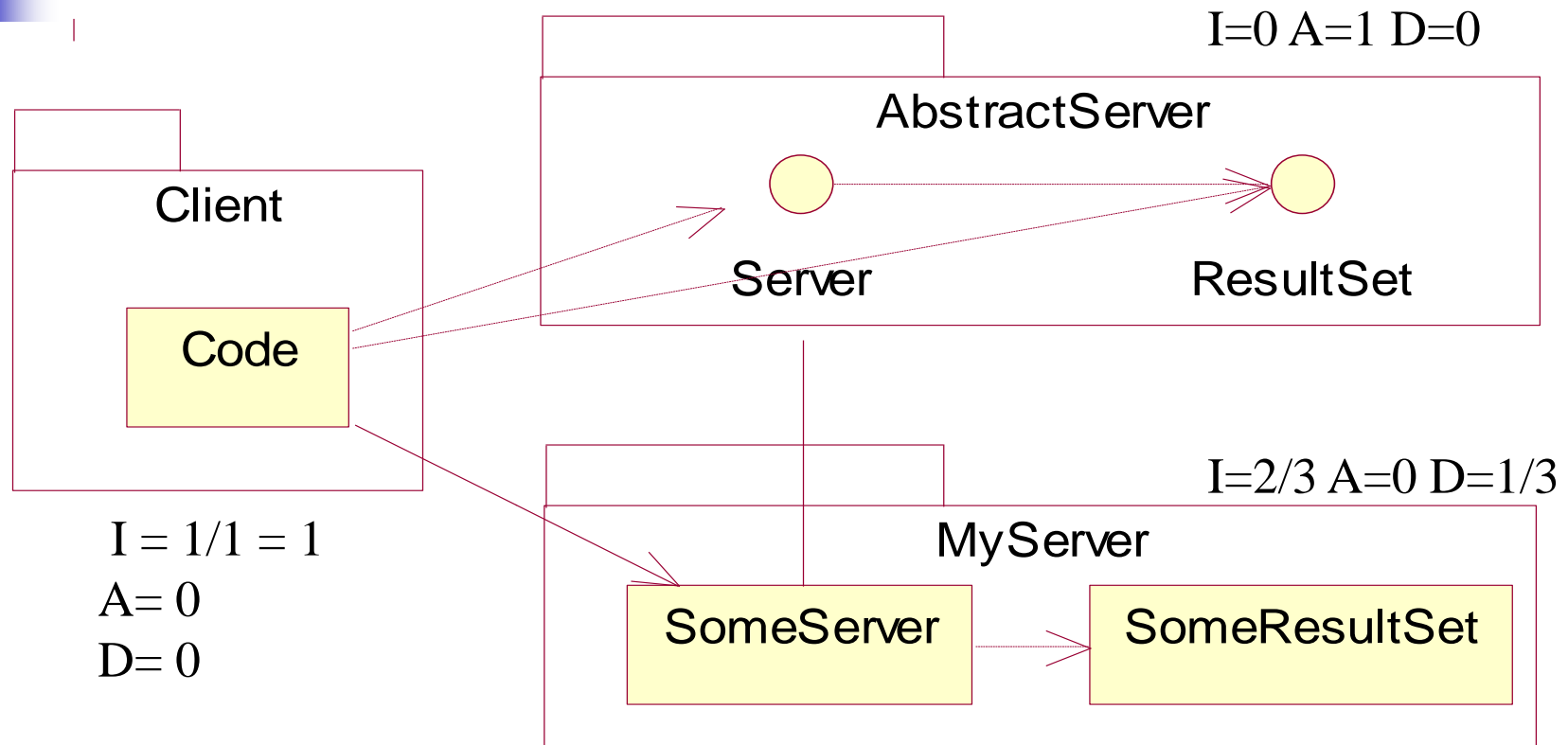
- R.Martin

Расчет I , A и D для Java реализован в утилите **Jdepend**

Main sequence

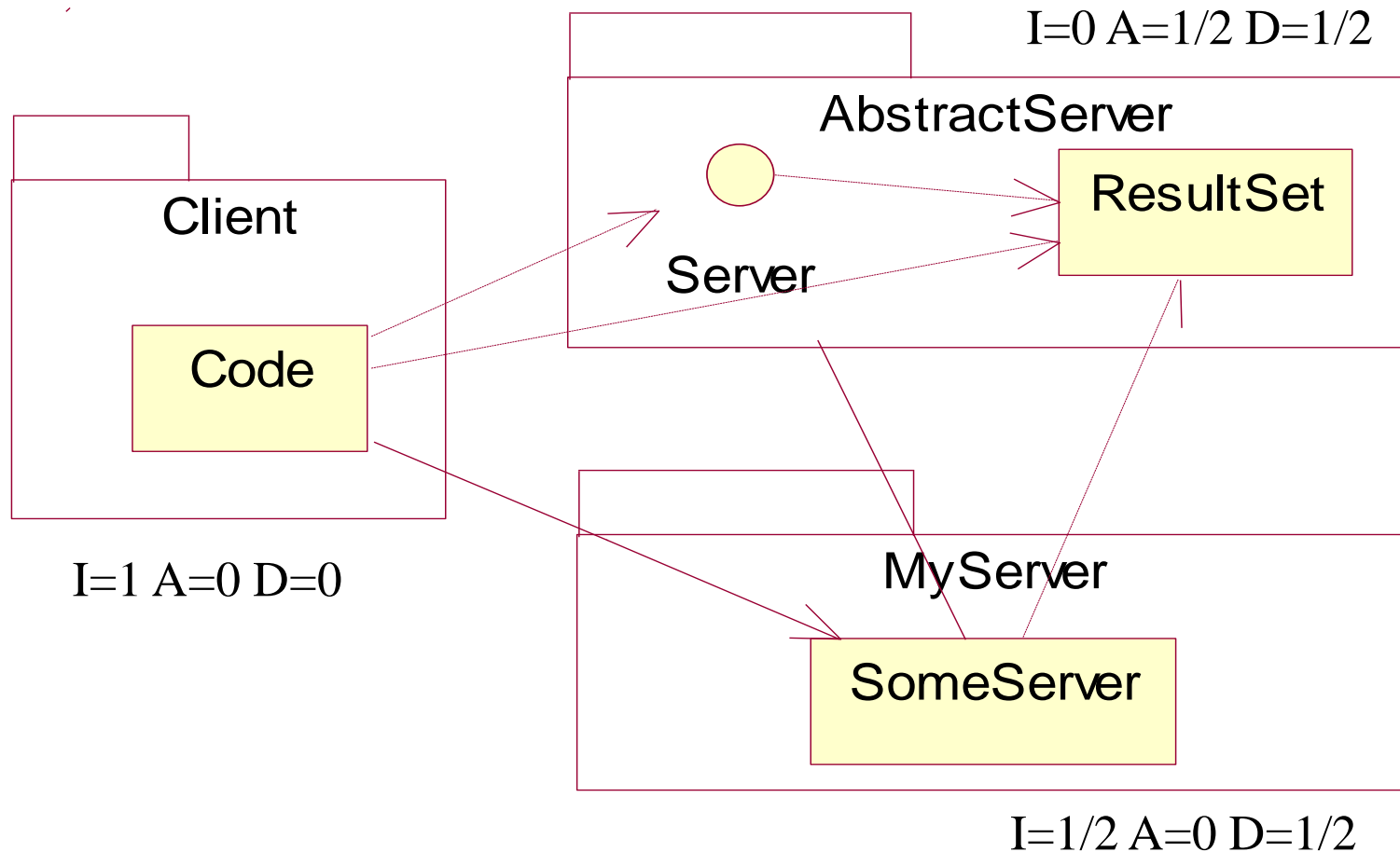


Пример: client-server



❖ Что случится если ResultSet не будет interface (что заведомо плохо)? Если наше правило работает – метрики должны измениться в худшую сторону

Пример: client-server



Увеличилась дистанция у обоих пакетов - MyServer и AbstractServer