



# Design Patterns

**Mikhail Karpuk**  
Software developer

Ittransition training courses

## Содержание

- Введение
- GoF patterns
- MVC & MVP
- Antipatterns

## Зачем это надо?

- Чтобы не решать каждую задачу с нуля
- Использовать проверенные временем решения
- Заранее представлять последствия выбора того или иного варианта
- Проектировать с учетом будущих изменений
- Писать понятный и компактный код, который легко можно будет использовать повторно

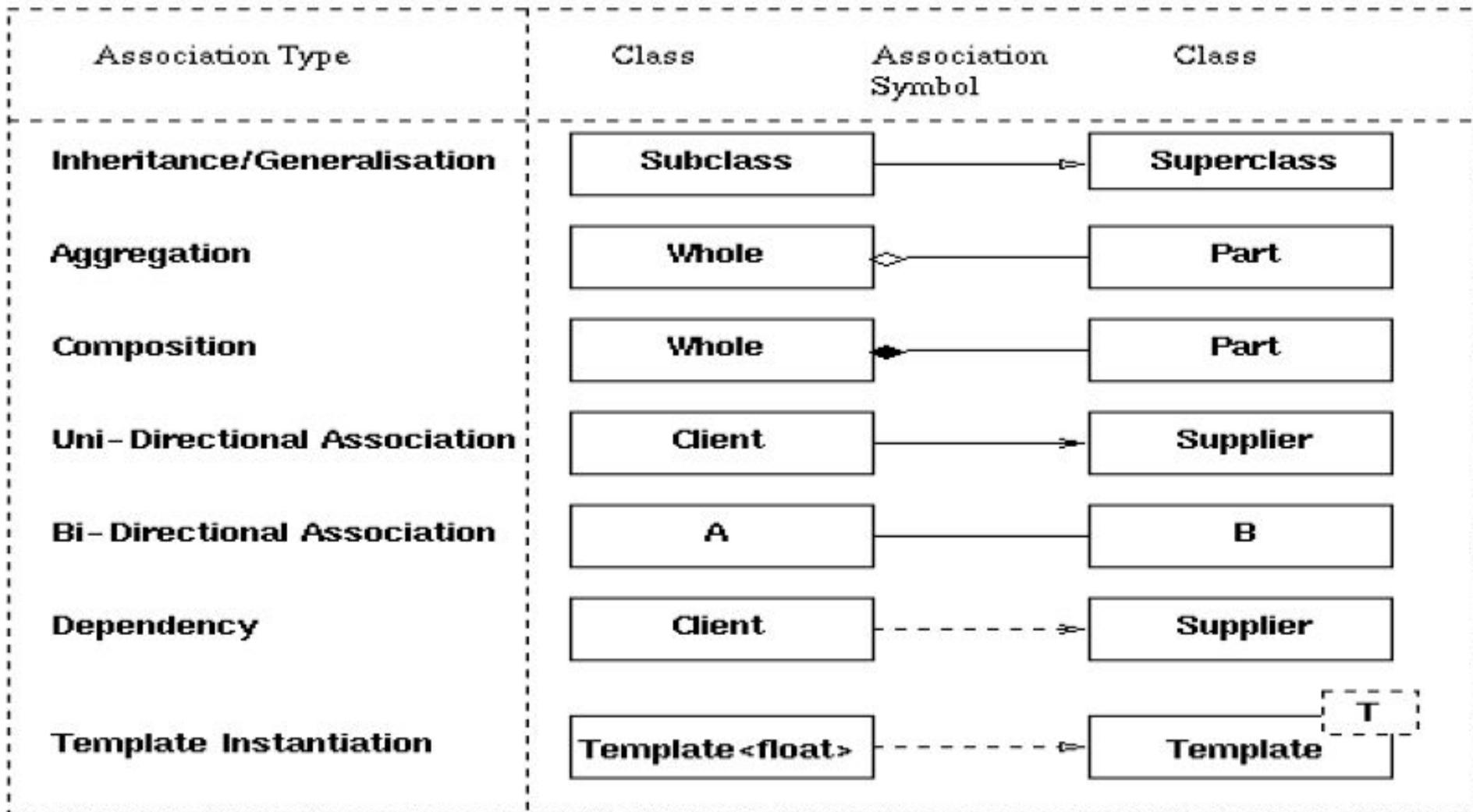
## Что это такое?

- Кристофер Александр: «Любой паттерн описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем таким образом, что это решение можно использовать миллион раз, ничего не изобретая заново»
- Это не кусок кода, класс или dll
- Это идея, метод решения, общий подход к целому классу задач, постоянно встречающихся на практике

## Описание паттерна

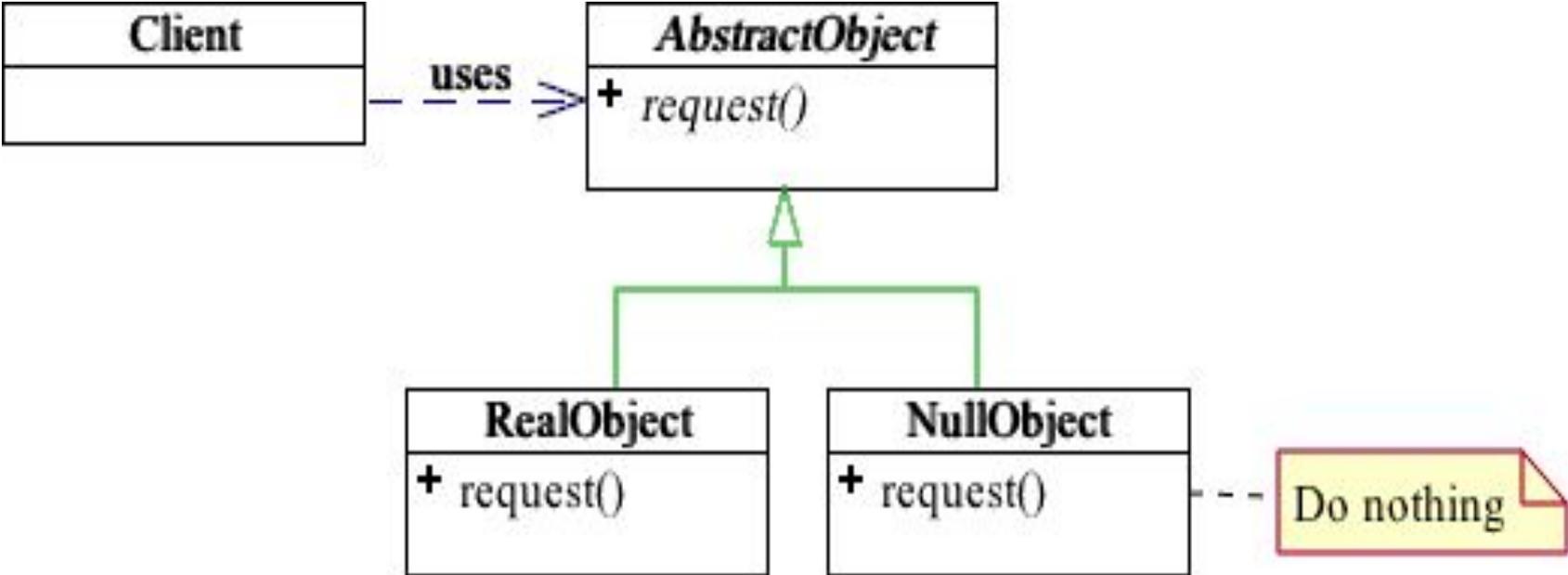
- Имя (имена)
- Задача (когда применять паттерн)
- Решение
- Результаты (следствия и компромиссы)

## Association-Symbols



## Null Value pattern

- В некоторых ситуациях объект не может быть создан или не должен ничего делать.
- Постоянные проверки сильно загромождают код и могут привести к ошибкам (Copy/Paste anti pattern)



## Результаты

- Упрощение кода
- Унификация вызовов
- Уменьшение количества ошибок

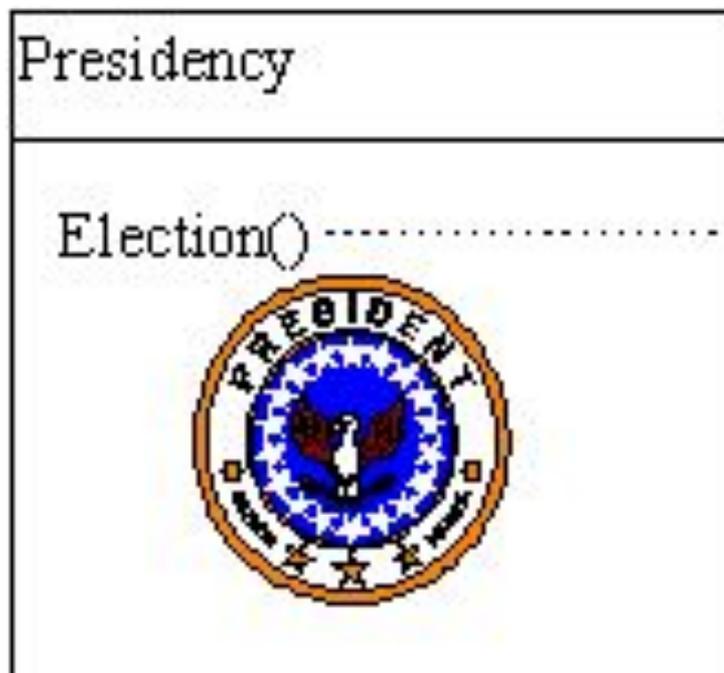


## Порождающие паттерны

- Обеспечивают независимость системы от процесса создания новых сущностей
- Singleton
- Abstract Factory

## Singleton

- Гарантирует, что у класса есть только один экземпляр и предоставляет к нему глобальную точку доступа
- Единственный экземпляр может расширяться путем порождения подклассов



Return unique-instance

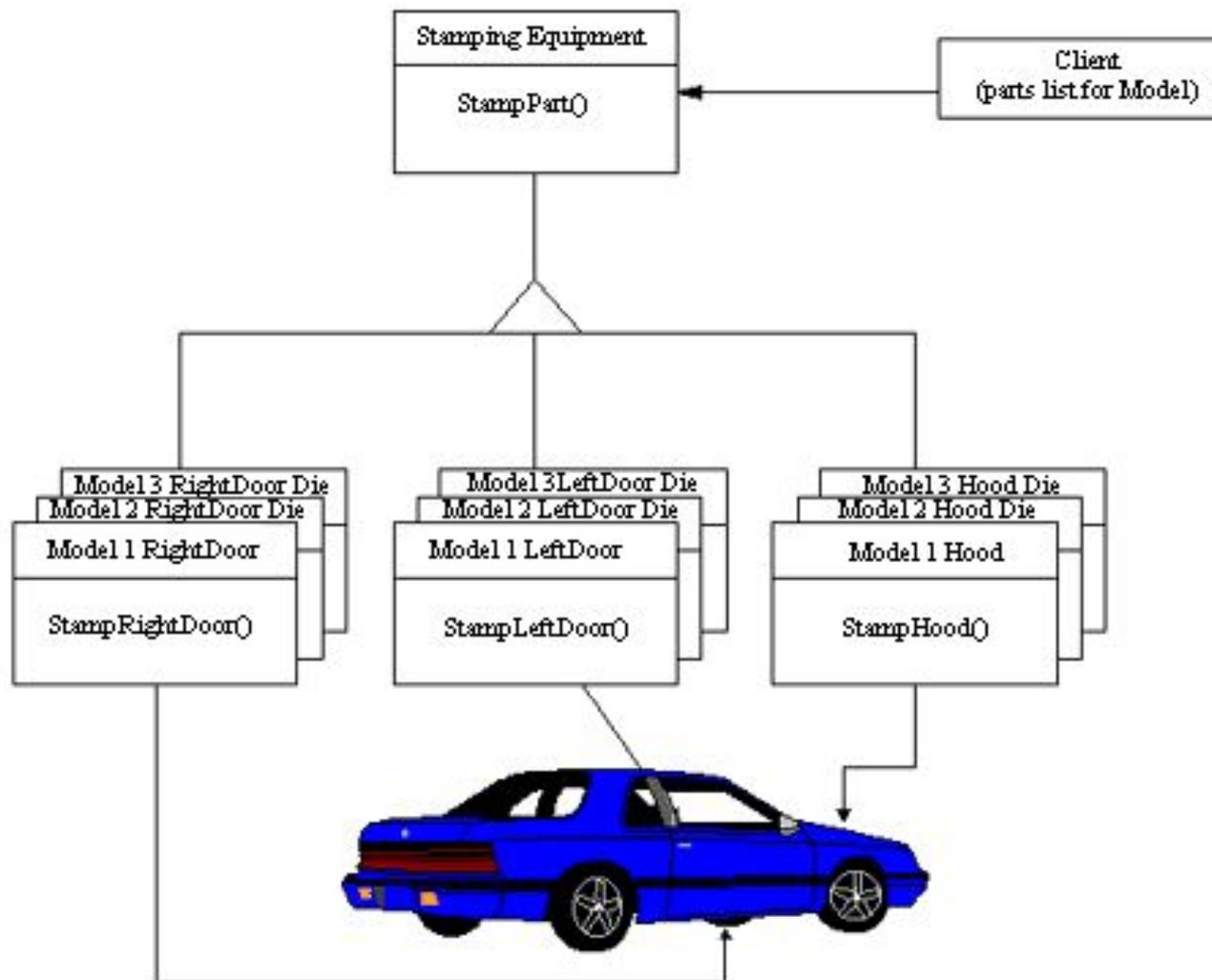
## Пример кода

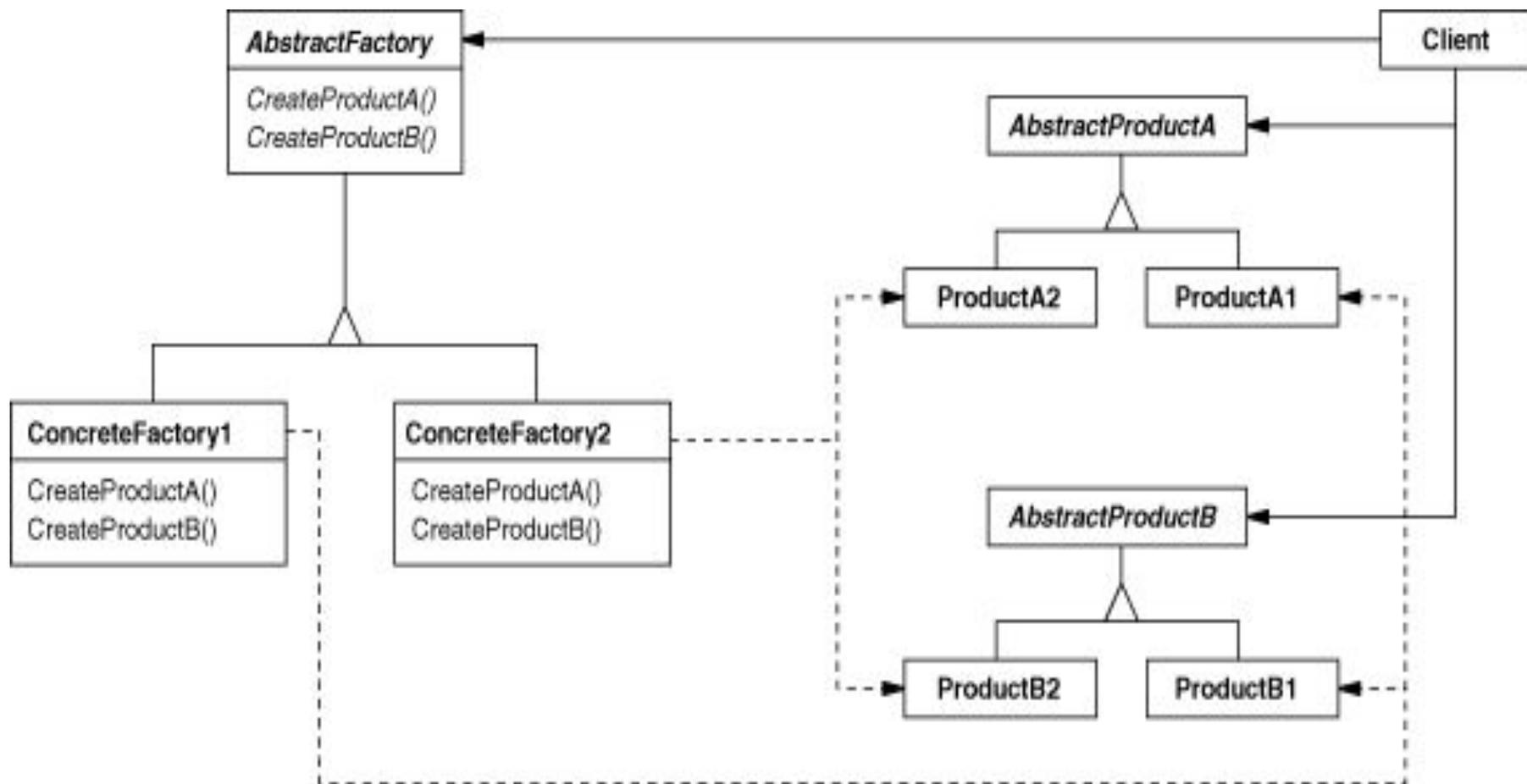
```
public class Singleton
{
    private static readonly Singleton singleton = new Singleton();
    private Singleton()
    {
        // Note private constructor
    }
    public static Singleton GetSingleton()
    {
        return singleton;
    }
    public void SomeFunction() { }
}

static void Main()
{
    Singleton.GetSingleton().SomeFunction();
}
```

## Abstract Factory (Kit)

- Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя конкретных классов
- Позволяет легко перейти на использование другого семейства
- Добавление новых объектов в семейство затруднено





## Пример

```
public class ToolFactory
{
    private ToolFactory(){}

    public static ToolFactory GetFactory()
    {
        //... - Singleton
    }

    public ButtonTool CreateButtonTool(Tool tool, EventHandler toolClickHandler)
    {
        //Create Button Tool, add handler, set icon, caption, etc
    }
    public ButtonTool CreateStateButtonTool(Tool tool)
    {
        //Create Button Tool , caption, etc
    }
}
```

# Паттерн Abstract Factory в действии

```
Public class SingleClickButtonTool : ButtonTool
{
    private bool inHandler = false;
    private EventHandler handler;

    public SingleClickButtonTool (EventHandler handler)

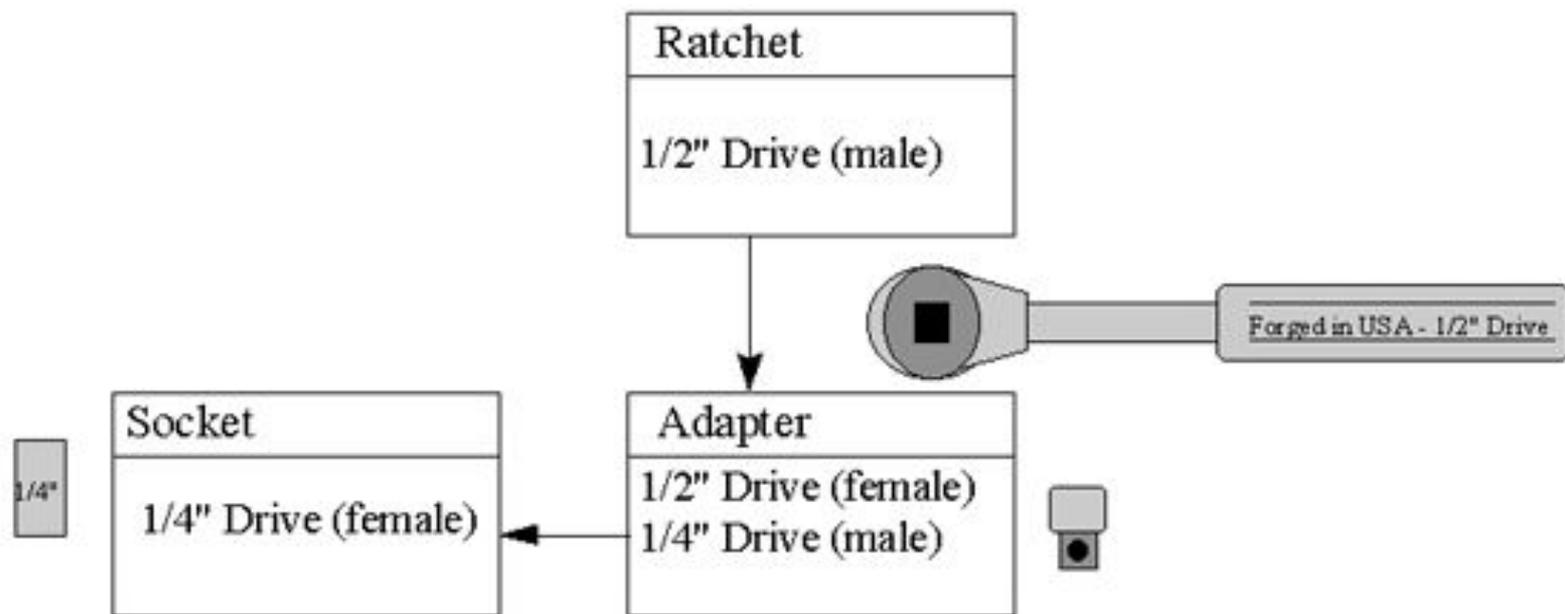
    OnButtonClick()
    {
        if (inHandler){return;}
        inHandler = true;
        try
        {
            handler.Invoke();
        }
        finally
        {
            inHandler = false;
        }
    }
}
```

## Структурные паттерны

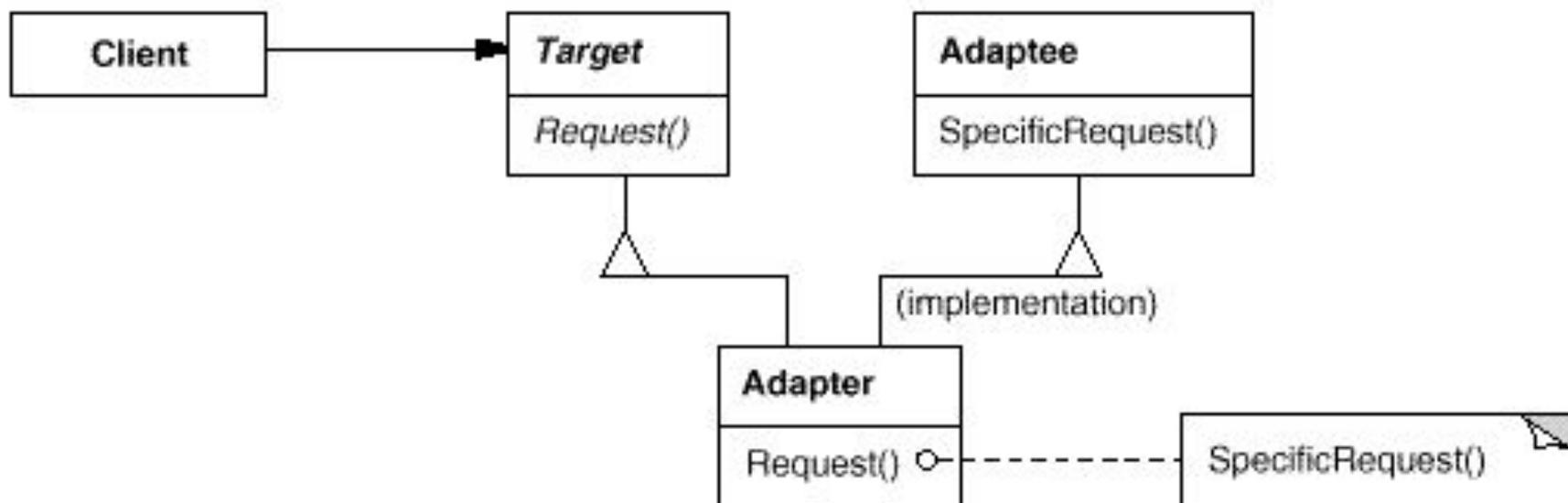
- Определяют, как классы и объекты группируются в более сложные структуры
- Adapter
- Facade
- Proxy
- Composite

## Adapter (Wrapper)

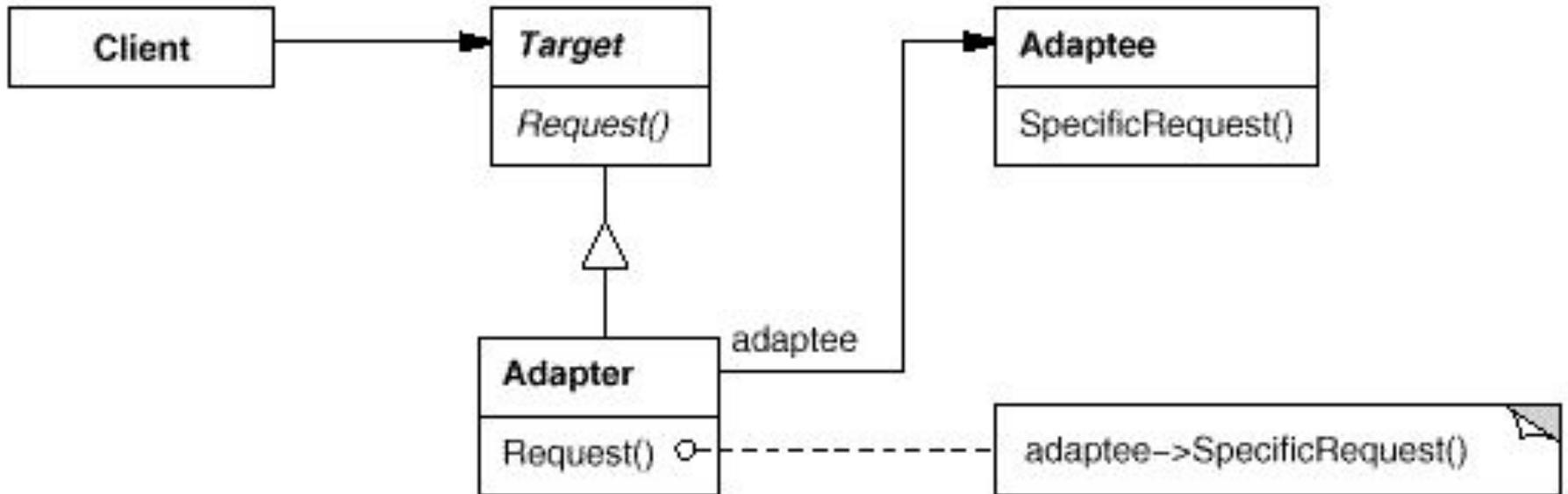
- Преобразует интерфейс одного класса к интерфейсу, ожидаемому клиентом



# Решение - множественное наследование



# Решение - использовать композицию объектов



## Пример кода

```
public class TextControlAdapter : BaseUserControl
{
    TextControl innerControl; String rtfText;
    public String RtfText
    {
        get
        {
            if (innerControl.Loaded) {return innerControl.Rtf;}
            else {return rtfText;}
        }
        set
        {
            if (innerControl.Loaded) {innerControl.Rtf = value;}
            else {rtfText = value;}
        }
    }
    public TextControlAdapter()
    {
        innerControl = new TextControl();
    }
}
```

## Facade

- Определяет интерфейс более высокого уровня
- Скрывает ненужные связи и зависимости
- Упрощает использование подсистемы

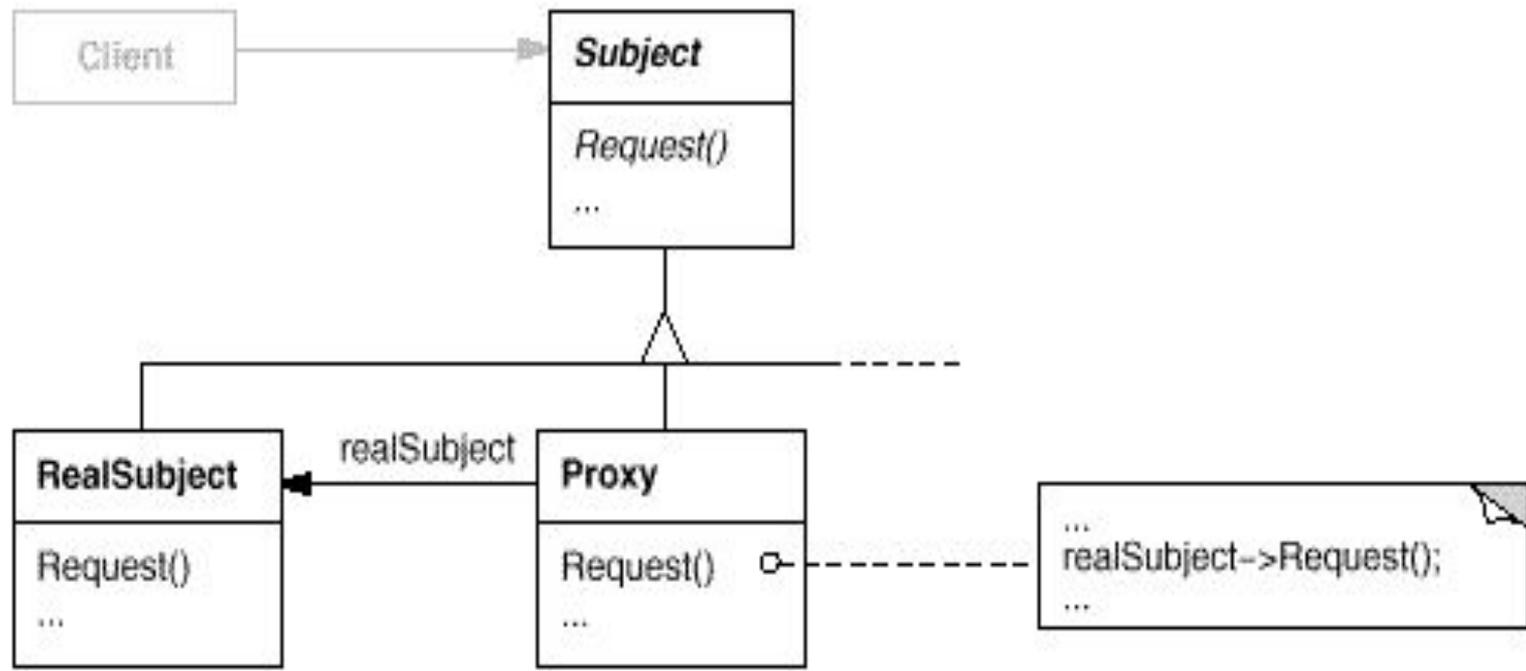


## Результат

- Фасад переадресует запросы клиента классам системы, сами классы не «знают» о существовании фасада
- Не нужно вникать во все тонкости и нюансы системы
- Фасад не закрывает доступ к элементам системы, при необходимости можно использовать систему и в обход фасада

## Proxy (Surrogate)

- Является суррогатом другого объекта и контролирует доступ к нему
- «Создает тяжелые объекты только по требованию»



## Пример кода

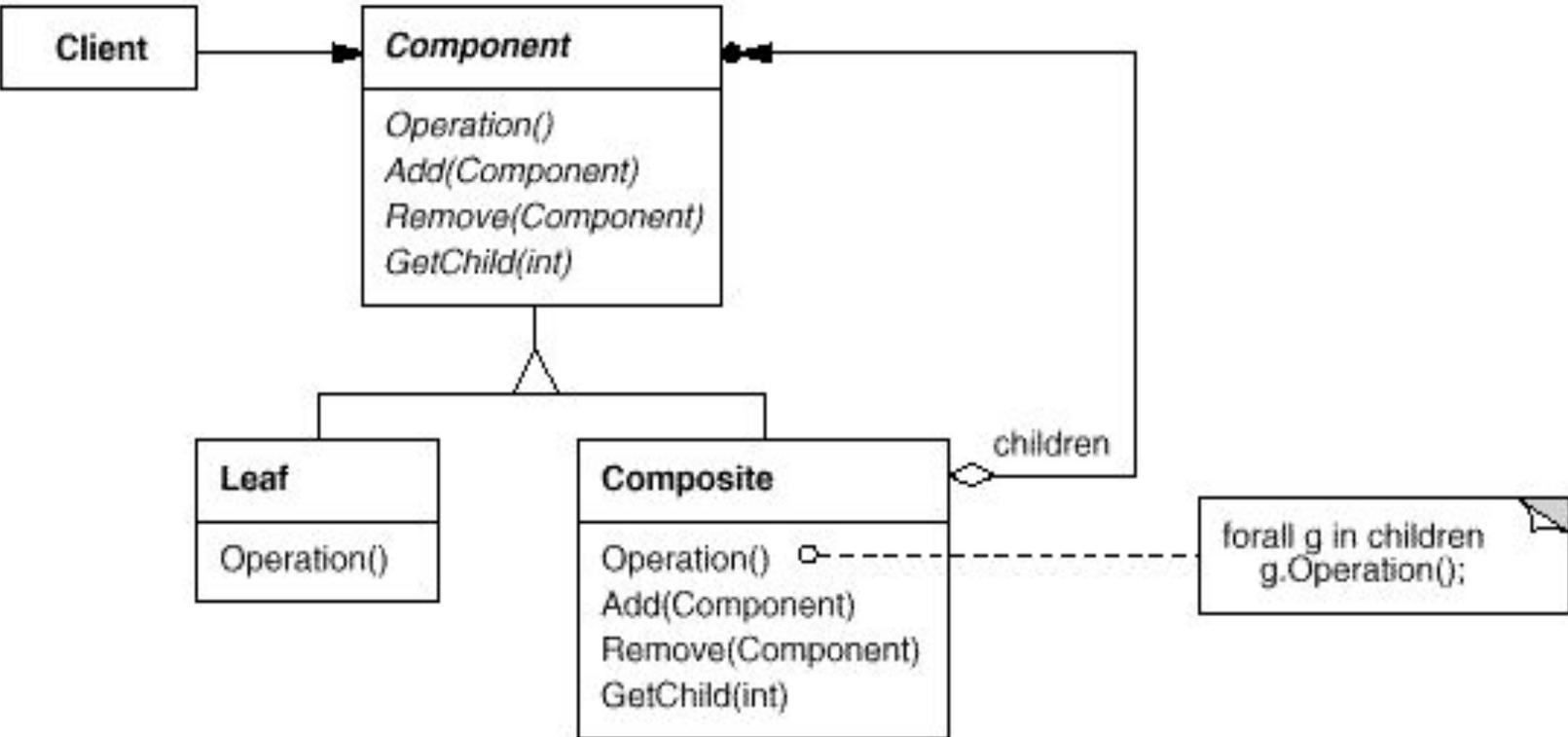
```
public class Graphic
{
    public virtual void Draw(Point point)
    {
        //...
    }
}
public class Image : Graphic
{
    public Image(string fileName)
    {
        //Load image from file
    }
    public override void Draw(Point point){...}
}
```

# Использование прокси

```
public class ImageProxy : Graphic
{
    private Image image;
    private string fileName;
    public void ImageProxy(string fileName)
    {
        this.fileName = fileName;
    }
    public void Draw(Point point)
    {
        if (image == null)
            image = new Image(fileName);
        image.Draw(point);
    }
}
public class TextDocument
{
    //...
    this.Inset(new ImageProxy("TestImage.png"));
}
```

## Composite

- Компонует объекты в древовидные структуры
- Позволяет единообразно трактовать индивидуальные и составные объекты



## Пример кода

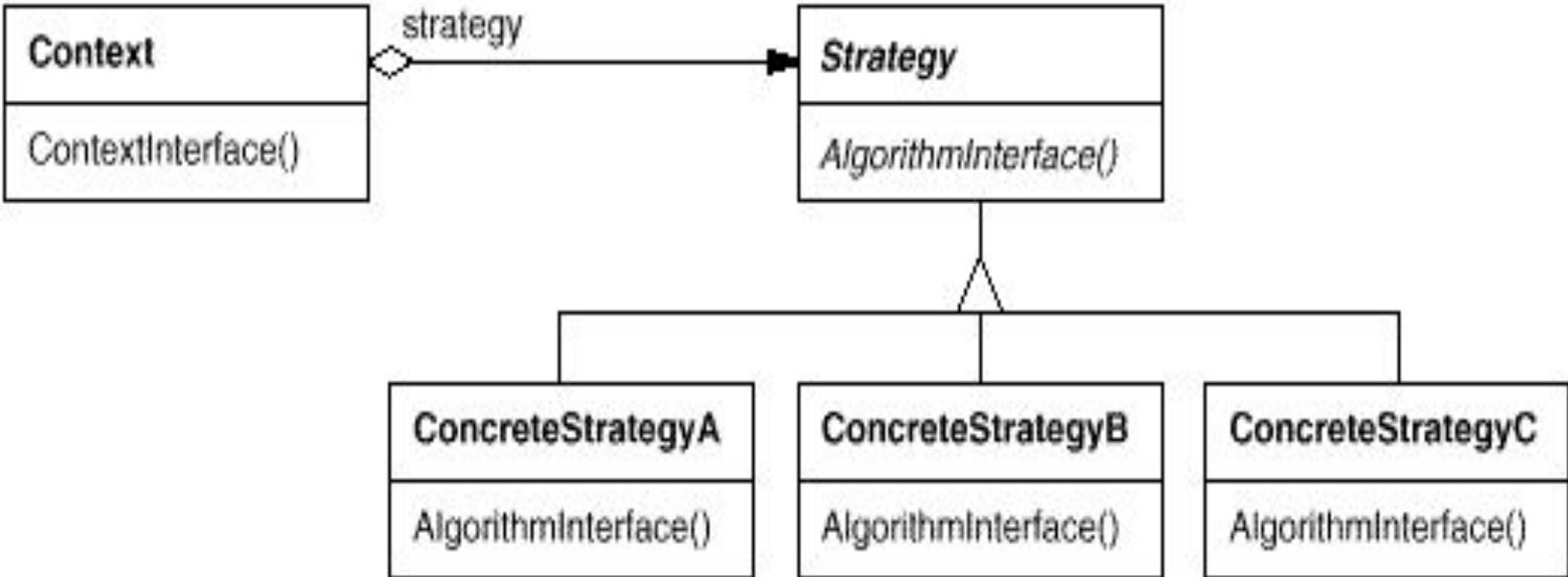
```
public class Equipment
{
    private double netPrice; //Labor, $
    public Equipment GetParent(){...}
    public Equipment[] GetParts(){...}
    public void AddPart(Equipment part){...}
    public void RemovePart(Equipment part){...}
    public virtual double GetPrice()
    {
        double amount = netPrice;
        foreach(Equipment part in GetParts())
        {
            amount += part.GetPrice();
        }
    }
    //...
}
```

## Паттерны поведения

- Инкапсулируют поведение и связи между объектами
- Strategy
- Template method
- Mediator

## Strategy (Policy)

- Определяет семейство алгоритмов и делает их взаимозаменяемыми
- Позволяет изменять алгоритмы независимо от клиентов, их использующих
- Значительно упрощает добавление новых вариантов поведения



## Пример кода

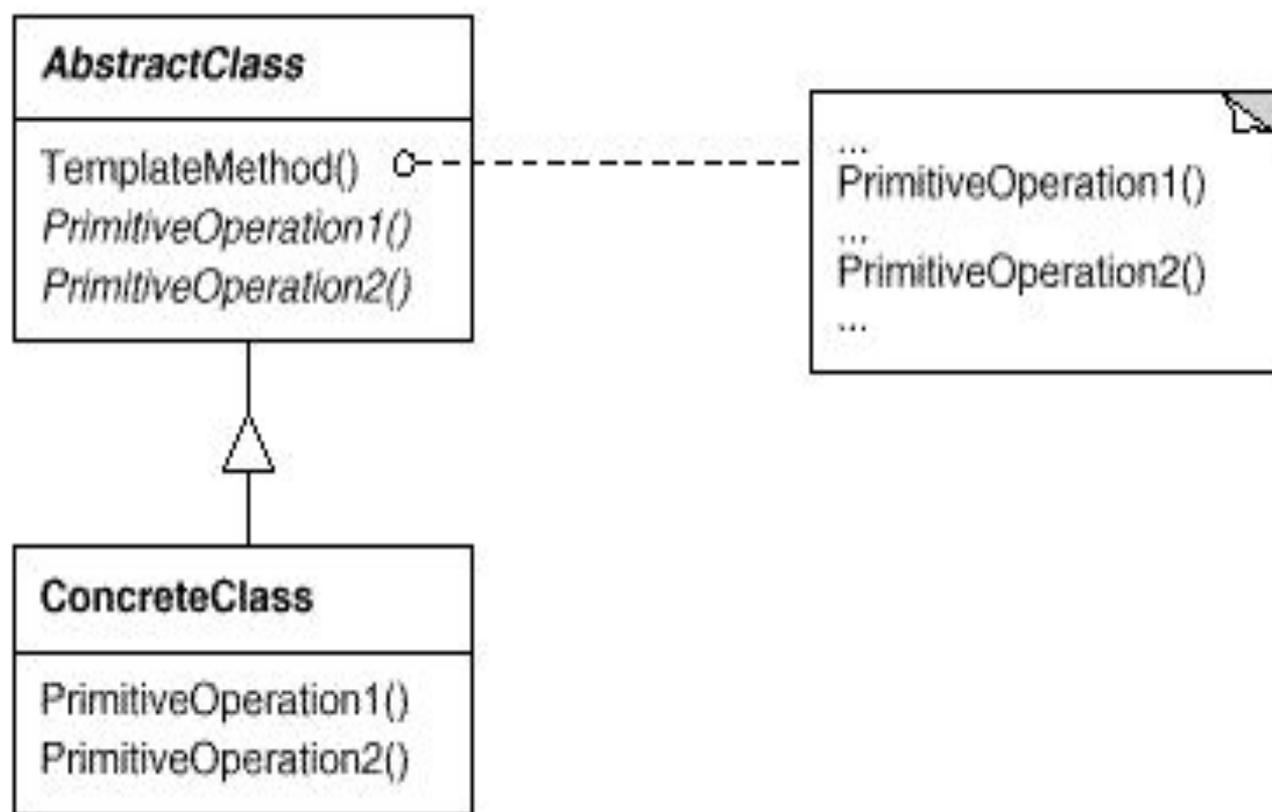
```
public class CalculationStrategy
{
    //...
    public virtual double GetPrice(ArrayList priceItems){...}
}
public class RoundPriceStrategy : CalculationStrategy
{
    int roundTo;
    public RoundPriceStrategy Current(roundTo){... //Singleton}
    public override double GetPrice(ArrayList priceItems){...}
}
public class LockPriceStrategy : CalculationStrategy
{
    double totalAmount;
    public LockPriceStrategy Current(totalAmount){... //Singleton}
    public override double GetPrice(ArrayList priceItems){...}
}
```

## Использование стратегий

```
public class MainForm
{
    //...
    OnLockPriceChecked()
    {
        pricesTable.CalculationStrategy = LockPriceStrategy.Current(totalAmount);
    }
    OnRoundPriceChecked()
    {
        pricesTable.CalculationStrategy = RoundPriceStrategy.Current(roundTo);
    }
}
```

## Template method

- Определяет основу алгоритма
- Позволяет подклассам переопределять отдельные шаги алгоритма, не меняя структуры
- Унифицирует поведение, позволяет легко изменять детали в каждом конкретном случае.



## Пример кода

```
public class BaseForm
{
    public void CloseForm()
    {
        bool sureToClose = ConfirmClose();
        if (!sureToClose) {return;}
        bool saved = Save();
        if (saved) {this.Close();}
    }
    protected virtual bool ConfirmClose() {return true;}
    protected virtual bool Save() {return true;}
}
```

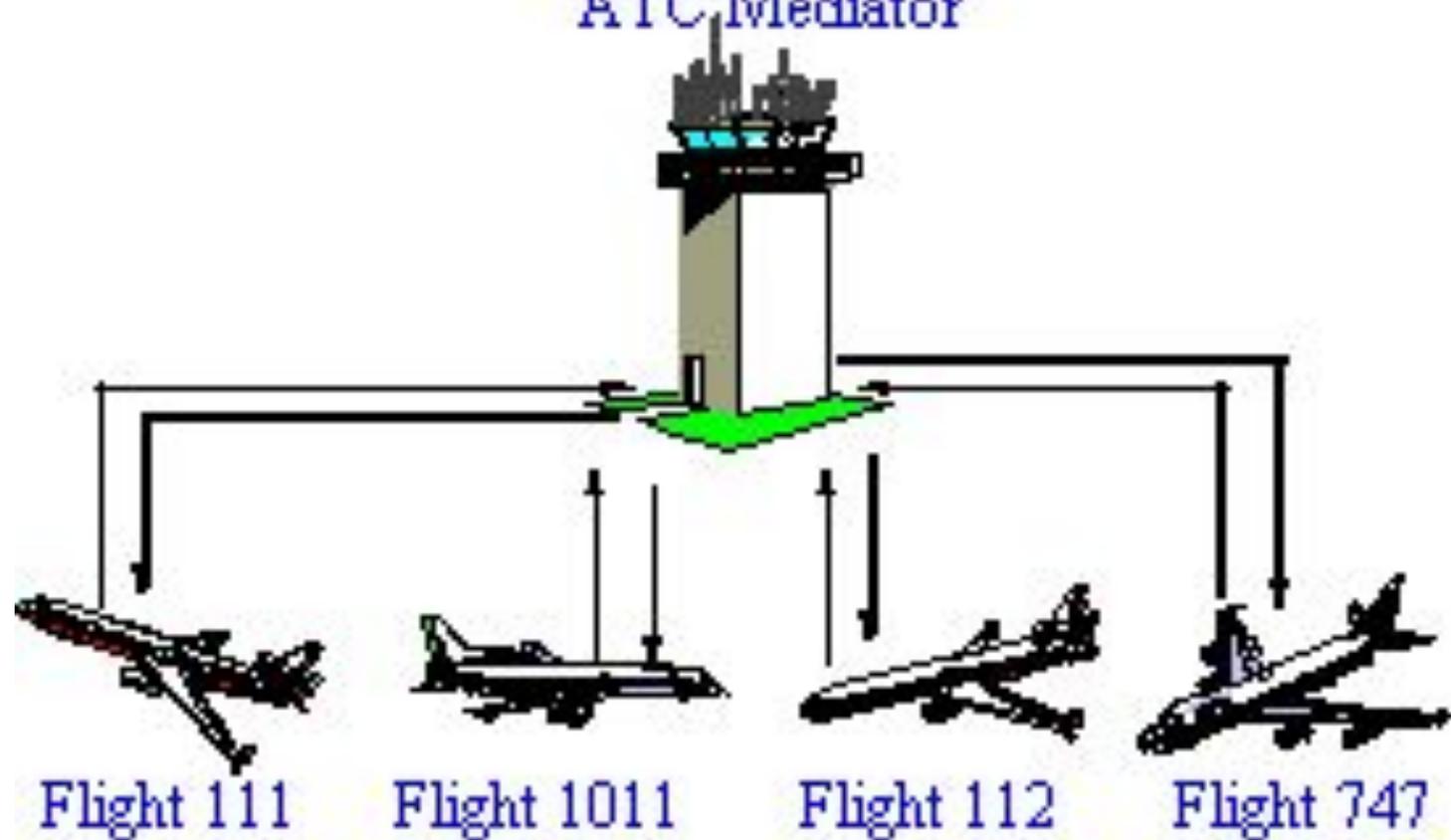
## Продолжение примера

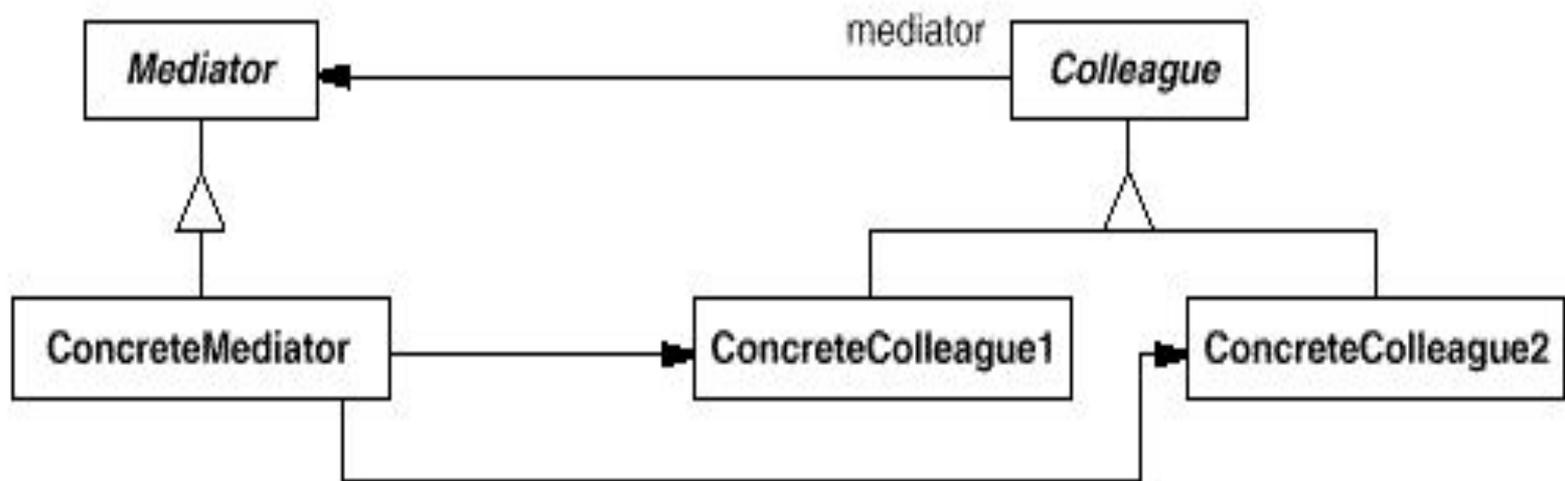
```
public class MainForm : BaseForm
{ //...
    protected override Save()
    {
        bool saved = true;
        if (this.Modified && ShowSaveQuestion())
        {
            try{SaveData();}
            catch(Exception exception)
            {
                ProcessException(exception); //e.g. show message
                saved = false;
            }
        }
        return saved;
    }
}
```

## Mediator

- Обеспечивает слабую связанность системы, избавляет объекты от необходимости явно ссылаться друг на друга
- Вместо многочисленных связей друг на друга все объекты системы хранят ссылку только на посредника

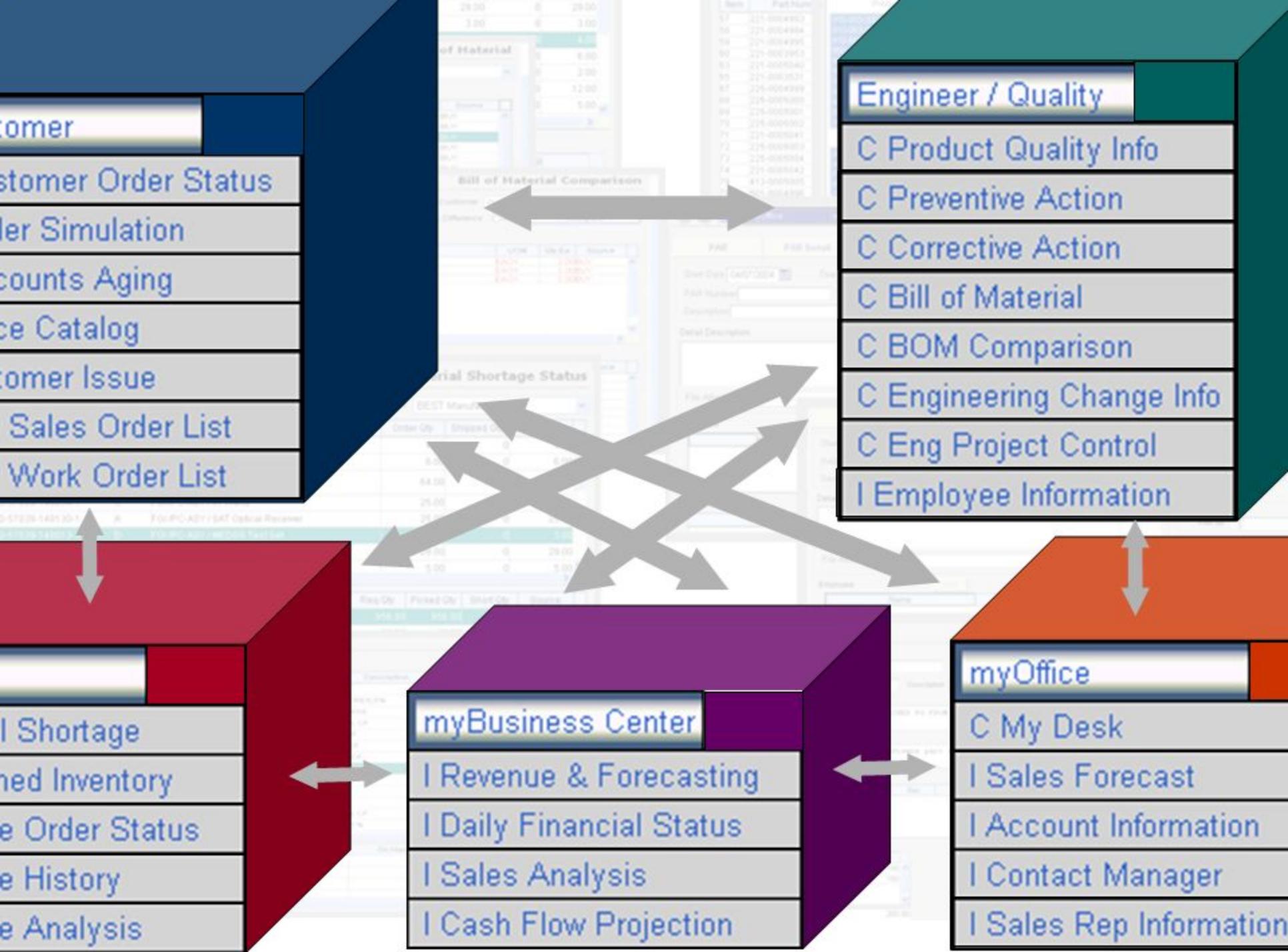
ATC Mediator





## Например

- Взаимодействие модулей системы
  - Нет нужды разбираться в хитросплетениях системы.
  - Упрощен перенос методов между модулями
  - Возможность добавления/удаления модулей

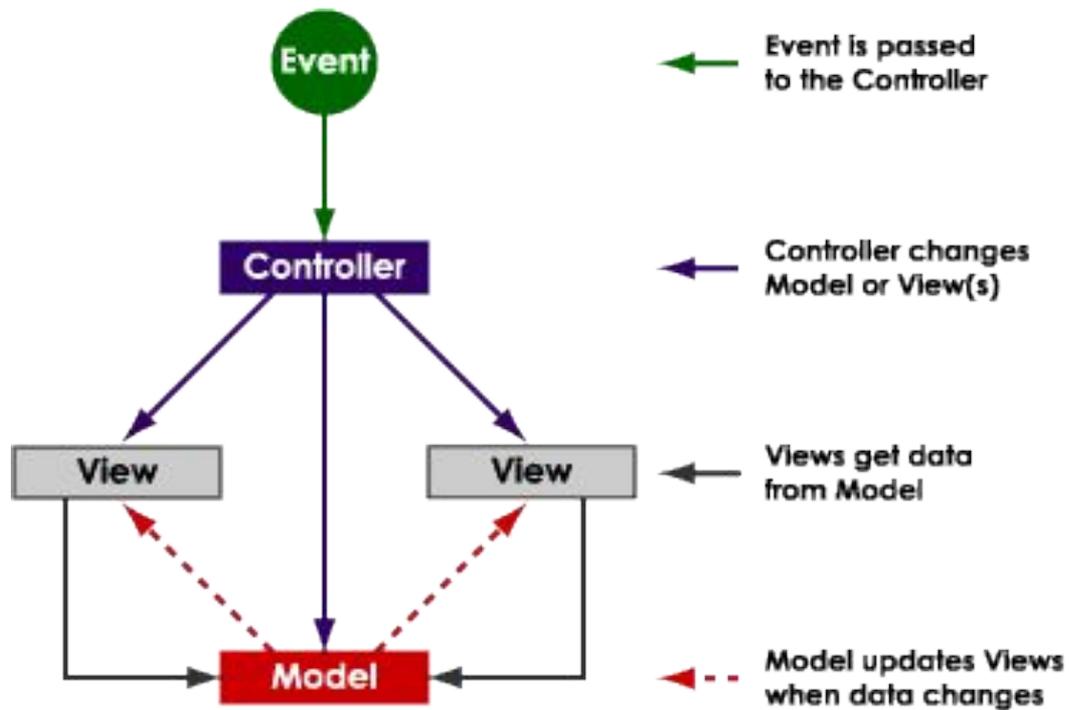


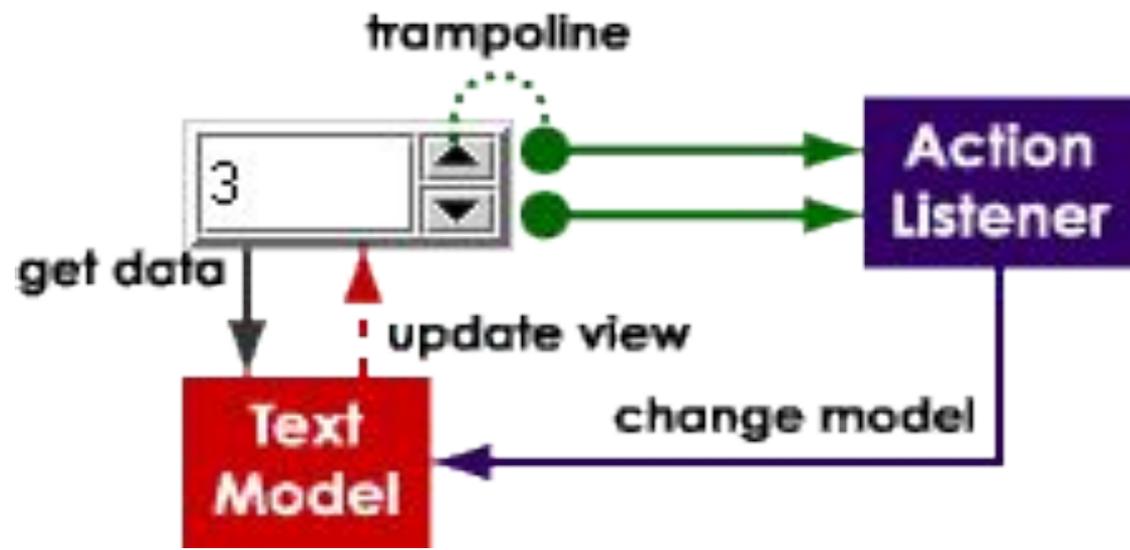
## Результаты

- Выносит взаимодействие между объектами в отдельный класс (легко изменять)
- Преобразует связь много-ко-многим в один-ко-многим (просто понять и поддерживать)

## MVC – Model-View-Controller

- Отделяет вид от модели, устанавливая между ними протокол взаимодействий
- Позволяет организовать различные представления одних и тех же данных





## MVP – Model-View-Presenter

- Смысл – тот же, что и у MVC (отделить данные от представления)
- Ориентируется на современные контролы, которые сами обрабатывают системные события
- Основное отличие – View само ловит события и перенаправляет классу Presenter, который решает, что с этим делать (в MVC события ловит Controller)

## Пример кода

```
public class Presenter
{...
    public Presenter(IView view, IModel model)
    {
        this.view = view;
        this.model = model;
        SubscribeToViewEvents();
        LoadViewFromModel();
    }
}
```

## Результаты

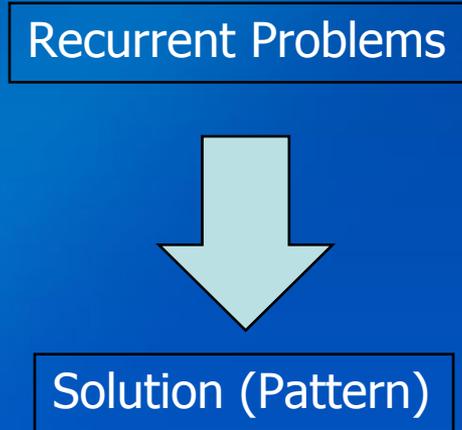
- Вынос поведения в отдельный класс – упрощает понимание кода
- Позволяет вызывать события и моделировать различные ситуации независимо от GUI – полезно как при тестировании, так и при изменении представления

## Antipatterns

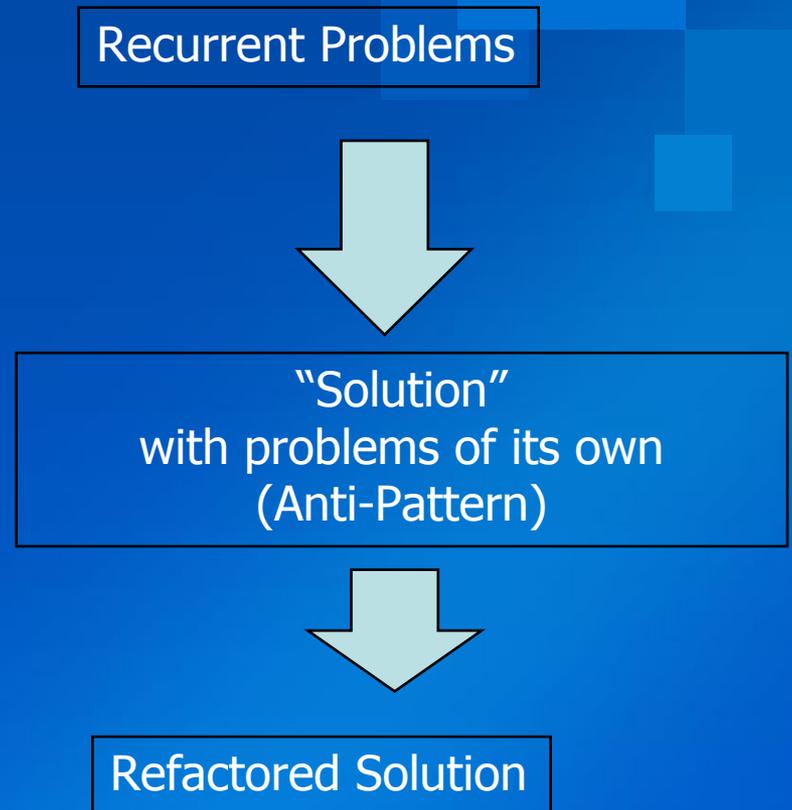
- Описывают наиболее распространенные недостатки (что плохо и почему это плохо)
- Как с этим бороться (refactoring)
- Как выглядит приемлемое решение (когда стоит останавливаться)

# Patterns vs. AntiPatterns

Patterns:



Anti-Patterns:



## The Blob

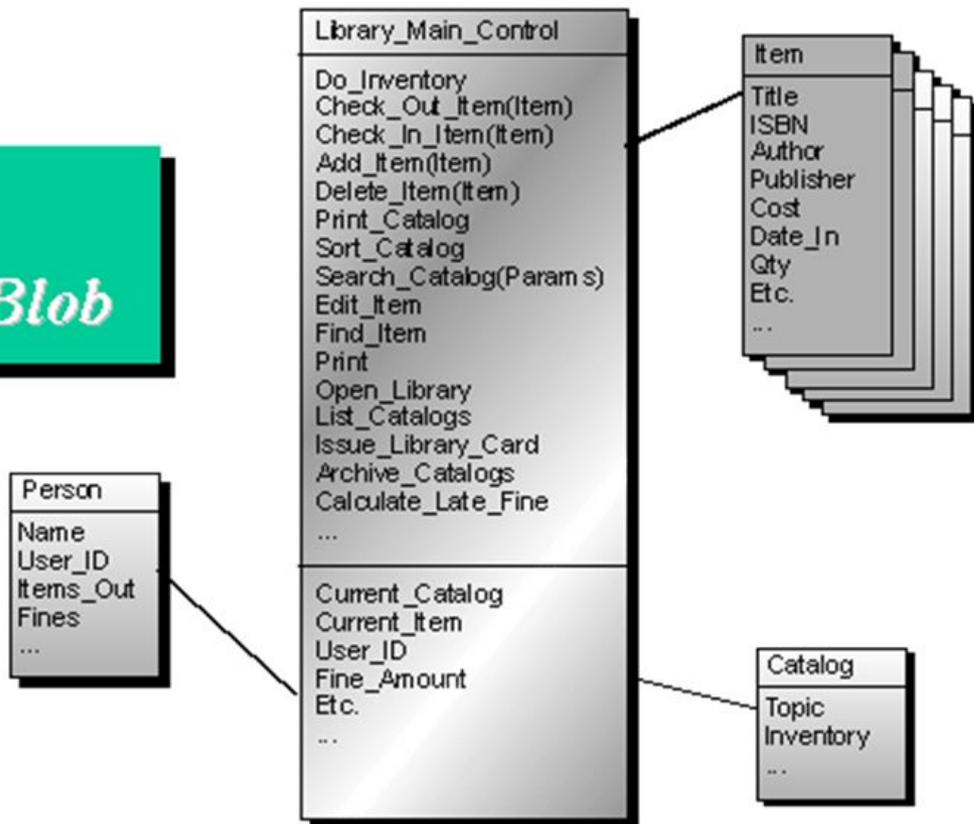
- Огромный класс, содержащий тучу методов, относящихся к логически разным объектам.
- Невозможность повторного использования
- Модификация крайне затруднена, так как приходится следить за консистентностью большого количества данных
- Да и вообще нарушает принципы ООП

*Development AntiPattern:*

# The Blob

*Example:*

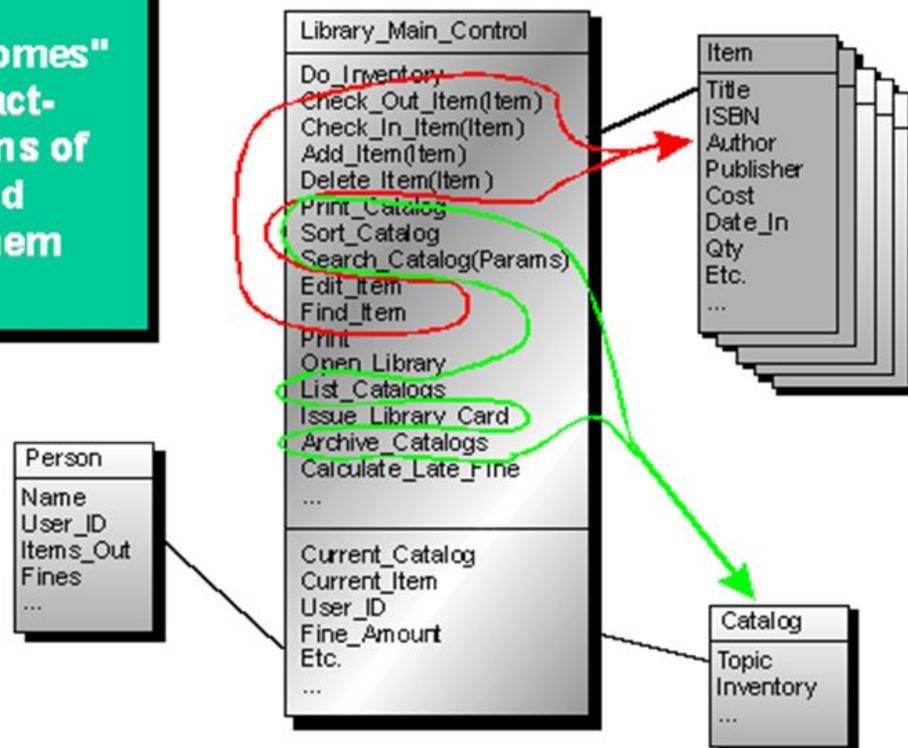
*The Library Blob*



*Development AntiPattern:*

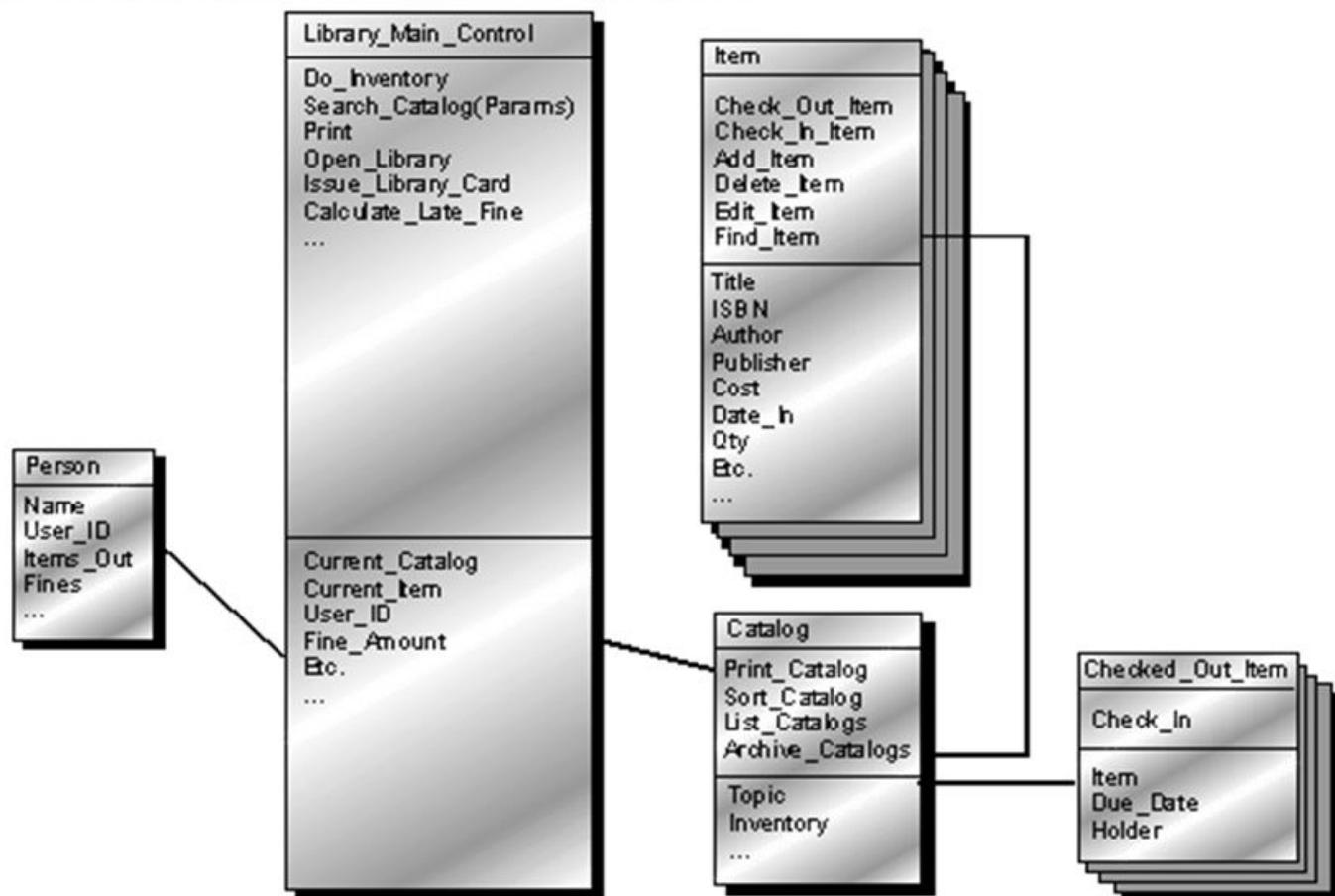
## The Blob - Refactoring

**Step 2:  
Find "natural homes"  
for these contract-  
based collections of  
functionality and  
then migrate them  
there**



*Development AntiPattern:*

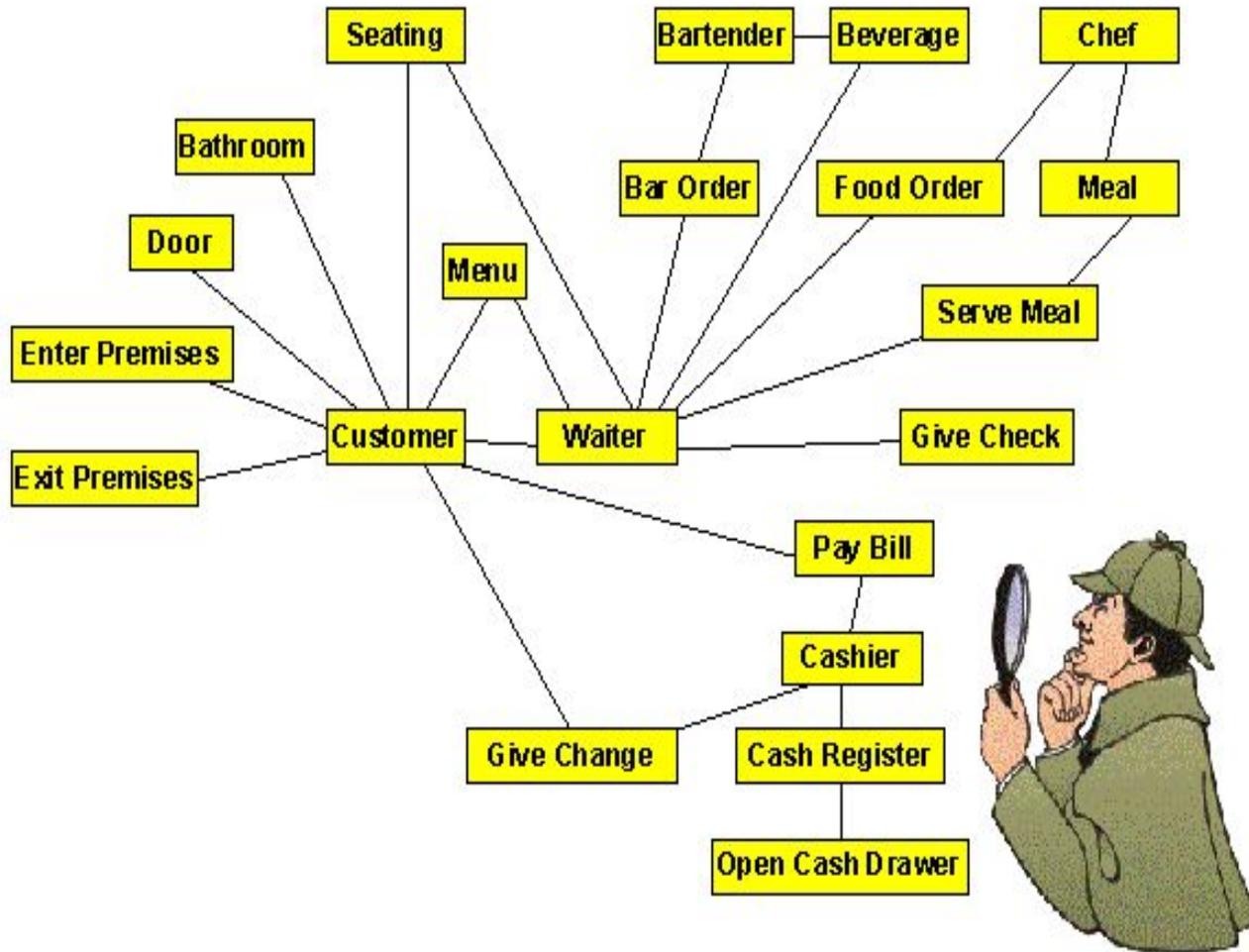
# The Blob Refactored



## Poltergeist

- Много мелких, перекрывающих друг друга классов
- Классы имеют ограниченное использование
- Большое количество связей – усложняет понимание и использование
- Низкая производительность, неожиданные эффекты

## Development AntiPattern: Poltergeists Example



## Исправление

- Удалить ненужные, несущественные классы
- Перегруппировать в более крупные связанные классы

## Lava Flow

- Код, который никто не понимает
- Недокументированный код
- Код, который невозможно тестировать
- Автор кода давно отсутствует на проекте

## Как бороться

- Не допускать (писать понятный, структурированный код и комментарии)
- Не допускать (документировать требования)
- Стиснуть зубы, разобраться и рефакторить

## Golden Hammer

- Использование одного и того же решения всюду, где только можно и даже там, где нельзя
- Игнорирование новых разработок
- Решение:
- Искать наиболее подходящие решения для данного конкретного случая

## Input Kludge

- Возможность ввода данных, несоответствующих контексту
- Возможность сохранения некорректных данных
- Результат – «падения» и непредсказуемая работа
- Решение:
- Проверка всех входных данных на разных уровнях (DB, GUI)

## Refactoring

- Это процесс систематического изменения программной системы, при котором внешнее поведение кода неизменно, но улучшается его внутренняя структура.
- Это непрерывное приведение кода и комментариев в порядок.
- Постоянный процесс, неотделимый от процесса разработки.
- *Мартин Фаулер «Рефакторинг».*

# Bce

[www.ittransition.com](http://www.ittransition.com)

