SOLID принципы

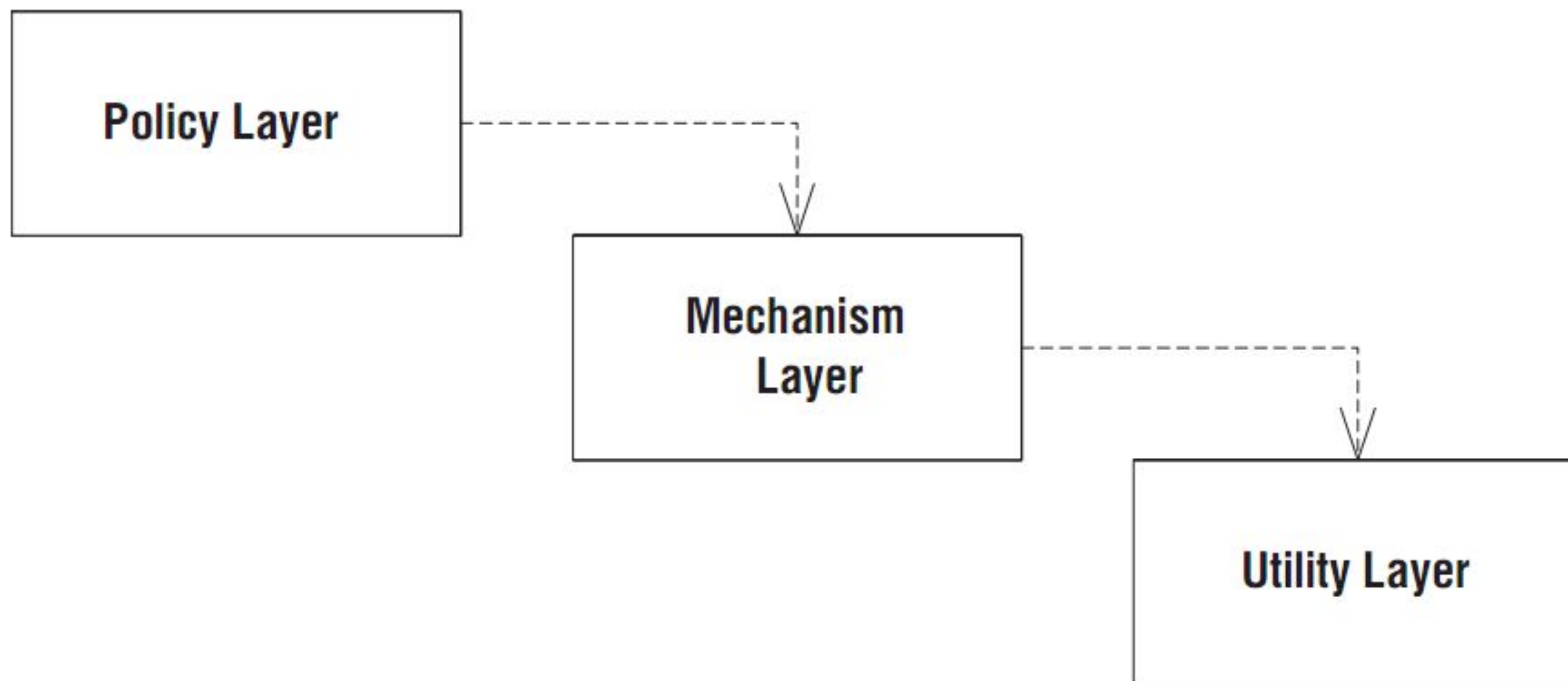
Принцип инверсии зависимости(DIP)

Принцип инверсии зависимости

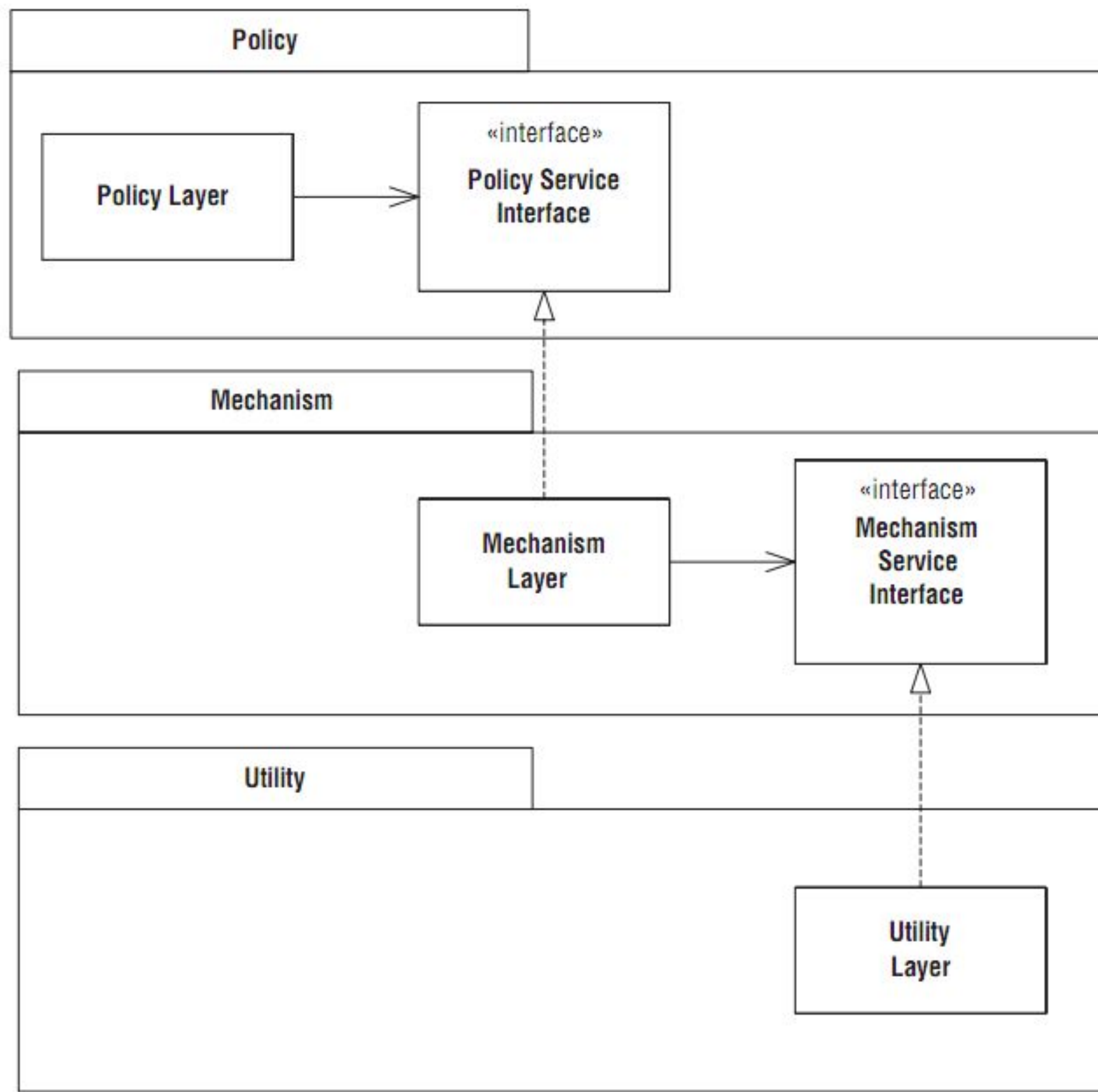
- Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те, и другие должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Разбиение на слои

В любой хорошо структурированной объектно-ориентированной архитектуре можно выделить ясно очерченные слои, на каждом из которых предоставляется некий набор тесно связанных служб – с помощью четко определенных и контролируемых интерфейсов



Наивная схема разбиения на слои



Инвертированные слои

Инверсия владения

Зависимость от абстракций

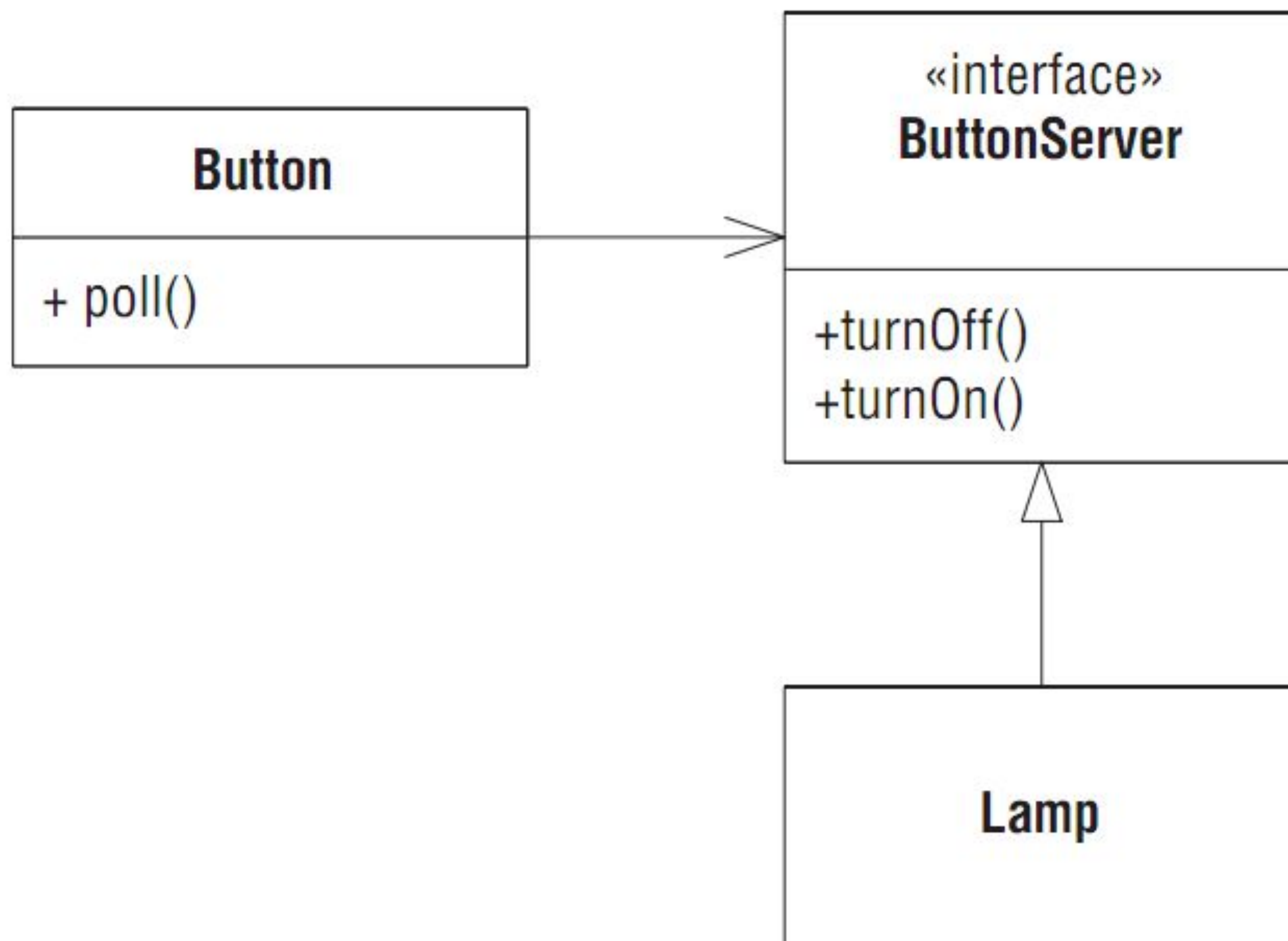
- Не должно быть переменных, в которых хранятся ссылки на конкретные классы.
- Не должно быть классов, производных от конкретных классов.
- Не должно быть методов, переопределяющих метод, реализованный в одном из базовых классов.

Простой пример принципа DIP

Наивная модель объектов Button и Lamp

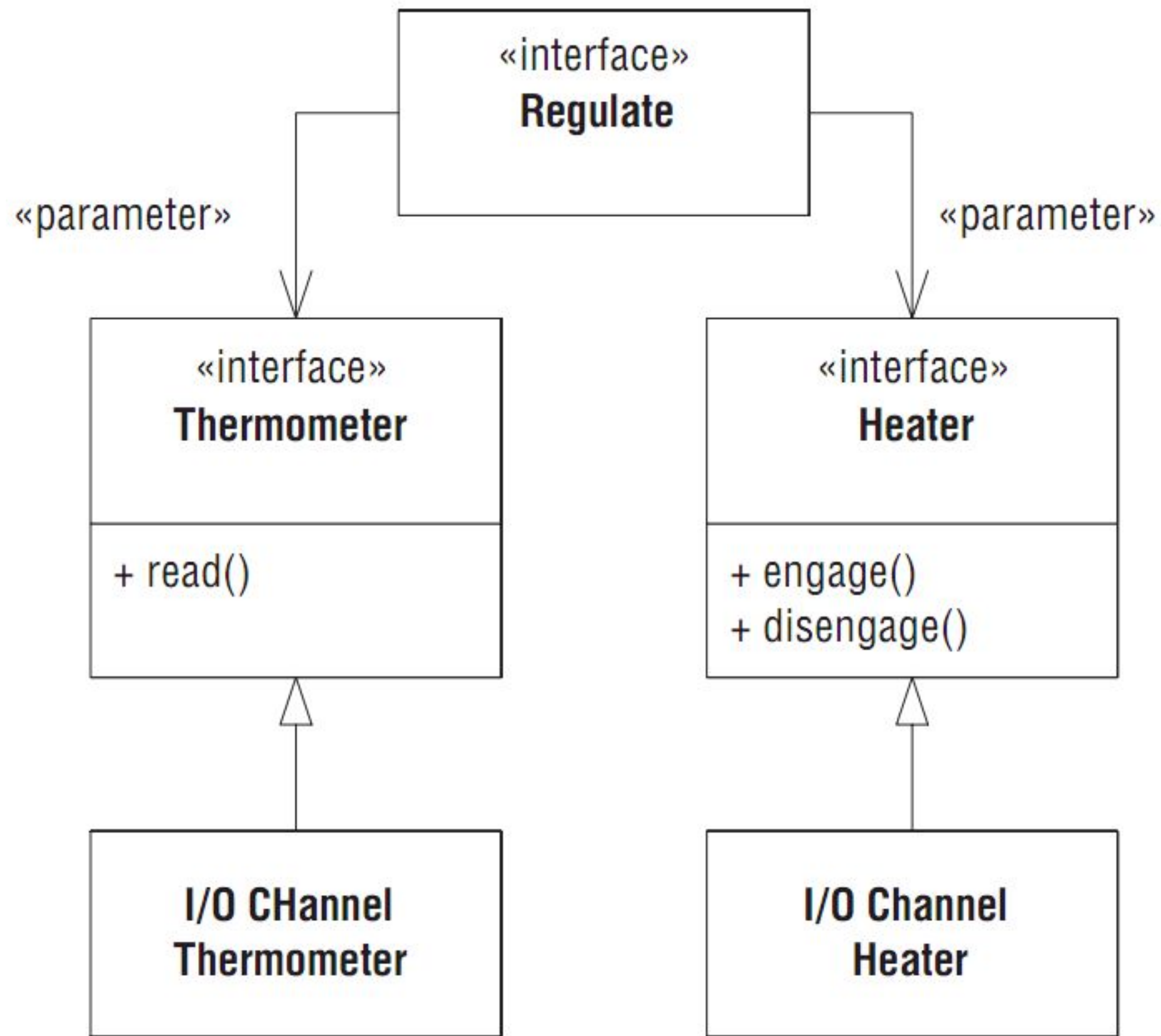
```
public class Button
{
    private Lamp lamp;

    public void Poll()
    {
        if ( /* какое-то условие */)
            lamp.TurnOn();
    }
}
```



Инверсия зависимости в применении к Lamp

```
const byte TERMOMETER = 0x86;
const byte FURNACE = 0x87;
const byte ENGAGE = 1;
const byte DISENGAGE = 0;
void Regulate(double minTemp, double maxTemp)
{
    for(;;)
    {
        while (in(TERMOMETER) > minTemp)
            wait(1);
        out(FURNACE,ENGAGE);
        while (in(TERMOMETER) < maxTemp)
            wait(1);
        out(FURNACE,DISENGAGE);
    }
}
```



Регулятор общего вида

```
void Regulate(Thermometer t, Heater h, double minTemp, double maxTemp)
{
    for(;;)
    {
        while (t.Read() > minTemp)
            wait(1);
        h.Engage();
        while (t.Read() < maxTemp)
            wait(1);
        h.Disengage();
    }
}
```

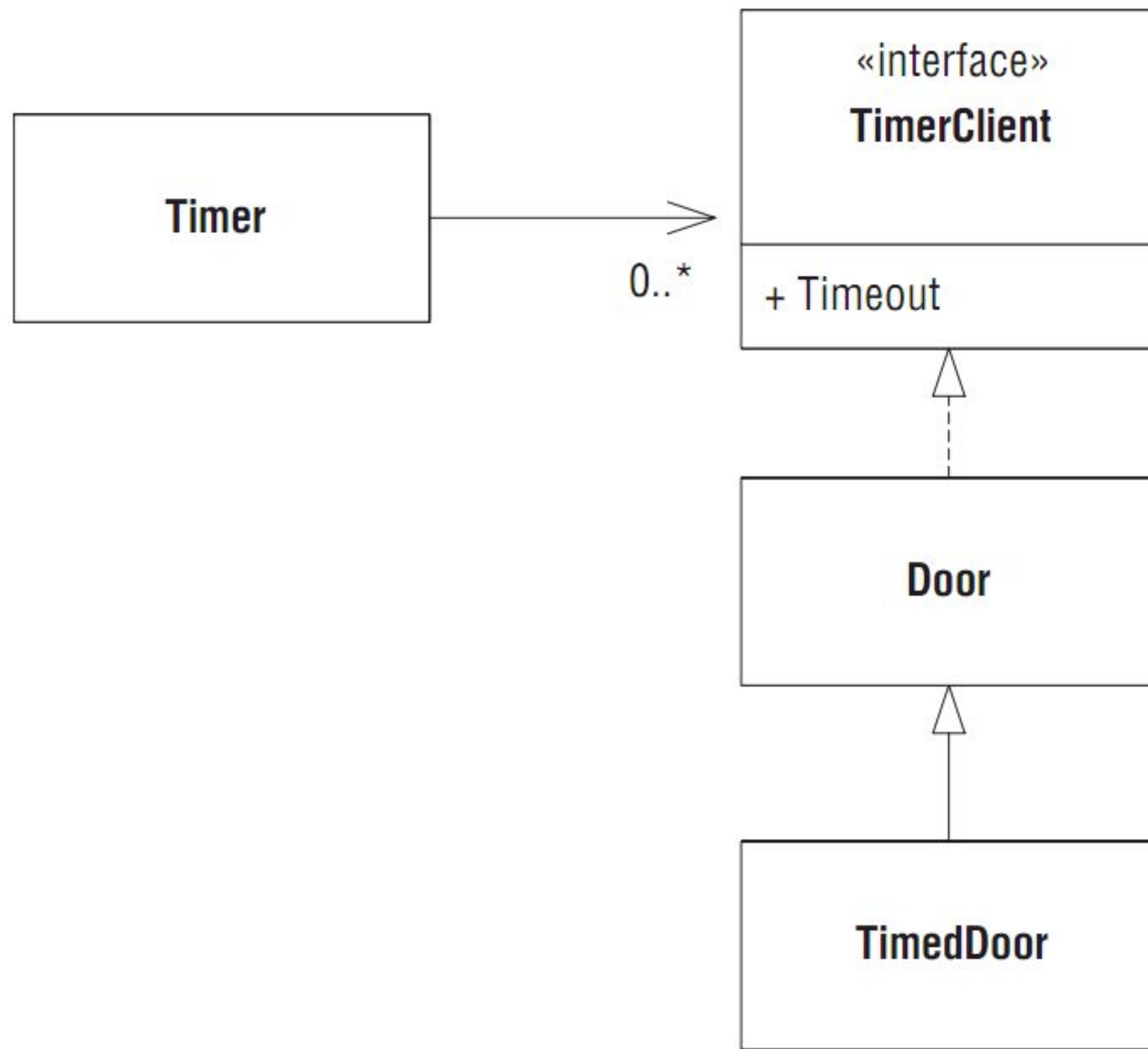
Принцип разделения интерфейсов (ISP)

Клиенты не должны вынужденно зависеть от методов, которы
ми не пользуются

Загрязнение интерфейса

```
public interface Door
{
    void Lock();
    void Unlock();
    bool IsDoorOpen();
}
```

```
public class Timer
{
    public void Register(int timeout, TimerClient client)
    {/* код */}
}
public interface TimerClient
{
    void TimeOut();
}
```



Класс TimerClient на вершине иерархии

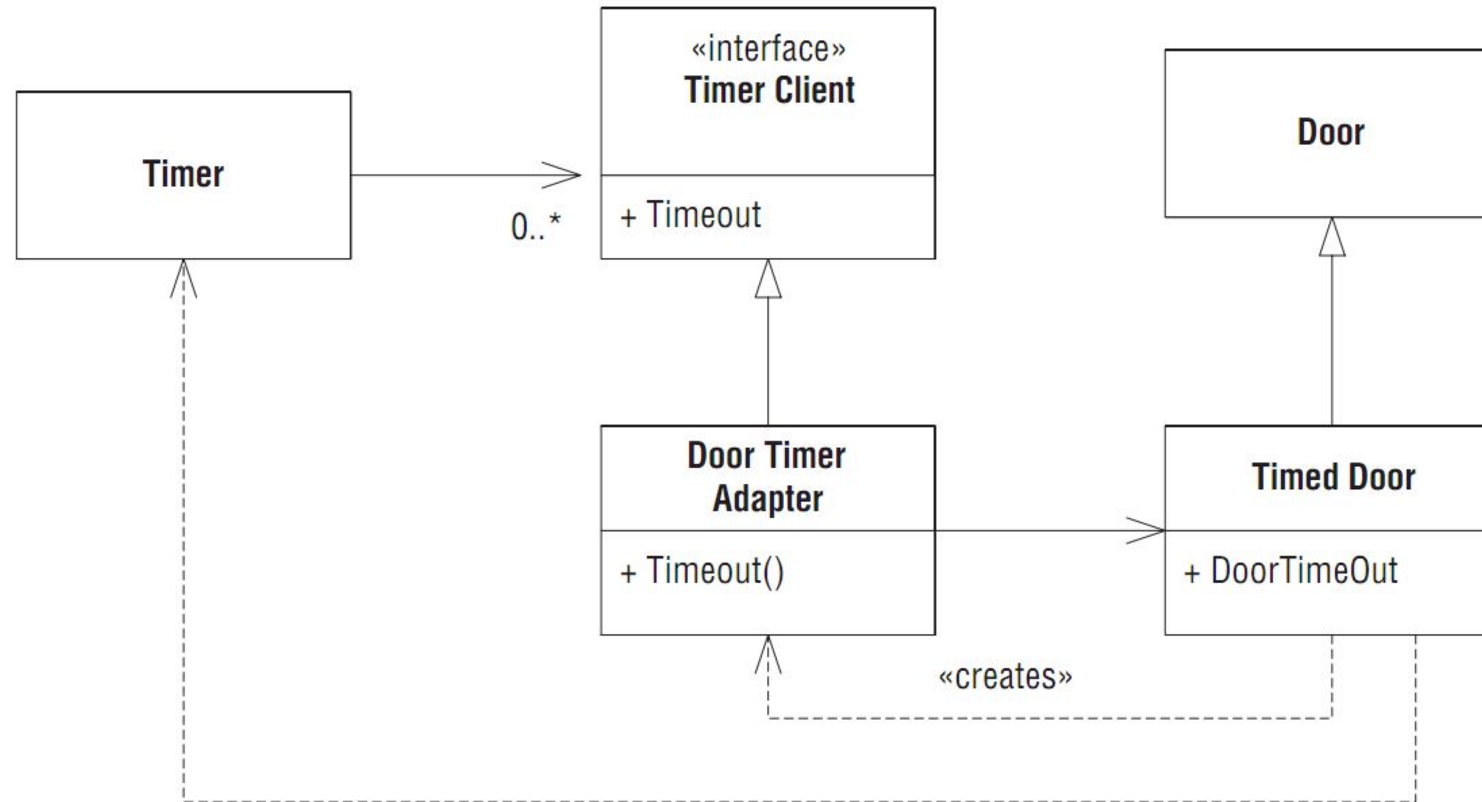
Разделение клиентов означает разделение интерфейсов

```
public class Timer
{
    public void Register(int timeout, int timeOutId, TimerClient client)
    {/* код */}
}
public interface TimerClient
{
    void TimeOut(int timeOutID);
}
```

Принцип разделения интерфейсов (Interface Segregation Principle)

Клиенты не должны вынужденно зависеть от методов, которы
ми не пользуются

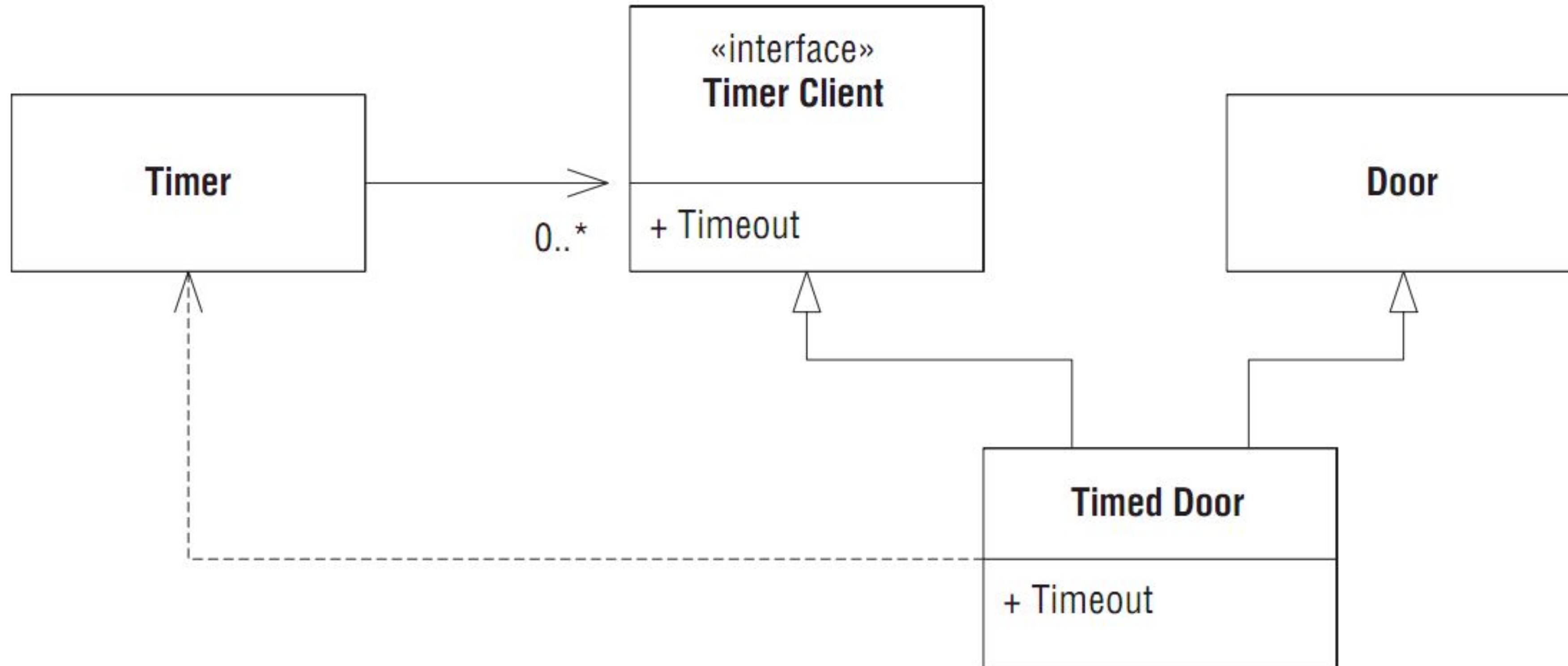
Разделение путем делегирования



Адаптер таймера двери

```
public interface TimedDoor : Door
{
    void DoorTimeOut(int timeOutId);
}
public class DoorTimerAdapter : TimerClient
{
    private TimedDoor timedDoor;
    public DoorTimerAdapter(TimedDoor theDoor)
    {
        timedDoor = theDoor;
    }
    public virtual void TimeOut(int timeOutId)
    {
        timedDoor.DoorTimeOut(timeOutId);
    }
}
```

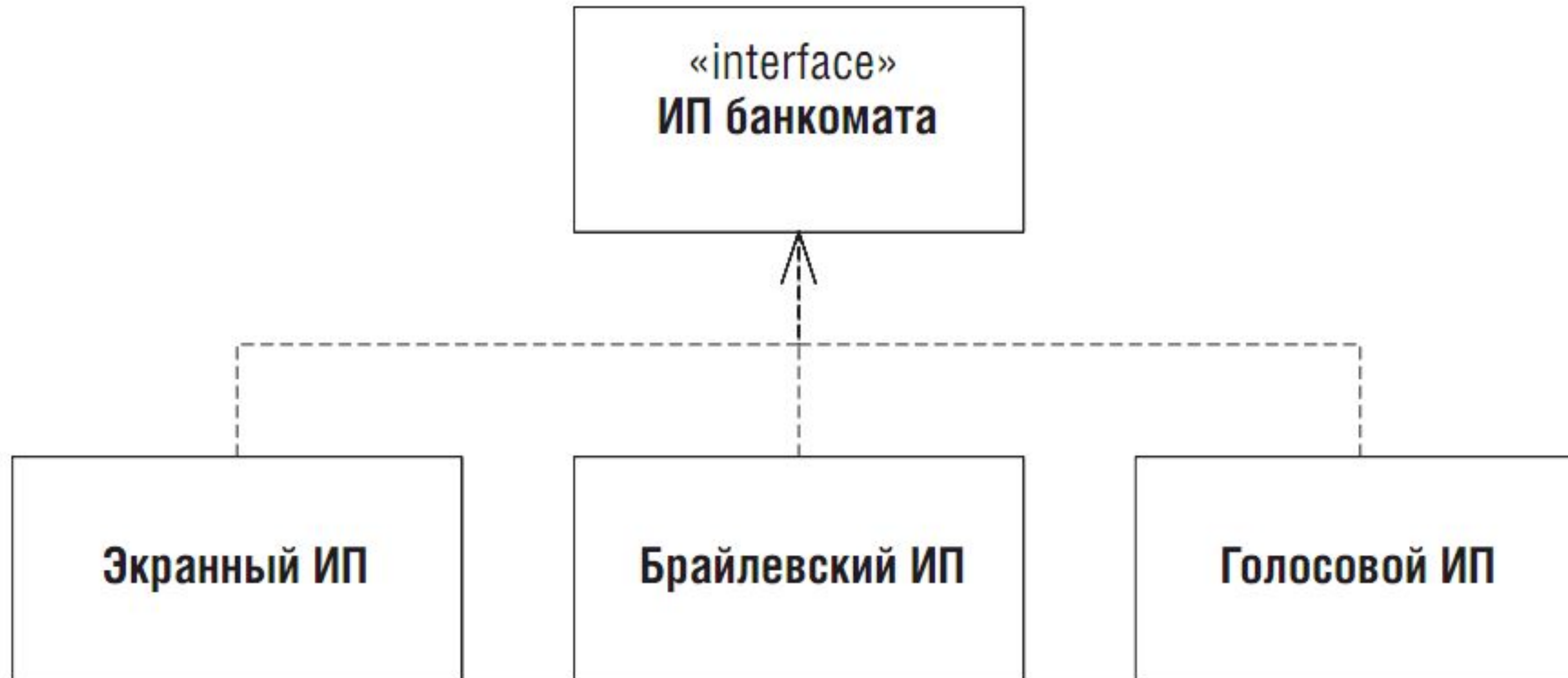
Разделение путем множественного наследования



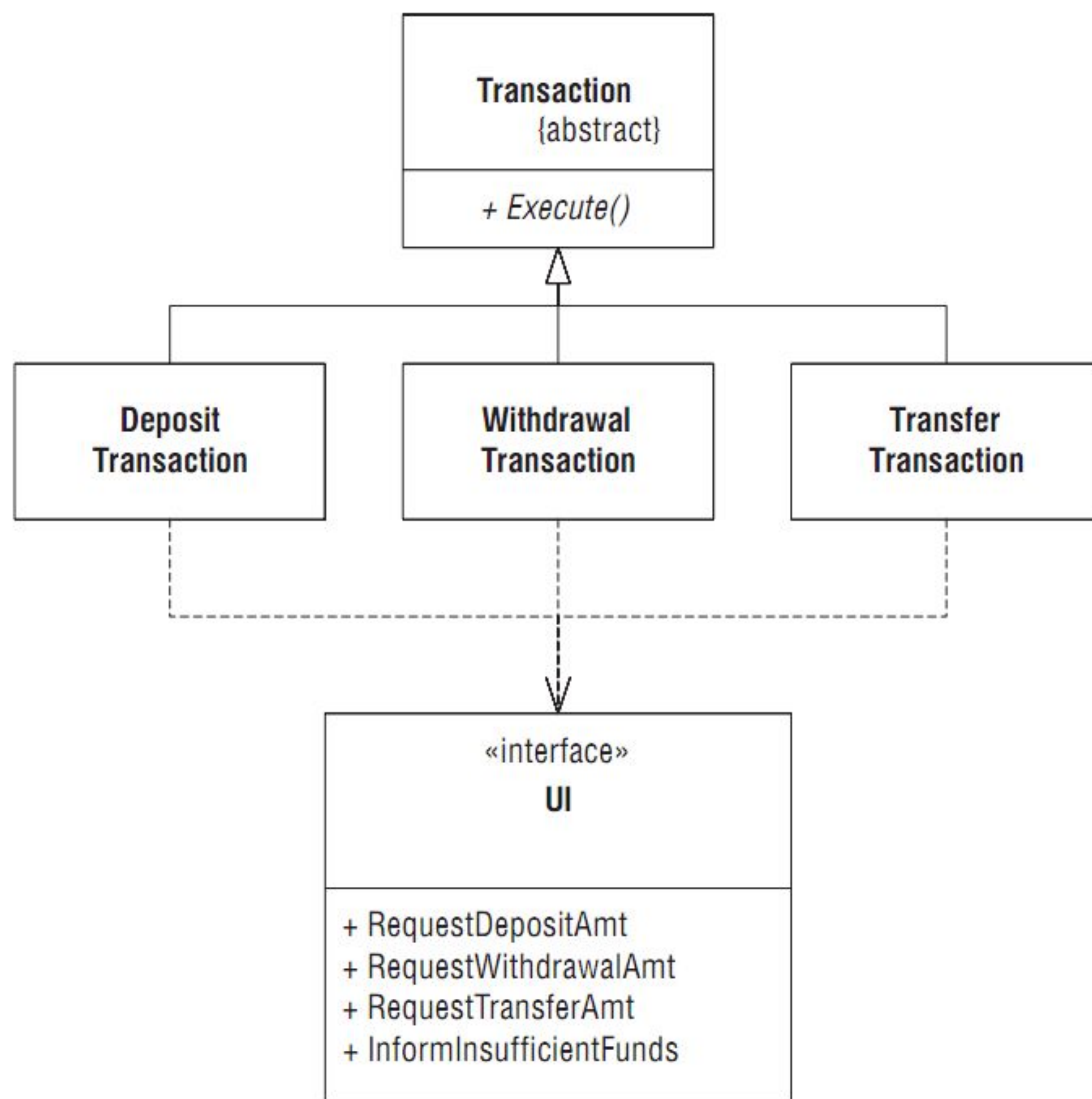
Класс TimedDoor с множественным наследованием

```
public interface TimedDoor : Door, TimerClient
{
}
```

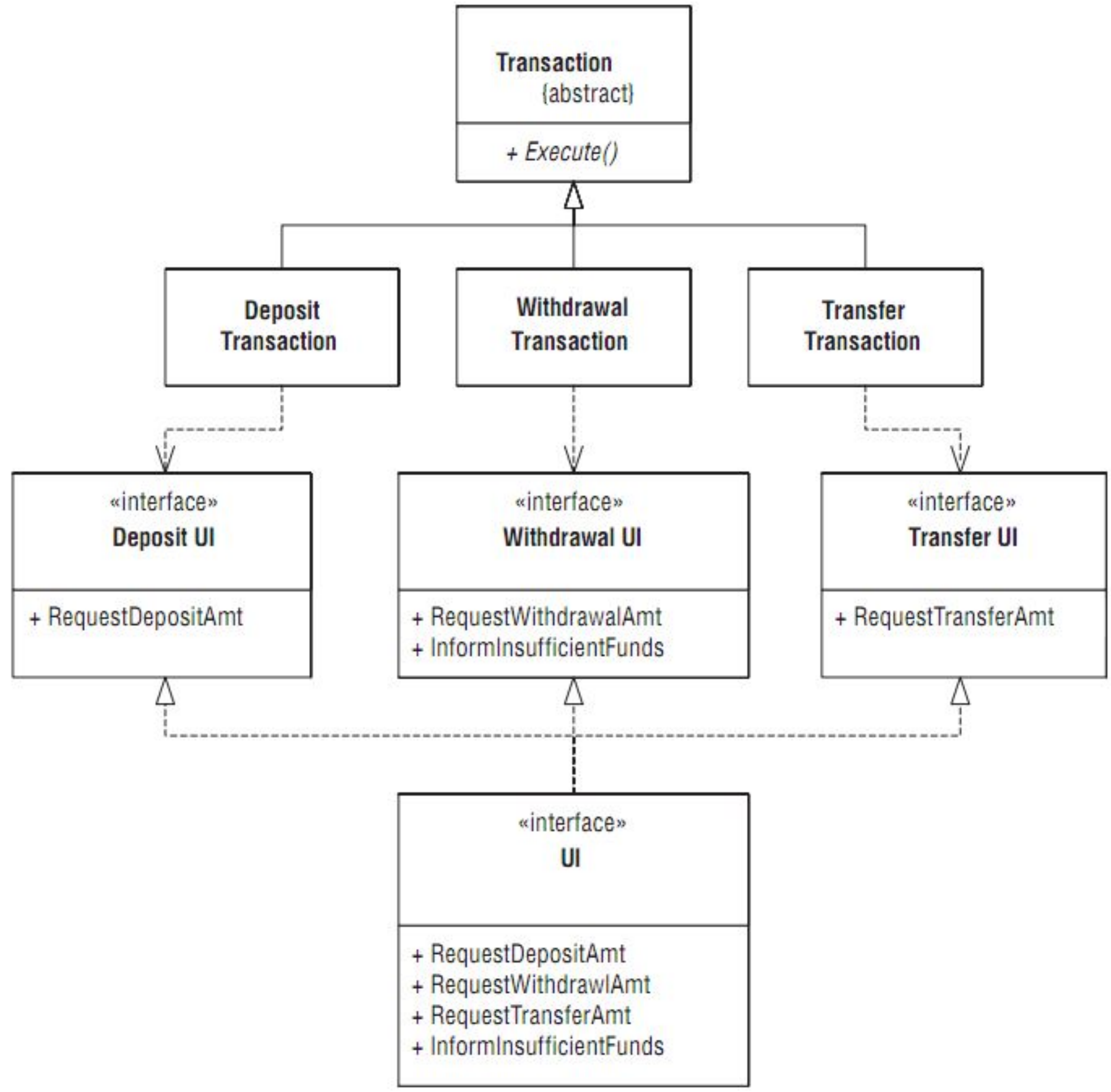
Пользовательский интерфейс банкомата



Пользовательский интерфейс банкомата



Иерархия транзакций банкомата



Разделенный интерфейс ИП банкомата

```csharp
public interface Transaction
{
    void Execute();
}
public interface DepositUI
{
    void RequestDepositAmount();
}

public interface WithdrawalUI
{
    void RequestWithdrawalAmount();
}

public interface TransferUI
{
    void RequestTransferAmount();
}

public interface UI : DepositUI, WithdrawalUI, TransferUI
{
}
```

```
public class DepositTransaction : Transaction
{
    private DepositUI depositUI;
    public DepositTransaction(DepositUI ui)
    {
        depositUI = ui;
    }
    public virtual void Execute()
    {
        /* код */
        depositUI.RequestDepositAmount();
        /* код */
    }
}
```

```
public class WithdrawalTransaction : Transaction
{
    private WithdrawalUI withdrawalUI;
    public WithdrawalTransaction(WithdrawalUI ui)
    {
        withdrawalUI = ui;
    }
    public virtual void Execute()
    {
        /* код */
        withdrawalUI.RequestWithdrawalAmount();
        /* код */
    }
}
```

```csharp
public class TransferTransaction : Transaction
{
    private TransferUI transferUI;
    public TransferTransaction(TransferUI ui)
    {
        transferUI = ui;
    }

    public virtual void Execute()
    {
        /* код */
        transferUI.RequestTransferAmount();
        /* код */
    }
}
```

```
public class UIGlobals
{
    public static WithdrawalUI withdrawal;
    public static DepositUI deposit;
    public static TransferUI transfer;
    static UIGlobals()
    {
        UI Lui = new AtmUI(); // какая-то реализация ИП
        UIGlobals.deposit = Lui;
        UIGlobals.withdrawal = Lui;
        UIGlobals.transfer = Lui;
    }
}
```