

Variability patterns:	Extensibility:	Glue Patterns:
<ul style="list-style-type: none"> Template method Template class Strategy Bridge 	<ul style="list-style-type: none"> Observer Decorator 	<ul style="list-style-type: none"> Adapter Facade Mediator Connector

Creational design patterns are **design patterns** that deal with **object creation** mechanisms.

Abstract factory	Класс, который представляет собой интерфейс для создания компонентов системы. Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Builder	Класс, который представляет собой интерфейс для создания сложного объекта. Separate the construction of a complex object from its representation allowing the same construction process to create various representations.
Factory method	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses
Prototype	Это паттерн создания объекта через клонирование другого объекта вместо создания через конструктор. Паттерн используется чтобы избежать дополнительных усилий по созданию объекта стандартным путем (имеется в виду использование ключевого слова 'new')
Singleton	Ensure a class has only one instance, and provide a global point of access to it.

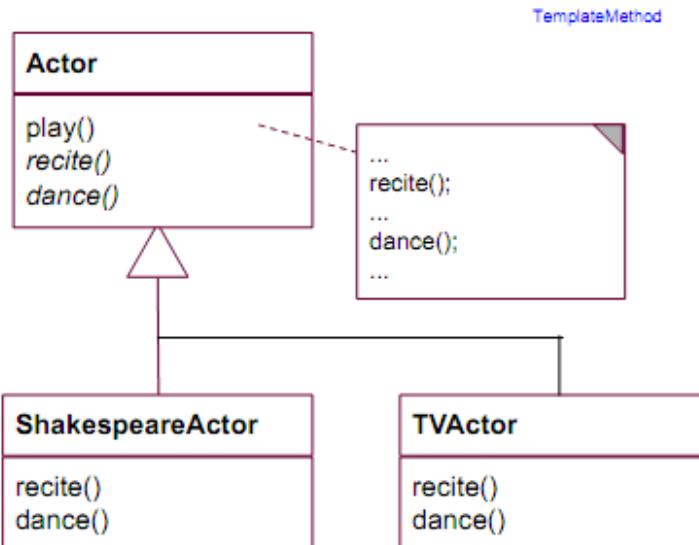
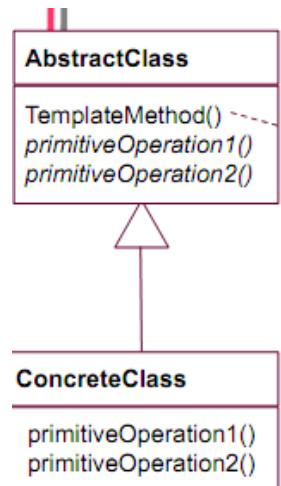
Structural patterns

Adapter	Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces.
Bridge	Decouple an abstraction from its implementation allowing the two to vary independently.
Composite	Объект, который объединяет в себе объекты, подобные ему самому. Composite lets clients treat individual objects and compositions of objects uniformly.
Decorator	Класс, расширяющий функциональность другого класса
Facade	Provide a unified interface to a set of interfaces in a subsystem and makes the subsystem easier to use.
Flyweight	"Приспособленец" эффективно использует большое количество объектов совместно. Примером "Приспособленца" является общая коммутируемая телефонная сеть
Proxy	Объект, который является посредником между двумя другими объектами, и который реализовывает/ограничивает доступ к объекту, к которому обращаются через него

Behavioral patterns

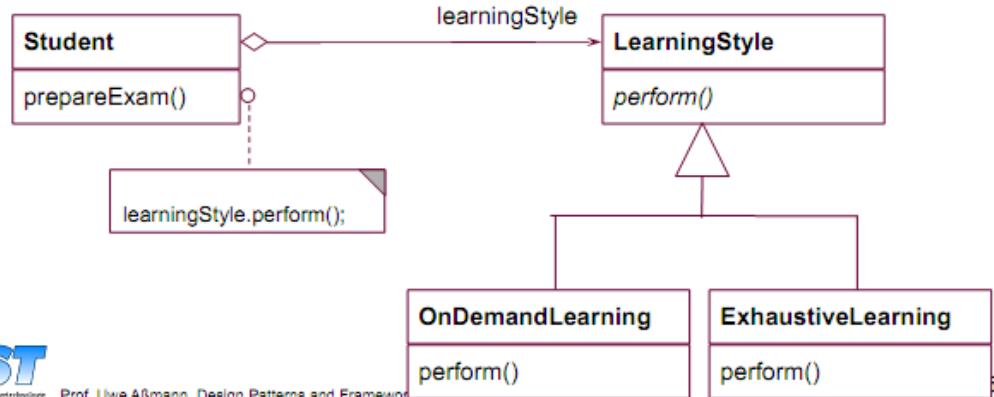
Chain of responsibility	Предназначен для организации в системе уровней ответственности
Command	Encapsulate a request as an object
Iterator	Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из объектов, входящий в состав агрегации
Mediator	Обеспечивает взаимодействие множества объектов, формируя при этом слабую связанность и избавляя объекты от необходимости явно ссылаться друг на друга.
Observer	Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Template method	Define the skeleton (основа) of an algorithm in an operation. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Visitor	Над объектами, входящими в состав структуры, надо выполнять разнообразные, не связанные между собой операции

TemplateMethod Design Pattern



The Objectifier Pattern

- Combined with an abstract class and abstract method
- Clients call objects polymorphically

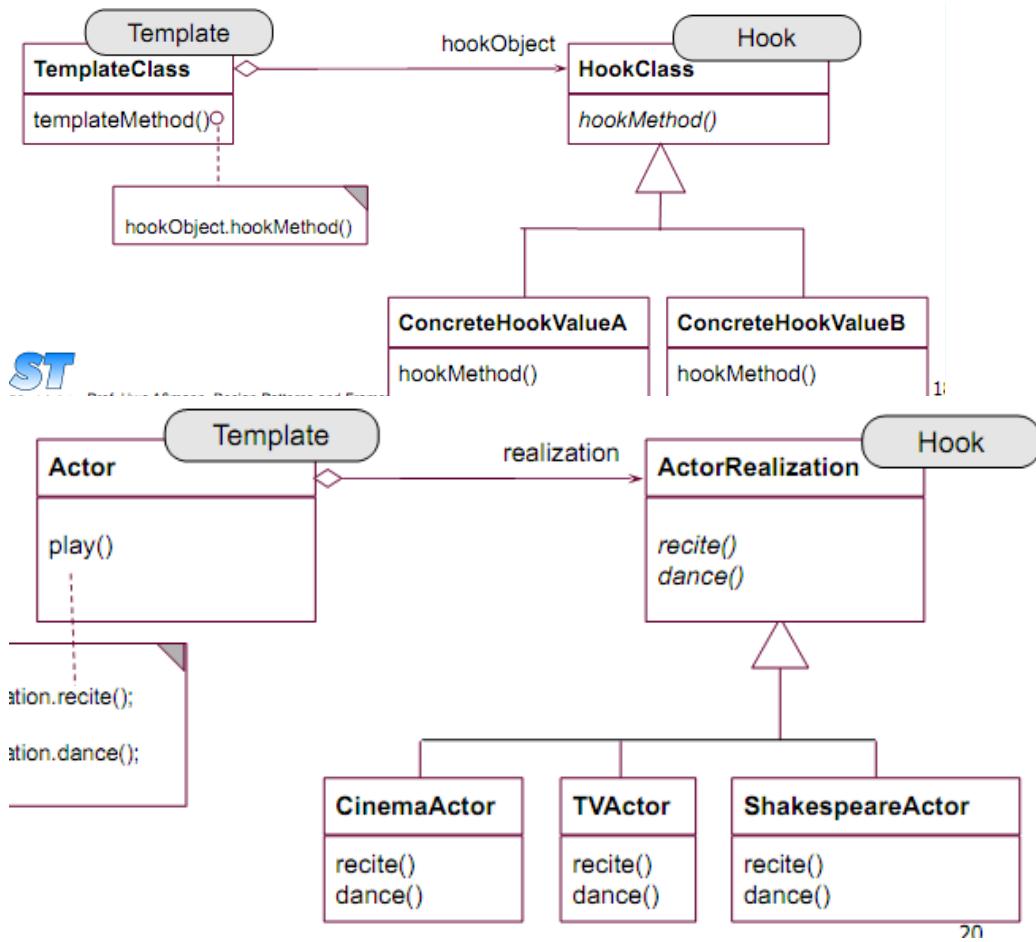


Prof. Uwe Aßmann, Design Patterns and Frameworks

Template Class

Is combined from TemplateMethod and Objectifier

- The template method and the hook method are found in different classes



MVC

Концепция *MVC* позволяет разделить данные, представление и обработку действий пользователя на три отдельных компонента:

- **Модель (англ. Model).** Модель предоставляет знания: данные и методы работы с этими данными, реагирует на запросы, изменяя своё состояние. Не содержит информации, как эти знания можно визуализировать.
- **Представление, вид (англ. View).** Отвечает за отображение информации (визуализацию). Часто в качестве представления выступает **форма (окно)** с графическими элементами.
- **Контроллер (англ. Controller).** Обеспечивает связь между пользователем и системой: контролирует ввод данных пользователем и использует модель и представление для реализации необходимой реакции.

Важно отметить, что как **представление**, так и **контроллер** зависят от **модели**. Однако **модель** не зависит ни от **представления**, ни от **контроллера**. Тем самым достигается назначение такого разделения: оно позволяет строить **модель независимо от визуального представления**, а также создавать несколько различных **представлений** для одной **модели**.

При обработке реакции пользователя вид выбирает, в зависимости от нужной реакции, нужный контроллер, который обеспечит ту или иную связь с моделью. Для этого используется шаблон проектирования [стратегия](#)

А для возможности однотипного обращения с подобъектами сложно-составного иерархического вида может использоваться шаблон проектирования [Компоновщик](#).

Delegation: When my object uses another object's functionality as is without changing it.

An **association** is when an object has a reference to another object that's not by definition a part of the first object, but instead performs some sort of function that the first object makes use of.

For instance, an instance of the class Person can have a reference to an instance of Car. The person is associated to that

car, but the car isn't part of the person. As a matter of fact, there might be another Person who also has a reference to the same car.

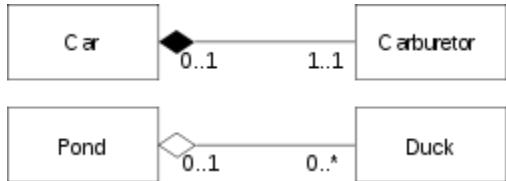
Aggregation is when an object has a reference to another object that's inherently a part of it. A Car can have references to its Engine; the engine is part of the car. However, once the car is scrapped, they can salvage the engine and put it in another Car.

Composition is when an object has a reference to another object, that's inherently a part of it, and only it. When the first object is 'scrapped', so are the objects it is composed of. Let's just say that the Doors of a Car can not be reused. They are constructed when the Car is constructed, and when the Car is demolished, so are the Doors.

UML notation

In [UML](#), composition is depicted as a **filled diamond** and a solid line. It always implies a multiplicity of 1 or 0..1, as no more than one object at a time can have lifetime responsibility for another object.

The more general form, [aggregation](#), is depicted as an **unfilled diamond** and a solid line.



1. **Composition:** House contains one or more rooms. Room's lifetime is controlled by House as Room will not exist without House.
2. **Aggregation:** Toy house built from blocks. You can disassemble it but blocks will remain.
3. **Delegation:** Your boss asked you to get him a coffee; you've had an intern do it for you instead.

агрегирование – включение одного объекта в состав другого.

В Java есть два типа классов – способные содержать реализацию, и неспособные на это. Вторые называются интерфейсами, хотя по сути – это полностью абстрактные классы.

Aggregation(ромбик) вхождение, внутри одного класса есть другой класс (город- здания, деревья, ...), ассоциация – link. Закрашенный ромбик – композиция, внутренний элемент зависит от внешнего.

Example of aggregation

The CRM system has a database of customers and a separate database that holds all addresses within a geographic area. Aggregation would make sense in this situation, as a Customer 'has-a' Address. It wouldn't make sense to say that an Address is 'part-of' the Customer, because it isn't.

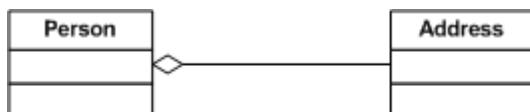
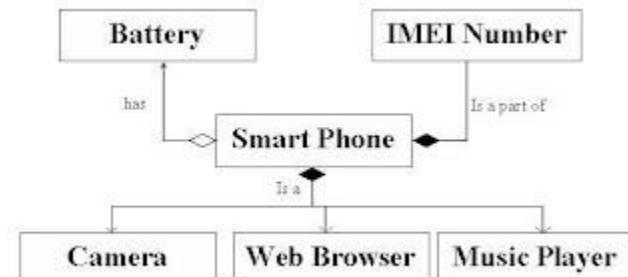


Figure 2 - Aggregation



The Battery can exist without the Smart Phone and it has a meaning without the Smart Phone. On the other hand, the IMEI Number is a part of the Smart Phone and the existence of the IMEI number completely depends on the existence of the Smart Phone. The IMEI Number has no meaning without the Smart Phone. While the relation between IMEI Number and the Smart Phone explains the 'is a part of' type of composition, the other relation explains the 'is a' type of composition. The Smart Phone, in this case, plays roles at different times. The Smart Phone is a Camera. The Smart Phone is a Web Browser. The Smart Phone is a Music Player. The Music Player role of the Smart Phone cannot exist when the Smart Phone itself is not available.

Good Questions for Pattern Mining (Vlissides)

- ▶ Why did you design this way? [Rationale, Motivation]
- ▶ Is what seems to be complexity here really worthwhile? [Consequences]
- ▶ What are your assumptions? [Rationale]
- ▶ Why are your assumptions realistic?
- ▶ What happens 6 months from now when I need new feature F? [extensibility, variability]

- ▶ Hint: Ask these questions yourself, if you write a pattern

Success Criteria for Single Patterns

- ▶ They must be compact
- ▶ They must be mined from working designs
- ▶ They must be mined from "best practice"
- ▶ They need not be object-oriented

Delegation

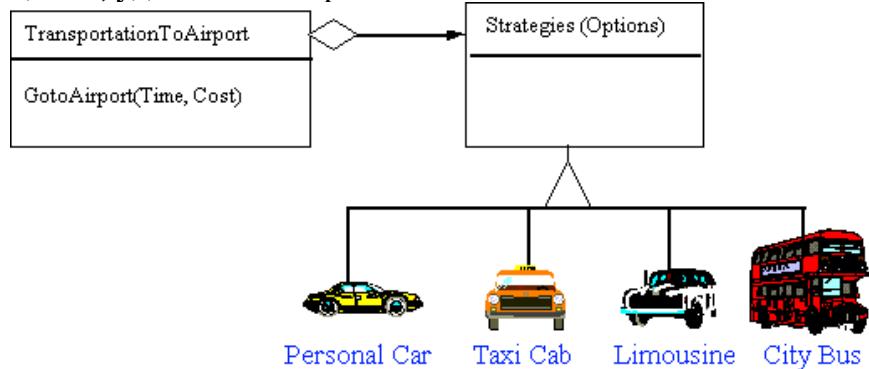
In object-oriented programming it is used to describe the situation where one object defers a task to another object, known as the delegate. Делегирование означает ситуацию, когда объект для предоставления определённого набора функциональности полагается на другой объект.

Delegation allows one object to override the behavior of another. The original object *a* (analogous to the base class behaviors) can *delegate* some of its methods to another object *b* (analogous to the derived class behaviors). If *a* delegates its *foo* method to the *bar* method of *b*, then any invocation of *foo* on *a* will cause *b*'s *bar* method to execute. However, *bar* executes in the context of the *a* object

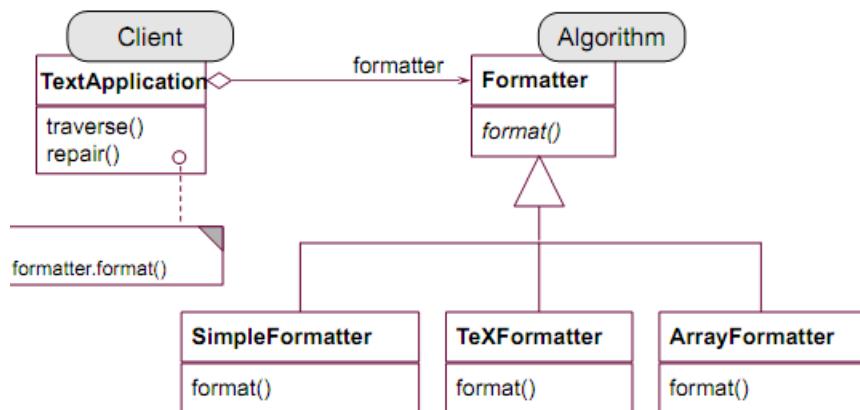
Пример "СТРАТЕГИИ"

"Стратегия" задает набор алгоритмов, которые могут быть использованы как взаимозаменяемые. Вид доставки в аэропорт является примером "Стратегии". Существует несколько вариантов: на собственном автомобиле, вызвать такси, на межаэропортном спецтранспорте, на городском автобусе или лимузине. До некоторых аэропортов также можно добраться и на метро или даже на вертолете. Любой из этих видов доставки путешественников в аэропорт может стать

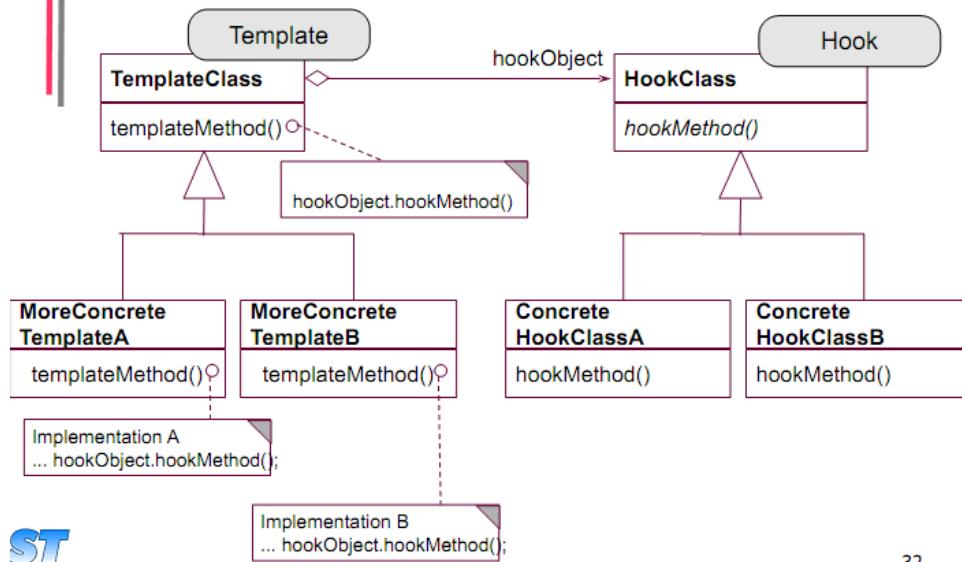
взаимозаменяемым. Пассажир должен выбрать "Стратегию", основываясь на компромиссе между ценой, удобством и временем.



Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

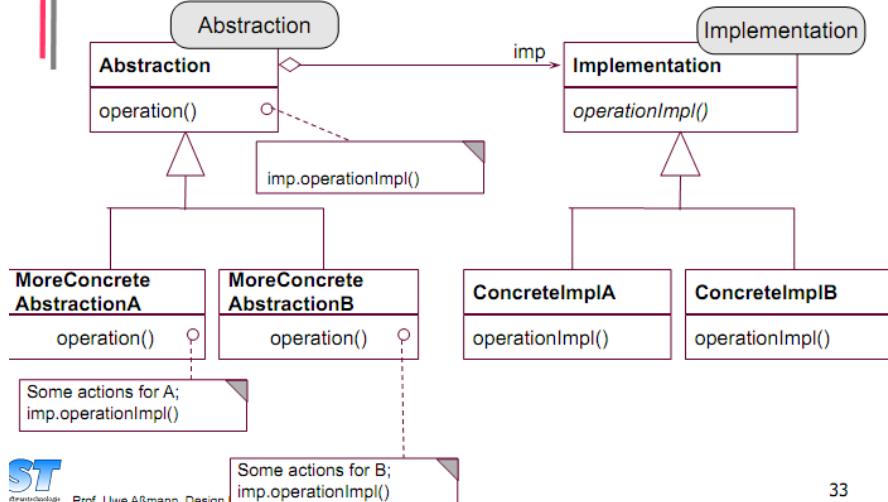


Remember Dimensional Class Hierarchy (Bridge with Template/Hook Constraint)



Bridge

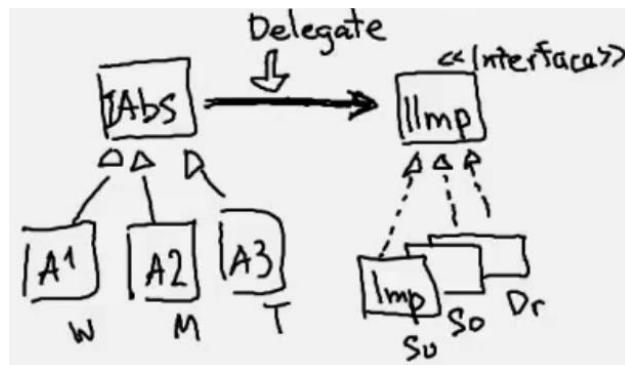
- Different incentive (Abstraction/Implementation)



33

Хороший пример в

http://www.youtube.com/watch?v=zI4icUNXgWk&list=UUj1txX_A5byHhrPKuGBKZIA&index=6&feature=plcp



The **Bridge** pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the *Bridge*. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, a simple two position switch, or a variety of dimmer switches.

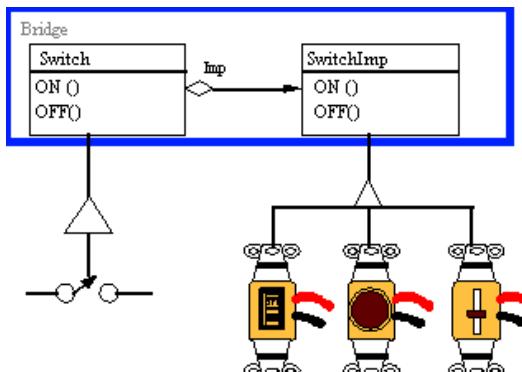
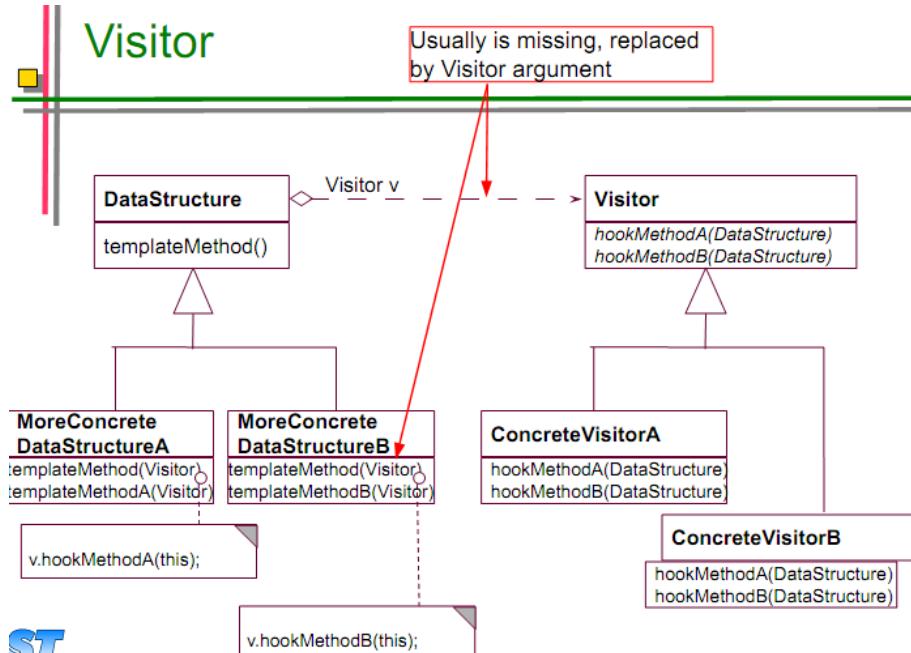


Figure 7: Object Diagram for **Bridge** using Electrical Switch Example

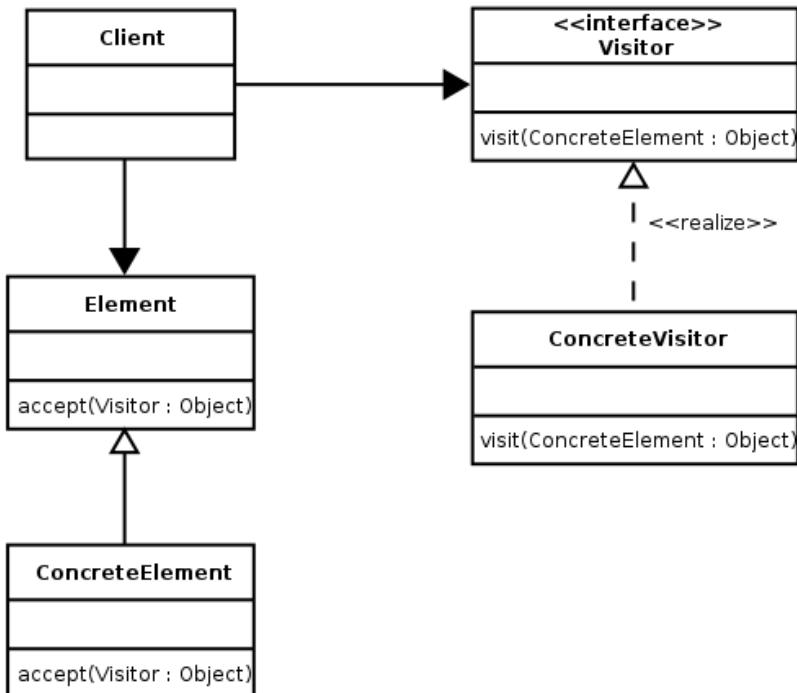
Нужно посмотреть недостатки из слайдов

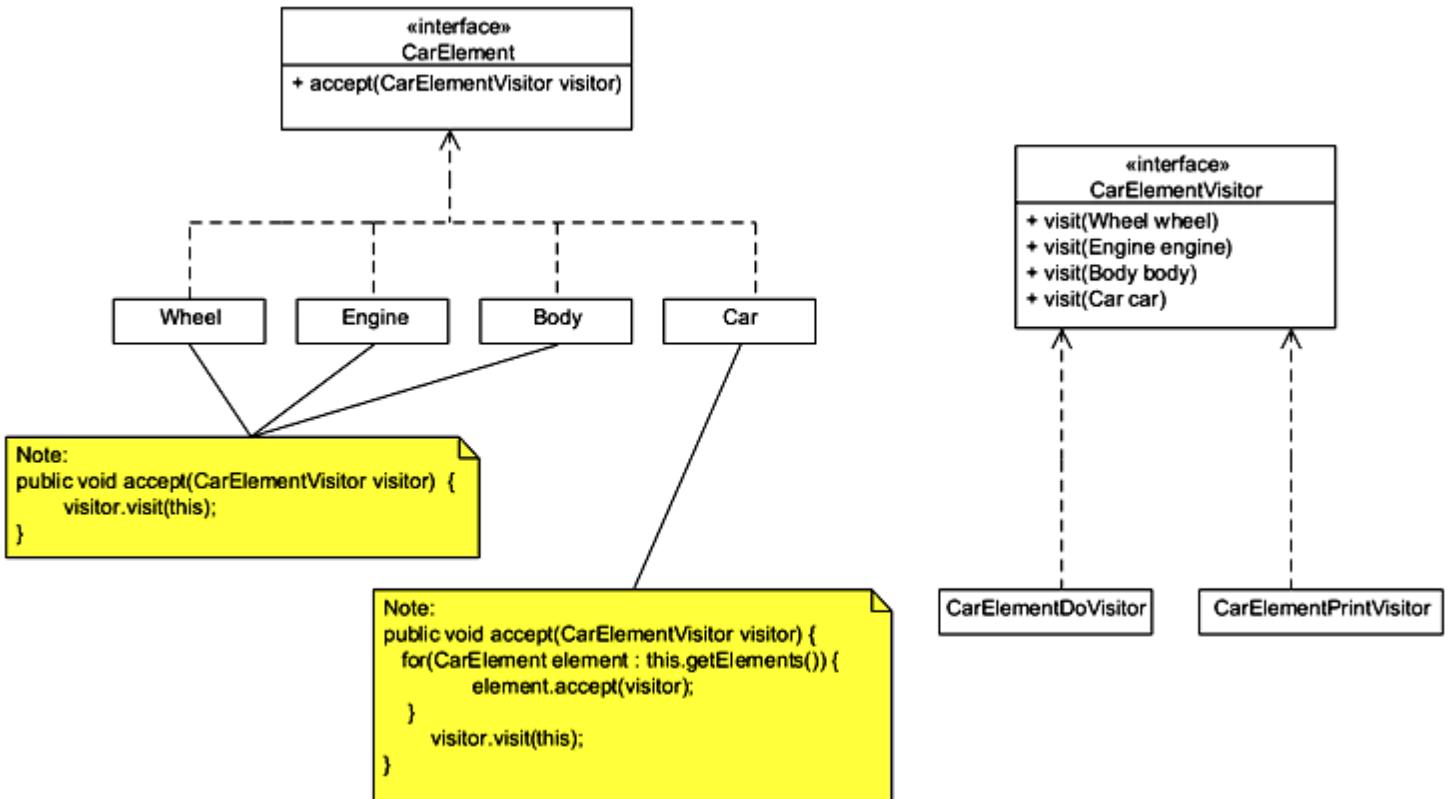
VISITOR



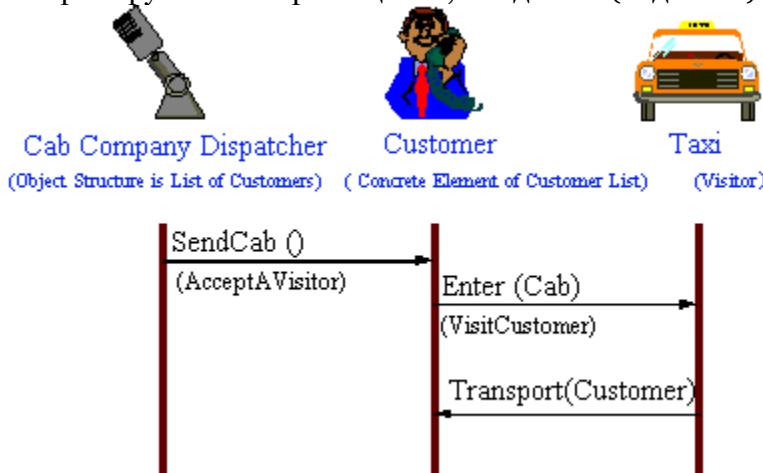
ST

Над объектами, входящими в состав структуры, надо выполнять разнообразные, не связанные между собой операции и вы не хотите «засорять» классы такими операциями. Посетитель позволяет объединить родственные операции, поместив их в один класс. Если структура объектов является общей для нескольких приложений, то паттерн посетитель позволит в каждое приложение включить только относящиеся к нему операции



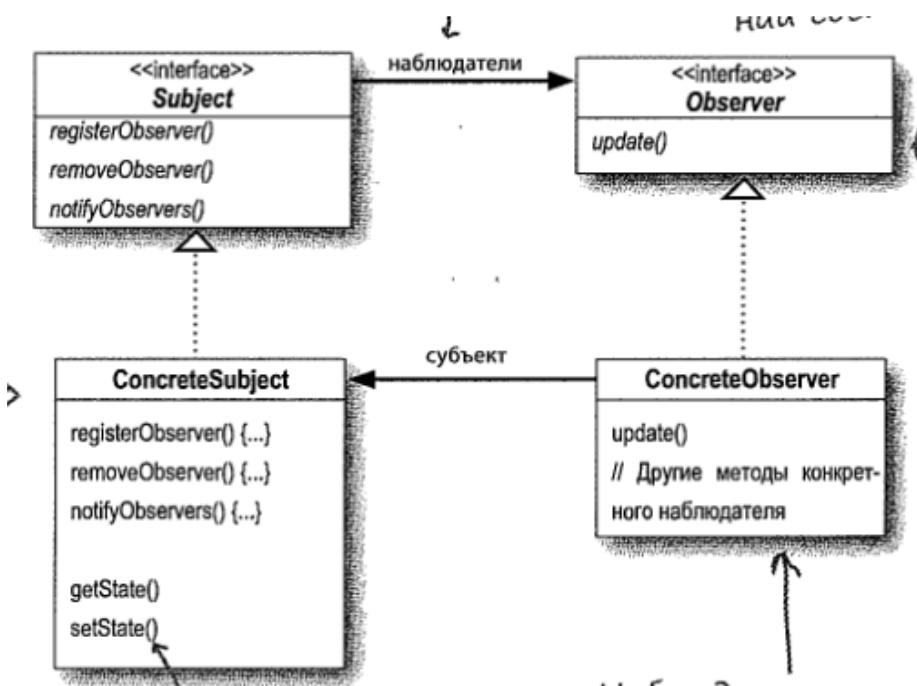
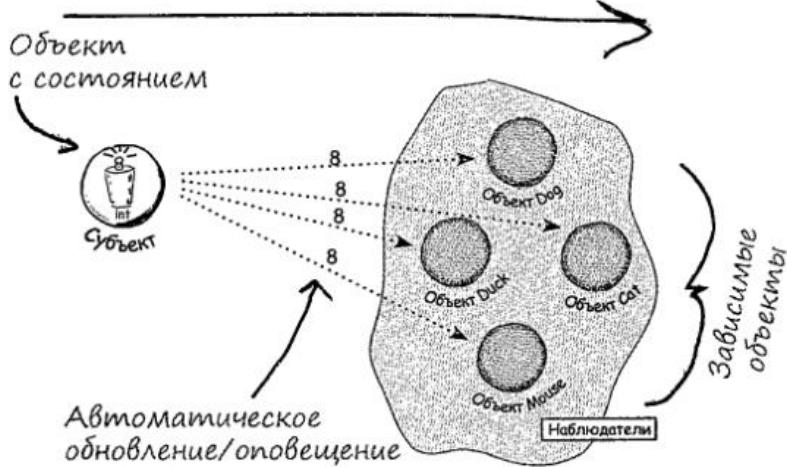


Шаблон "Посетитель" представляет собой операцию, исполняемую на элементах объектной структуры, без изменения классов, с которыми он работает. Этот шаблон можно обнаружить в работе службы такси. Когда кто-либо звонит в службу такси, он становится частью списка заказчиков службы. Затем служба направляет машину заказчику (принимает посетителя). Садясь в такси, или становясь его Посетителем, заказчик уже не контролирует свое перемещение, это делает (водитель) такси.



Паттерн Наблюдатель определяет отношение «один-ко-многим» между объектами таким образом, что при изменении состояния одного объекта происходит автоматическое оповещение и обновление всех зависимых объектов.

ОТНОШЕНИЕ «ОДИН-КО-МНОГИМ»



Subject:

- Неизвестно конкретные классы наблюдателей, только известно об интерфейсе, который реализовывается наблюдателями
- предоставляет интерфейс для добавления/удаления наблюдателей
- посылает notify всем наблюдателям, если произошли какие то изменения в subject
 1. ConcreateObserver изменяет состояние ConcreateSubject
 2. ConcreateSubject исполняет notify() сообщая всем остальным наблюдателям о произошедшем изменении в ConcreateSubject
 3. все остальные наблюдатели изменяют свое состояние

Наблюдатели не имеют связи друг с другом. Согласованность между ними происходит за счет субъекта. То есть, при возникновении каких либо изменений сам субъект извещает наблюдателей о произошедшем изменении. Иначе, если имеется связь между наблюдателями, добавление нового наблюдателя затрудняется (придется установить связь всех наблюдателей с новым наблюдателем)

Наблюдение более чем за одним субъектом

- Наблюдатель иногда может зависеть от более чем одного субъекта
 - Окно визуализации (наблюдатель) игрового поля в стратегических играх зависит от состояния нескольких боевых единиц (субъектов)
- В этом случае в метод Update передается еще и ссылка на субъект, приславший уведомление
 - Субъект может передать себя в качестве параметра операции Update

Важно!

- Модель «вытягивания» (pull model)
 - Субъект посылает минимум информации об изменении, наблюдатели запрашивают детали позднее
 - Недостаток – наблюдателям приходится выяснять без помощи субъекта, что изменилось
- Модель «проталкивания» (push model)
 - Субъект посылает детальную информацию об изменении, независимо нужно это наблюдателям или нет
 - Недостаток – снижение повторного использования кода, т.к. классы Subject обладают некоторой информацией об Observer, что не всегда может быть верным

DECORATOR p120

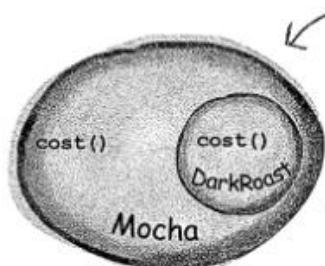
Паттерн Декоратор динамически наделяет объект новыми возможностями и является гибкой альтернативой субклассированию в области расширения функциональности.

1 Начинаем с объекта DarkRoast.



Напоминаем, что DarkRoast наследует от Beverage и содержит метод cost() для вычисления стоимости напитка.

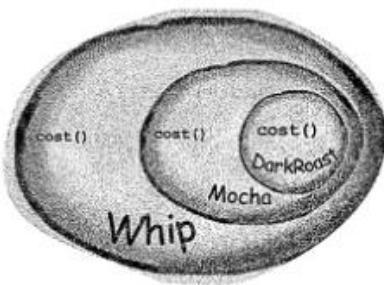
2 Клиент заказывает шоколад, поэтому мы создаем объект Mocha и «заворачиваем» в него DarkRoast.



Объект Mocha является декоратором. Его тип повторяет тип декорируемого объекта — в данном случае Beverage.

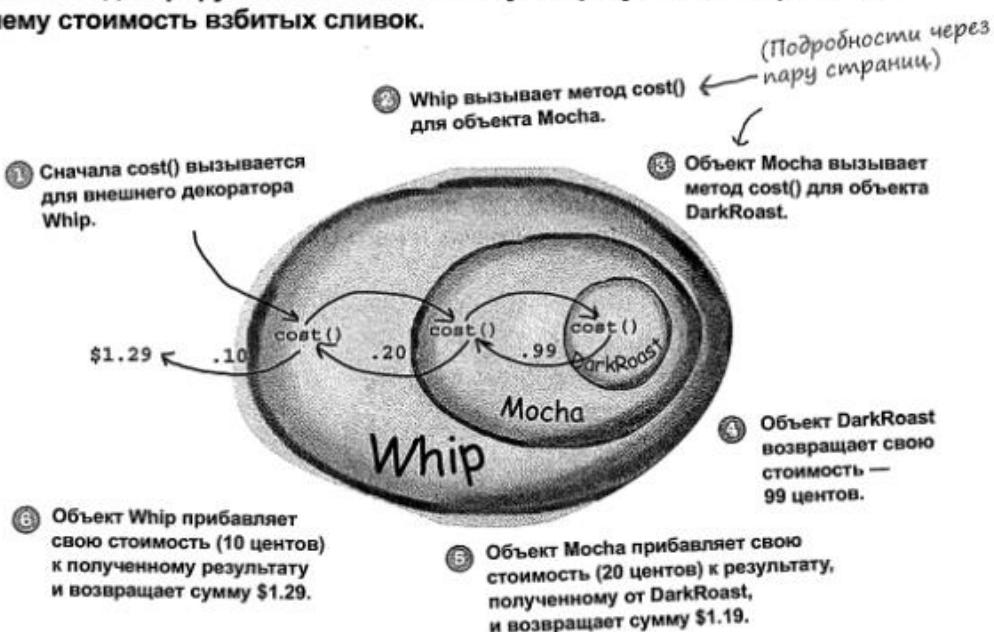
Объект Mocha тоже содержит метод cost(), а благодаря полиморфизму он может интерпретироваться как Beverage (так как Mocha является субклассом Beverage).

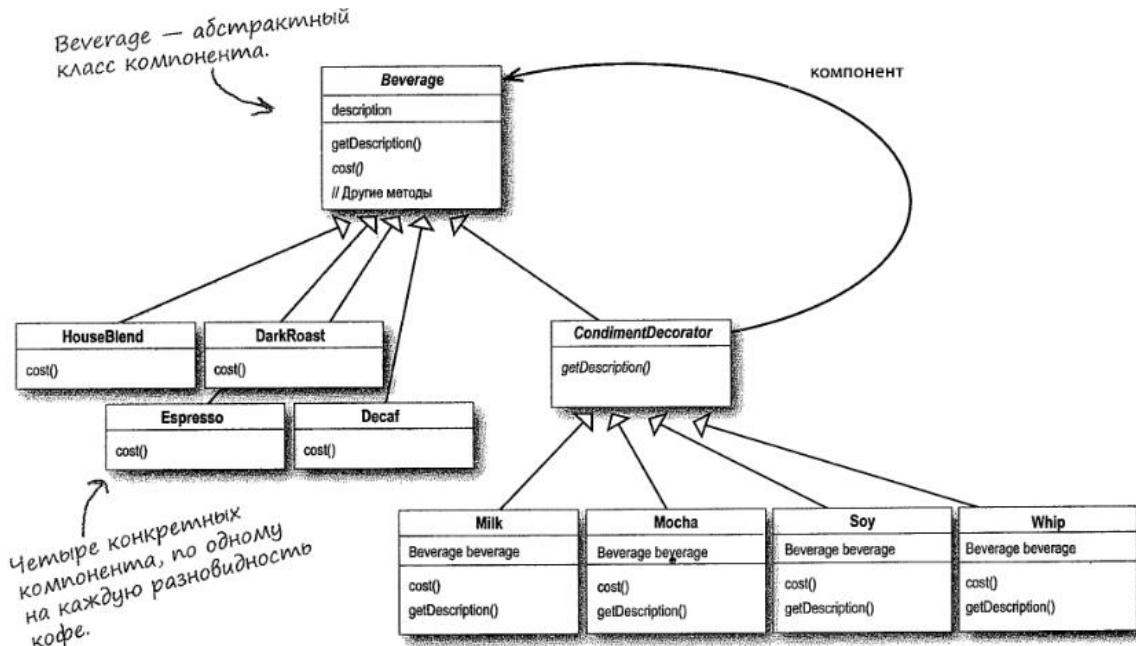
3 Клиент также хочет взбитые сливки, поэтому мы создаем объект Whip и «заворачиваем» в него Mocha.



Декоратор Whip также повторяет тип DarkRoast и содержит метод cost().

4 Пришло время вычислить общую стоимость напитка. Для этого мы вызываем метод cost() внешнего декоратора Whip, а последний делегирует вычисление декорируемым объектам. Получив результат, он прибавляет к нему стоимость взбитых сливок.

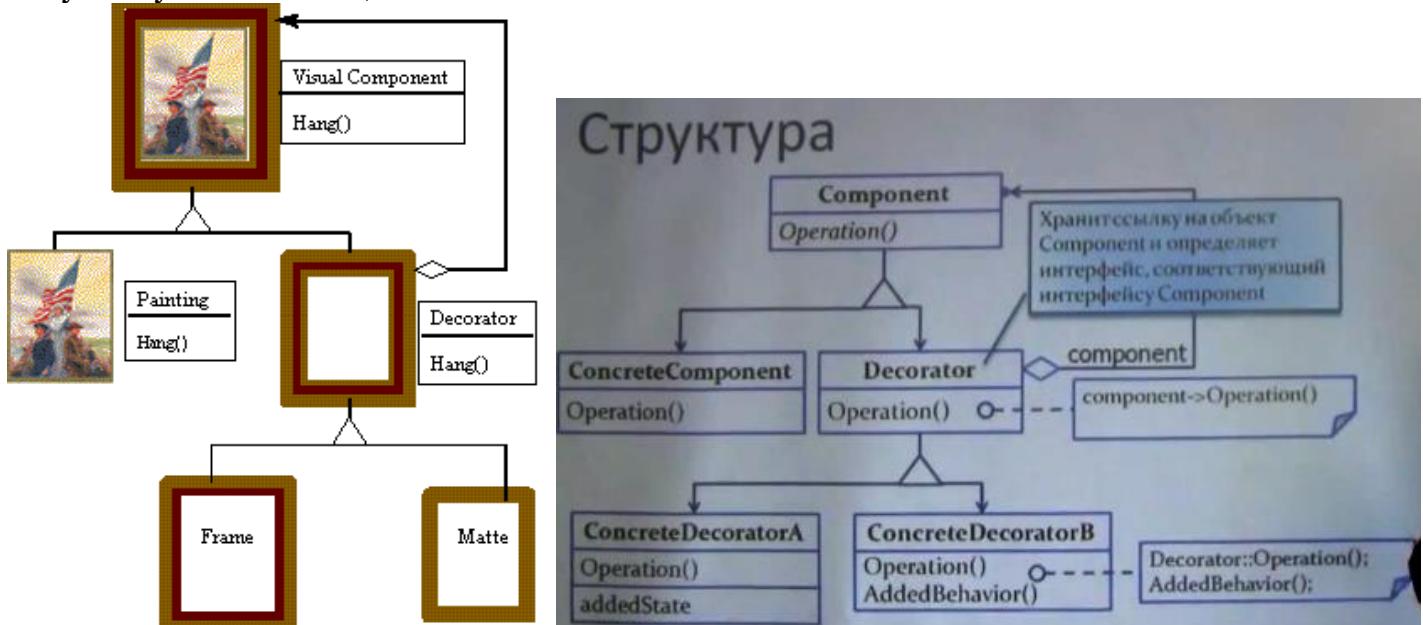




Декораторы представляют собой дополнения к кофе.
Обратите внимание: они должны реализовать не только `cost()`, но и `get>Description()`. Вскоре мы увидим, почему это необходимо...

В Beverage beverage хранится декорируемый объект, то есть Beverage
Декоратор является гибкой алтернативой порождения подклассов для расширения функциональности.

"Декоратор" динамически подключает к объектам дополнительные функции. Хотя картины могут быть повешены на стену как в рамке, так и без, рамками пользуются часто, и как раз за рамку картину вешают на стену. Перед тем как вешать, картина может быть украшена багетом или установлены в рамку так, чтобы и картина, и багет, и рамка образовали единую визуальную композицию.



Достоинство: возможность добавления новых возможностей во время выполнения программы (at runtime)

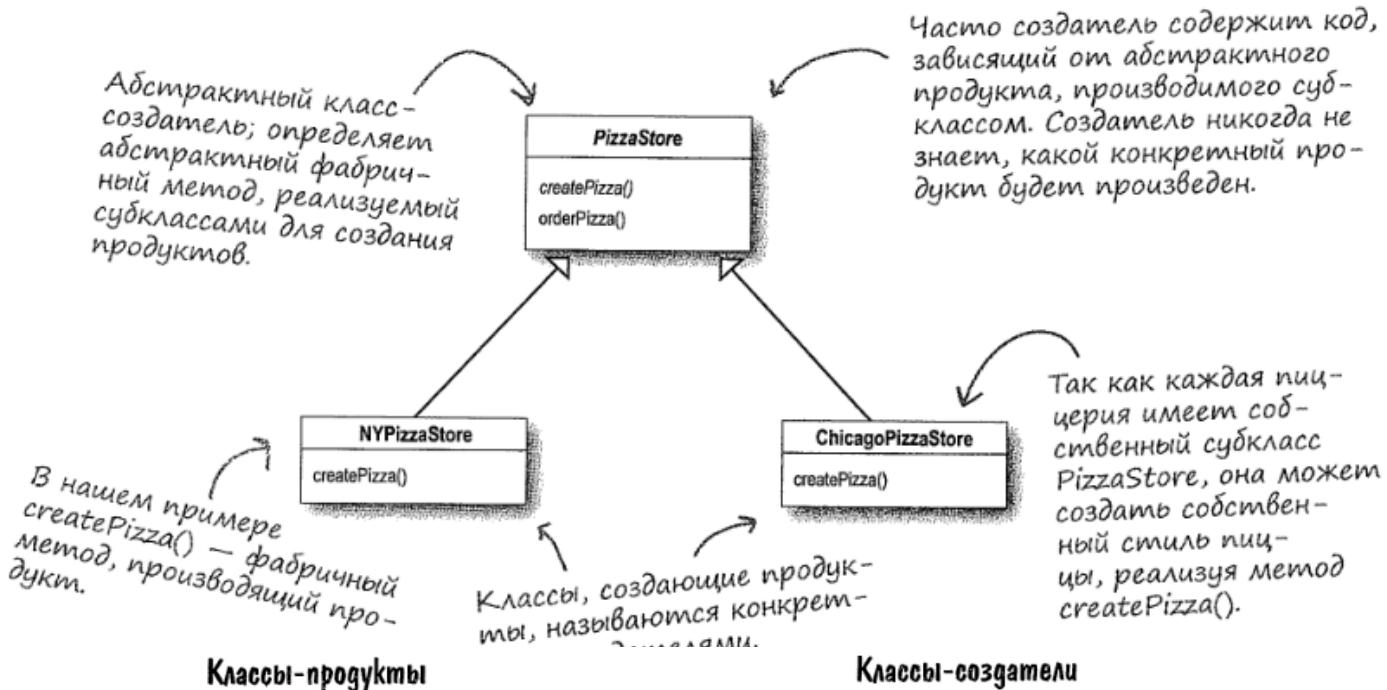
FACTORY METHOD PATTERN

http://www.youtube.com/watch?v=yDEmb5XYc_s&feature=related

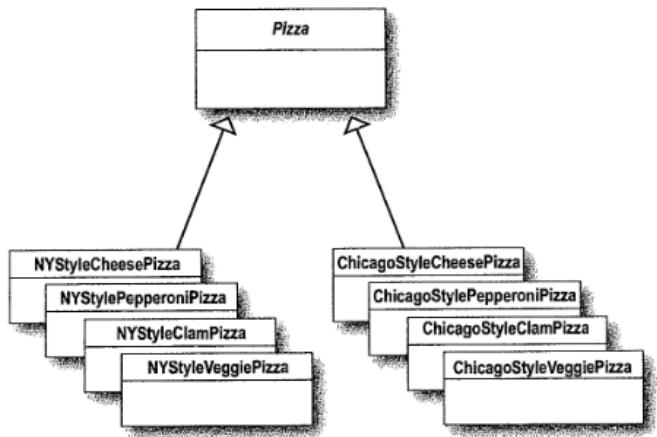
Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Все фабричные паттерны инкапсулируют операции создания объектов. Паттерн Фабричный Метод позволяет субклассам решить, какой объект следует создать. На следующих диаграммах классов представлены основные участники этого паттерна:

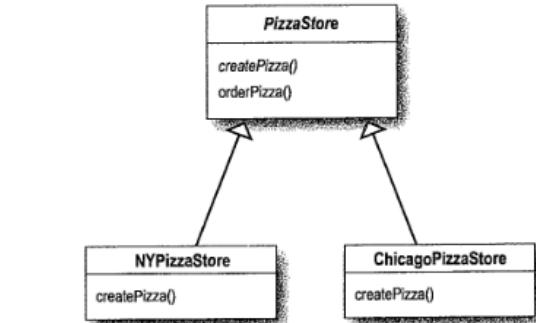
Классы-создатели



Классы-продукты



Классы-создатели



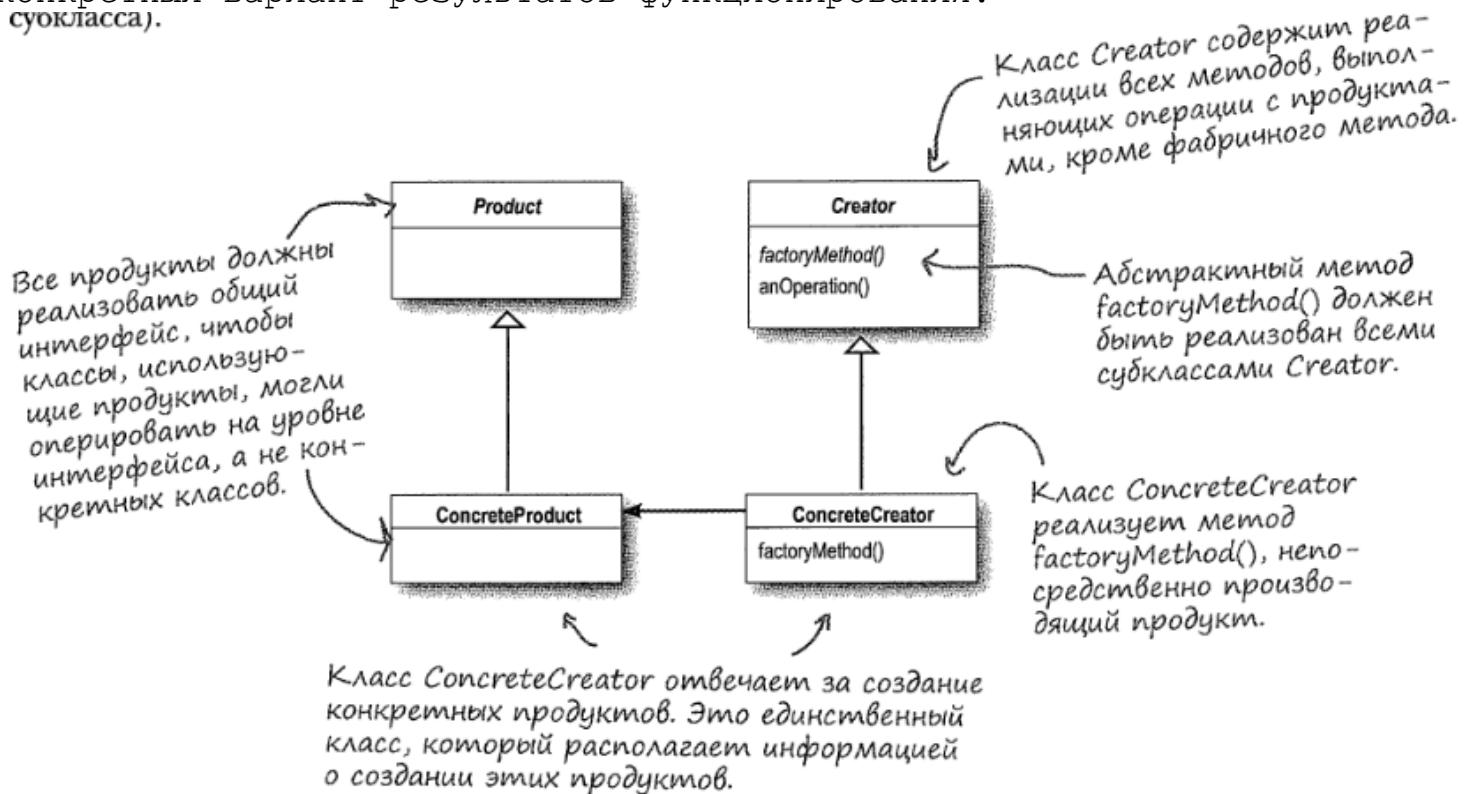
NYPizzaStore инкапсулирует сведения о том, как готовить новйоркскую пиццу.

ChicagoPizzaStore инкапсулирует все сведения о том, как готовить чикагскую пиццу.

Фабричный метод играет ключевую роль в инкапсуляции этих сведений.

Таким образом в каркасе приложения фигурируют только классы **Product** (продукты) и **Creator** (создатели). С помощью них выполняется какой-то общий алгоритм. Дальше – в уже клиентской части приложения, использующей этот каркас, на каждом этапе отдельно решается какой конкретно подкласс **ConcreteProduct** и **ConcreteCreator** инстанцируется, после чего запускается на выполнение этот общий алгоритм определенный каркасом, который уже используя эти подклассы получает на выходе конкретный вариант результатов функционирования.

суперкласса).



Участники паттерна Фабричный метод (Factory Method)

1. **Product** (Document) – продукт, абстрактный класс.
Определяет структуру, интерфейс объектов, создаваемых фабричным методом.
2. **ConcreteProduct** (MyDocument) – конкретный продукт, класс.
Реализует интерфейс **Product**, т.е. представляет один из вариантов реализации всех действий, возложенных на сущность **Product**.
3. **Creator** (Application) – создатель («главный класс»), здесь он пока абстрактный.
Объявляет фабричный метод, возвращающий объект типа **Product**.
Creator может также определять реализацию по-умолчанию фабричного метода, который возвращает объект **ConcreteProduct**.
Может вызывать фабричный метод для создания объекта **Product**
4. **ConcreteCreator** (MyApplication) – продукт, абстрактный класс.
Определяет структуру, интерфейс объектов, создаваемых фабричным методом.

Плюсы

- позволяет сделать код создания объектов более универсальным, не привязываясь к конкретным классам (**ConcreteProduct**), а оперируя лишь общим интерфейсом (**Product**);
- позволяет установить связь между параллельными иерархиями классов.

Минусы

- необходимость создавать наследника Creator для каждого нового типа продукта (ConcreteProduct).

Пример

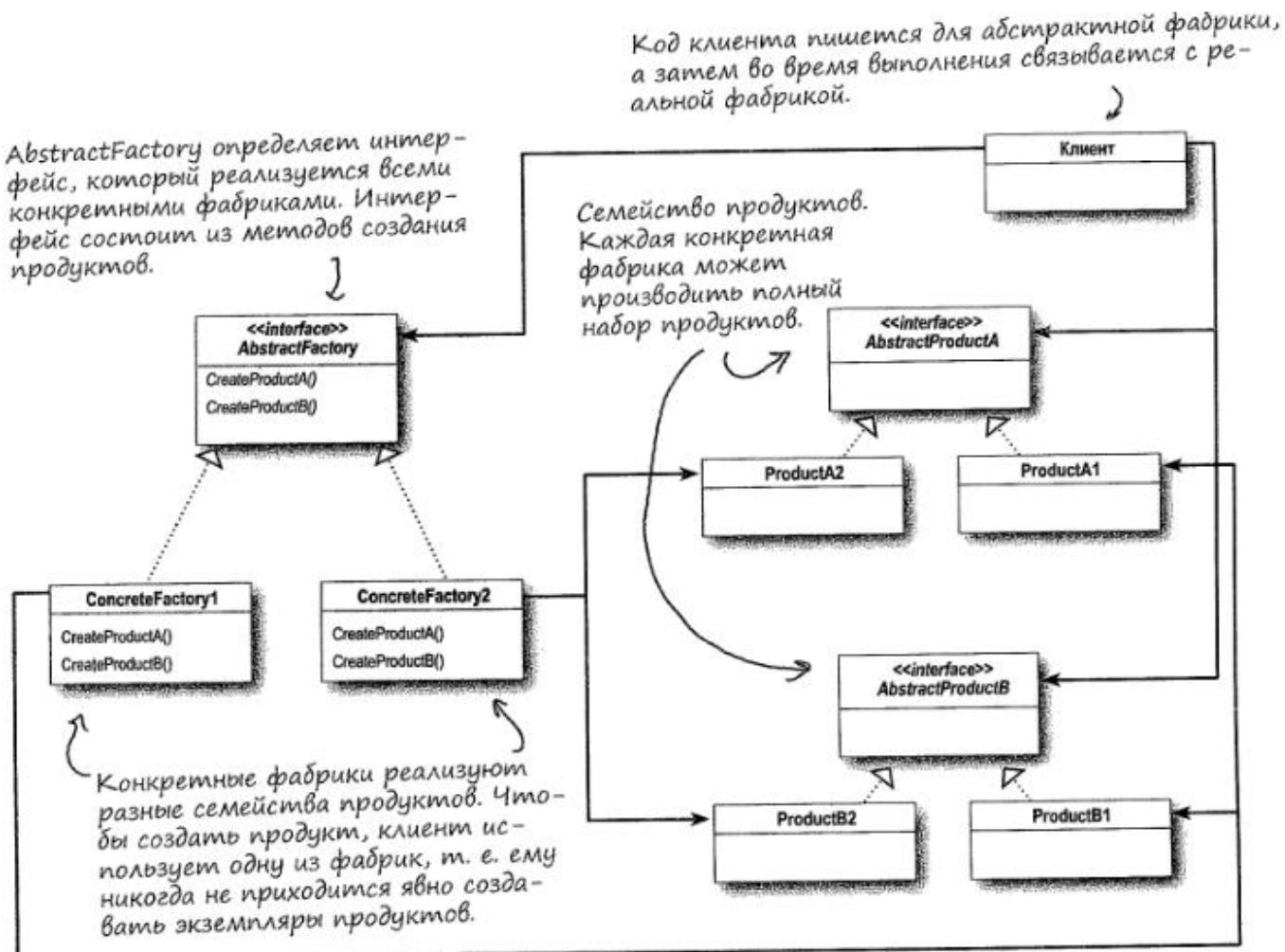
Требуется автоматизировать процесс сборки компьютера из комплектующих. Т.к. каждая новая конфигурация имеет детали разных производителей – уместно сделать единый общий абстрактный класс (Creator) с различными фабричными методами, получающими отдельные детали (Product-ы) и соответственно одна общая функция собирающая все вместе: [ComputerCreator.java](#)

Каждый фабричный метод возвращает одну из комплектующих. Класс ComputerCreator предоставляет реализации по умолчанию, которые возвращают простейшие варианты процессора, монитора, клавиатуры: [Monitor.java](#), [Processor.java](#), [KeyBoard.java](#). Далее для каждой конфигурации создаем уже свои, конкретные классы (**ConcreteCreator/ConcreteProduct**). Т.к. в результате нам нужен пентиум с расширенной клавиатурой, то все что там нужно будет сделать, учитывая, что сборку монитора можно оставить прежней это 3 класса, 1 **Creator** и 2 новых продукта: [PentiumCreator.java](#), [KeyBoardFull.java](#), [Processor64.java](#).

ABSTRACT FACTORY

Паттерн Абстрактная Фабрика предоставляет интерфейс создания семейств взаимосвязанных или взаимозависимых объектов без указания их конкретных классов.

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.



Можно добавлять сколько угодно много ConcreteFactory и Product, это не будет влиять на работоспособность клиента. Клиент работает только с интерфейсами AbstractFactory и AbstractProduct и не знает ничего о ConcreteFactory и Product.

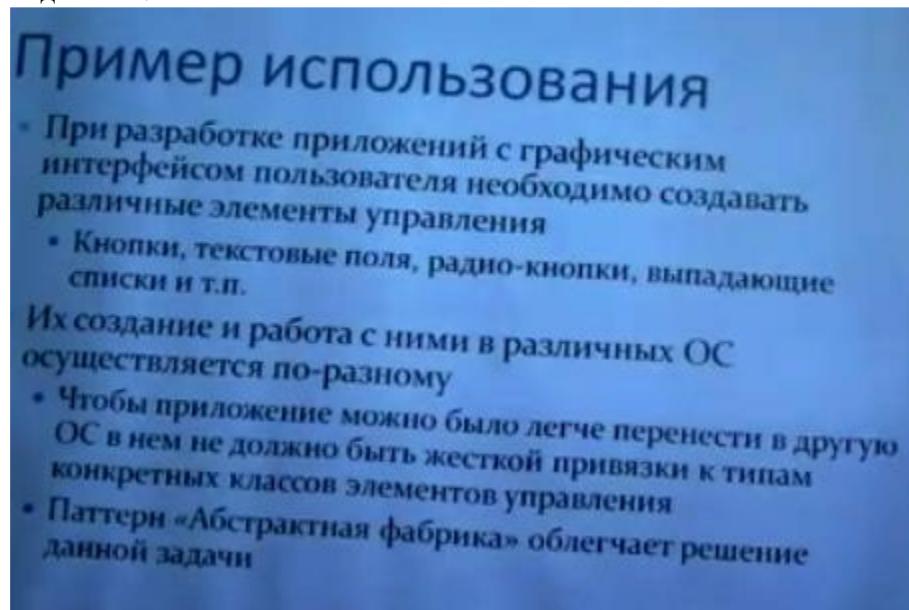
Участники

- AbstractFactory (WidgetFactory) - абстрактная фабрика: объявляет интерфейс для операций, создающих абстрактные объекты-продукты;
- ConcreteFactory (MotifWidgetFactory, PMWidgetFactory) - конкретная фабрика
 - реализует операции, создающие конкретные объекты-продукты;
- AbstractProduct (Window, ScrollBar) - абстрактный продукт:
 - объявляет интерфейс для типа объекта-продукта;
- ConcreteProduct (MotifWindow, Motif ScrollBar) - конкретный продукт:
 - определяет объект-продукт, создаваемый соответствующей конкретной фабрикой;
 - реализует интерфейс AbstractProduct; a Client - клиент;
- Client - клиент:
 - пользуется исключительно интерфейсами, которые объявлены в классах AbstractFactory и AbstractProduct.

Класс AbstractFactory объявляет только интерфейс для создания продуктов. Фактическое их создание - дело подклассов ConcreteProduct. Чаще всего для этой цели определяется фабричный метод для каждого продукта (см. паттерн фабричный метод). Конкретная фабрика специфицирует свои продукты путем замещения фабричного метода для каждого из них. Хотя такая реализация проста, она требует создавать новый подкласс конкретной фабрики для каждого семейства продуктов, даже если они почти ничем не отличаются.

Трудность

- поддерживать новый вид продуктов трудно. Расширение абстрактной фабрики для изготовления новых видов продуктов - непростая задача. Интерфейс AbstractFactory фиксирует набор продуктов, которые можно создать. Для поддержки новых продуктов необходимо расширить интерфейс фабрики, то есть изменить класс AbstractFactory и все его подклассы.



Не должно быть жесткой связки к платформенно- зависимым элементам.

Плюсы

- изолирует конкретные классы;
- упрощает замену семейств продуктов;
- гарантирует сочетаемость продуктов.

Минусы

- сложно добавить поддержку нового вида продуктов.

- Фабричный Метод основан на наследовании: создание объектов делегируется субклассам, реализующим фабричный метод для создания объектов.
- Абстрактная Фабрика основана на композиции: создание объектов реализуется в методе, доступ к которому осуществляется через интерфейс фабрики.

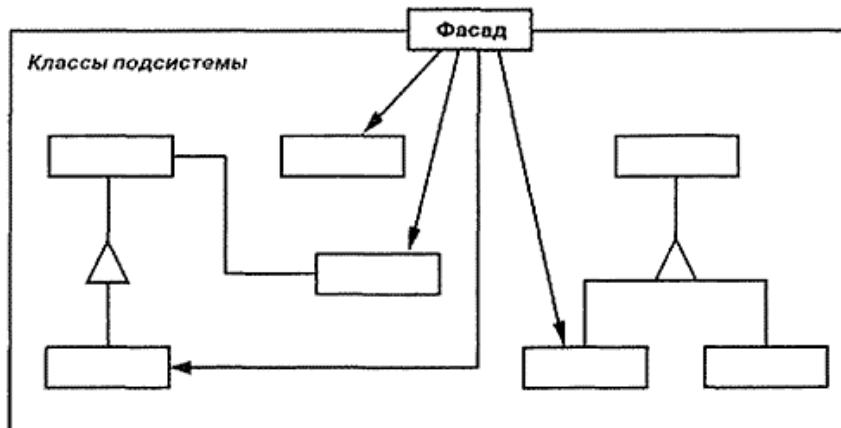
- Задача Фабричного Метода — перемещение создания экземпляров в субклассы.
- Задача Абстрактной Фабрики — создание семейств взаимосвязанных объектов без зависимости от их конкретных классов.

ФАСАД

<http://codelab.ru/pattern/facade/>

http://sourcemaking.com/design_patterns/facade/c%2523

Предоставляет единый, унифицированный интерфейс ко всей некоторой подсистеме вместо набора отдельных и многочисленных интерфейсов. Единый интерфейс целой большой системы начинает выступать как интерфейс «черного ящика» и мы не должны знать, как он устроен, какие внутри него взаимосвязи других объектов, сколько их вообще и т.д. Мы просто используем его в соответствии с его интерфейсом, задавая на вход определенные данные и получая соответствующие результаты.



Участники паттерна Фасад (Facade)

- Facade** (Compiler) – фасад.
Осведомлен о том, каким классам подсистемы адресовать запрос.
- Классы подсистемы** (Scanner, Parser, ProgramNode и т.д.).
Реализуют функциональность подсистемы.
Выполняют работу, запрошенную объектом **Facade**, которую в свою очередь запросил у **Facade**-а один из клиентов.
Ничего не «знают» о существовании самого фасада, то есть не хранят ссылок на него.

Схема использования паттерна Фасад (Facade)

Клиенты общаются с подсистемой, посылая запросы фасаду. Он переадресует их подходящим объектам внутри подсистемы. Хотя основную работу выполняют именно объекты подсистемы, фасаду, возможно, придется преобразовать свой интерфейс в интерфейсы подсистемы.

Клиенты, пользующиеся фасадом, не имеют прямого доступа к объектам подсистемы.

У [паттерна фасад](#) есть следующие преимущества:

1. Изолирует клиентов от компонентов подсистемы

Уменьшая тем самым число объектов, с которыми клиентам приходится иметь дело, упрощая работу с подсистемой.

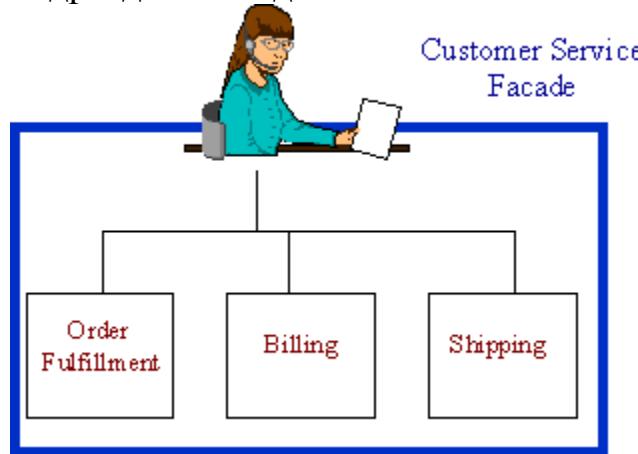
2. Позволяет ослабить связанность между подсистемой и ее клиентами.

Практическая задача

Используя паттерн «Фасад», реализуем унифицированный интерфейс к некоторой подсистеме авторизации пользователей. Сам процесс авторизации достаточно прост. На основании имени пользователя ищется соответствующая запись в базе данных, посредством интерфейса DB. Затем, сравнивается пароль найденной записи с паролем указанным пользователем.

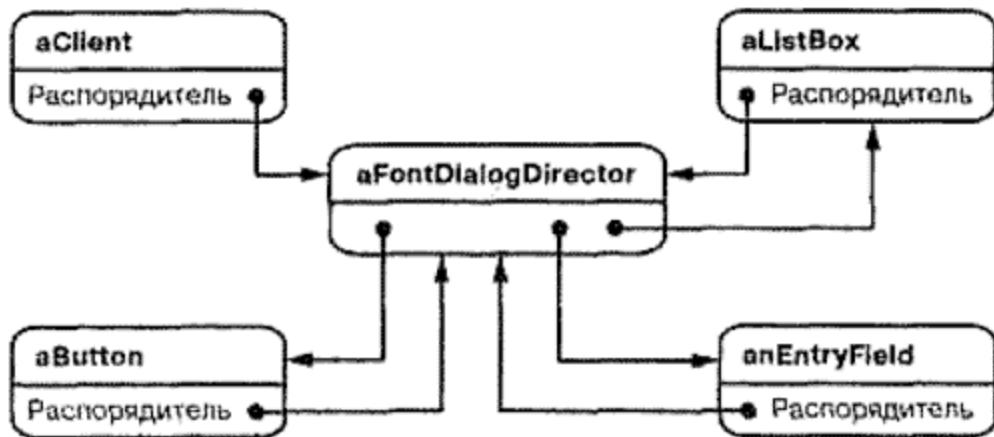
Пример "Фасада"

"Фасад" задает единообразный, высокоуровневый интерфейс для подсистем, что делает его применение проще. Заказчики сталкиваются с "Фасадом", когда заказывают товар по каталогу. Заказчик набирает единый номер и разговаривает с обслуживающим клиентов персоналом. Обслуживающий клиентов сервис действует как "Фасад", обеспечивая интерфейс к подразделению исполнения заказов, подразделению финансовых расчетов и подразделению доставки.



MEDIATOR

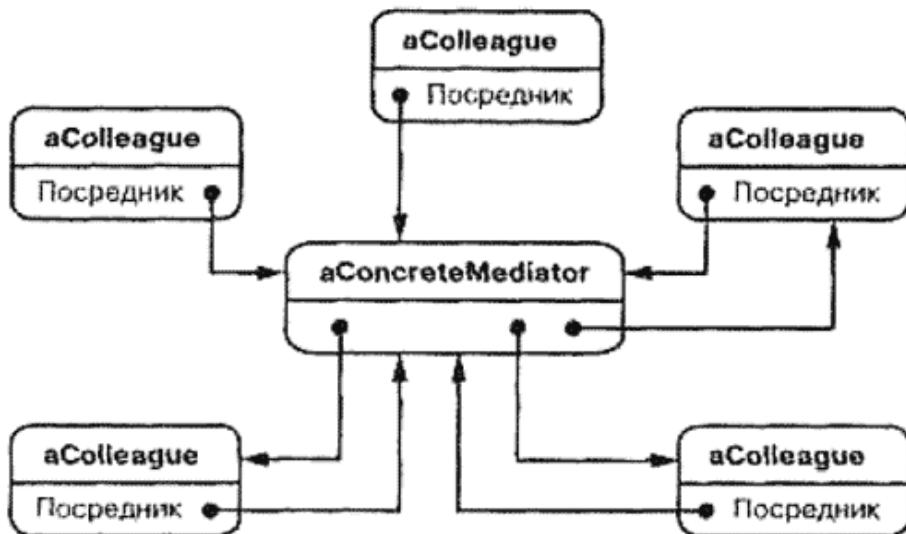
Посредник отвечает за координацию взаимодействий между группой объектов. Он избавляет входящие в группу объекты от необходимости явно ссылаться друг на друга. Все объекты располагают информацией только о посреднике, поэтому количество взаимосвязей сокращается.



Признаки применения, использования паттерна Посредник (Mediator)

Используйте [паттерн посредник](#), когда:

1. Имеются объекты, связи между которыми сложны и четко определены. Получающиеся при этом взаимозависимости не структурированы и трудны для понимания.
2. Нельзя повторно и независимо использовать объект (для решения тех же задач, но в другом контексте, например), поскольку он обменивается информацией со многими другими объектами.
3. Поведение, распределенное между несколькими классами, должно поддаваться настройке без порождения множества подклассов.



Участники паттерна Посредник (Mediator)

1. **Mediator** (DialogDirector) – посредник.

Определяет интерфейс для обмена информацией с объектами **Colleague**.

2. **ConcreteMediator** (FontDialogDirector) – конкретный посредник.

Реализует кооперативное поведение, координируя действия объектов **Colleague**.

Владеет информацией о коллегах и подсчитывает их.

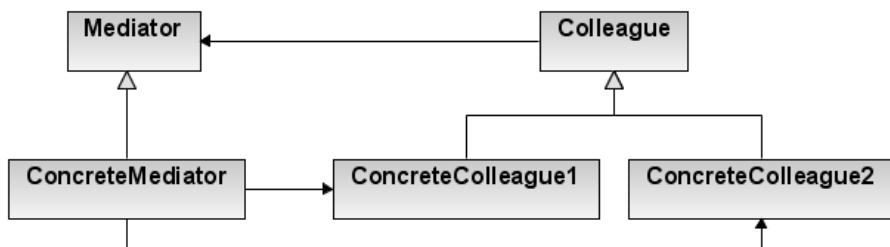
3. Классы **Colleague** (ListBox, EntryField) – коллеги:

Каждый класс **Colleague** «знает» о своем объекте распорядителе **Mediator**.

Все коллеги обмениваются информацией только с посредником, так как при его отсутствии им пришлось бы общаться между собой напрямую.

Схема использования паттерна Посредник (Mediator)

Коллеги (**Colleague**) посылают запросы посреднику (**ConcreteMediator**) и получают запросы от него. Посредник реализует заданную логику взаимодействия (кооперативное поведение) путем переадресации каждого запроса подходящему коллеге (или сразу нескольким).

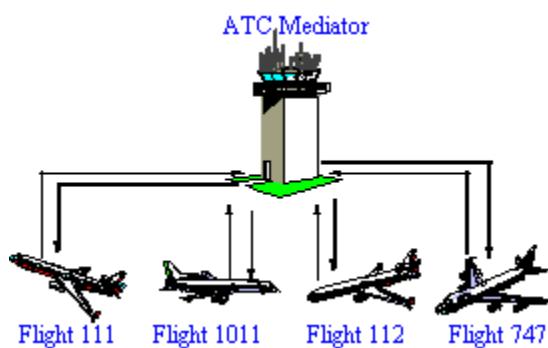


Родственные паттерны

Фасад отличается от посредника тем, что абстрагирует некоторую подсистему объектов для предоставления более удобного интерфейса. Его протокол односторонний, то есть объекты фасада направляют запросы классам подсистемы, но не наоборот. Посредник же обеспечивает совместное поведение, которое объекты-коллеги не могут или не «хотят» реализовывать, и его протокол, таким образом, дву направленный.

Коллеги могут обмениваться информацией с посредником посредством наблюдатель.

The control tower at a controlled airport demonstrates this pattern very well. The pilots of the planes approaching or departing the terminal area communicate with the tower, rather than explicitly communicating with one another. The constraints on who can take off or land are enforced by the tower. It is important to note that the tower does not control the whole flight. It exists only to enforce constraints in the terminal area (Там отслеживают и управляют поведением самолетов только в пределах терминала.).



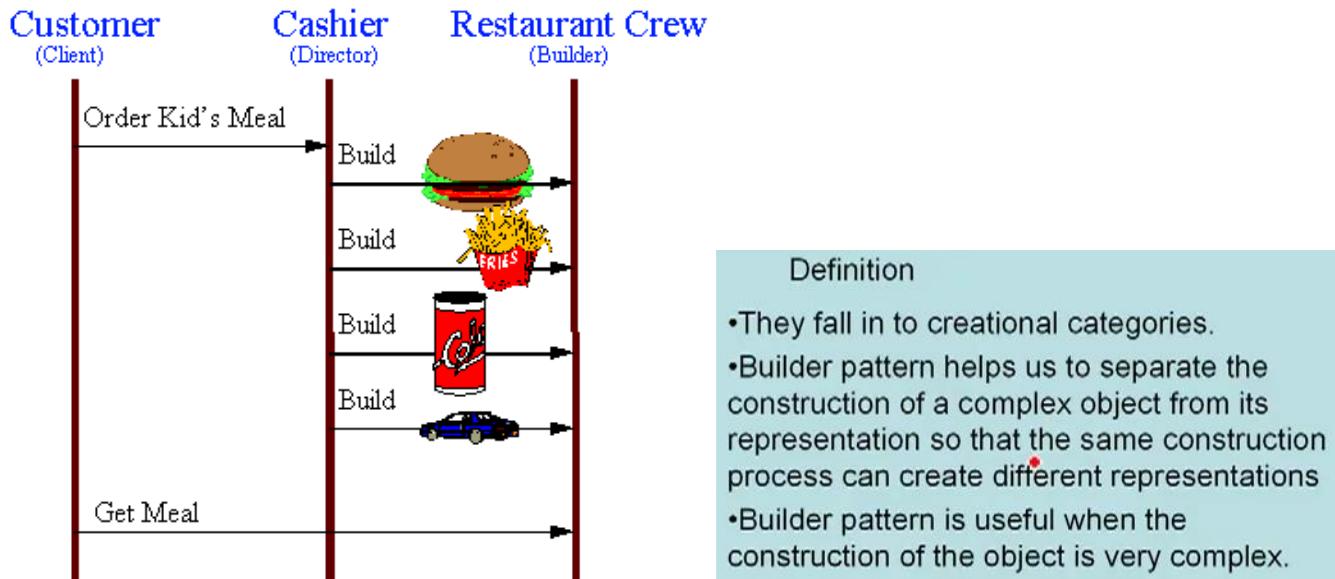
СТРОИТЕЛЬ

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Построение продукта происходит не сразу а шаг за шагом.

Шаблон "Строитель" отделяет процесс построения сложного объекта от его представления так, что один и тот же процесс построения может создавать различные представления. Этот шаблон применяется

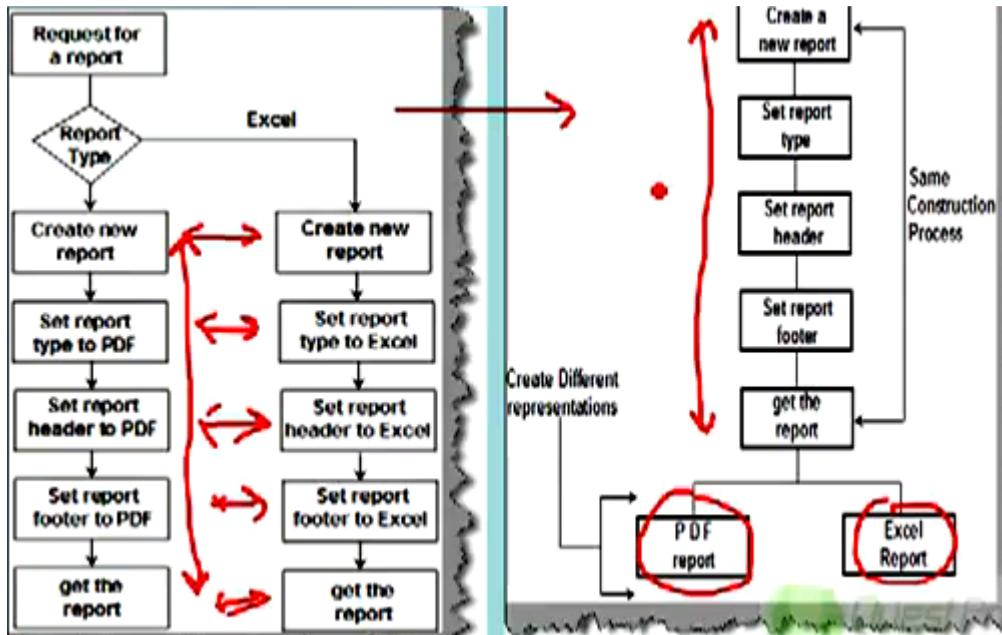
ресторанами быстрого питания для комплектования детского меню. Детское питание обычно состоит из главного блюда, гарнира, напитка и игрушки (например, гамбургер, картофель фри, кока-кола и автомобильная игрушка). Заметьте, что в содержание детского меню могут вводиться вариации, но процесс комплектования остается одинаковым. Работник за прилавком указывает команде собрать главное блюдо, гарнир и игрушку. Эти предметы помещаются в пакет. Напиток наливается в чашку и ставится рядом с пакетом.



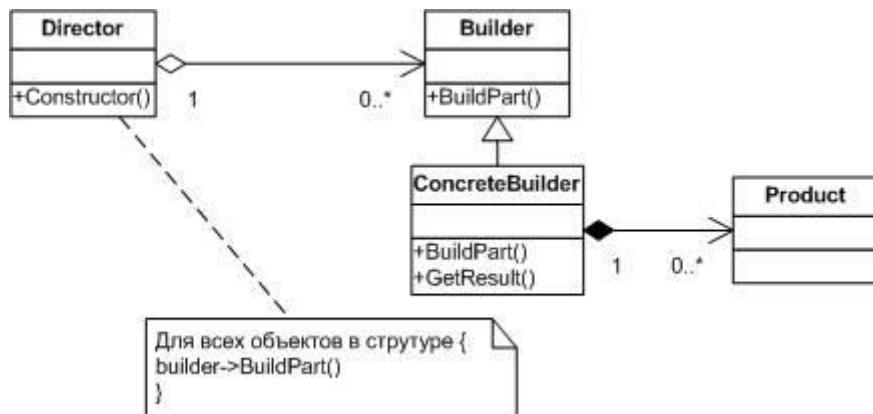
Builder: - Builder is responsible for defining the construction process for individual parts. Builder has those individual processes to initialize and configure the product.

Director: - Director takes those individual processes from the builder and defines the sequence to build the product.

Product: - Product is the final object which is produced from the builder and director coordination.



Принцип действия:



Участники

- **Builder** (TextConverter) -строитель:
- задает абстрактный интерфейс для создания частей объекта **Product**;
- **ConcreteBuilder**(ASCIIConverter,TeXConverter,TextWidgetConverter)-конкретный строитель:
- конструирует и собирает вместе части продукта посредством реализации интерфейса **Builder**;
- определяет создаваемое представление и следит за ним;
- предоставляет интерфейс для доступа к продукту (например, GetASCIIText,GetTextWidget);
- **Director** (RTFReader) - распорядитель:
- конструирует объект, пользуясь интерфейсом **Builder**;
- **Product** (ASCIIText, TeXText, TextWidget) - продукт:
- представляет сложный конструируемый объект. **ConcreteBuilder** строит внутреннее представление продукта и определяет процесс его сборки;
- включает классы, которые определяют составные части, в том числе интерфейсы для сборки конечного результата из частей.

Отношения

- -клиент создает объект-распорядитель **Director**
- -распорядитель уведомляет строителя о том, что нужно построить очередную часть продукта;
- -строитель обрабатывает запросы распорядителя и добавляет новые части к продукту;
- -клиент забирает продукт у строителя.

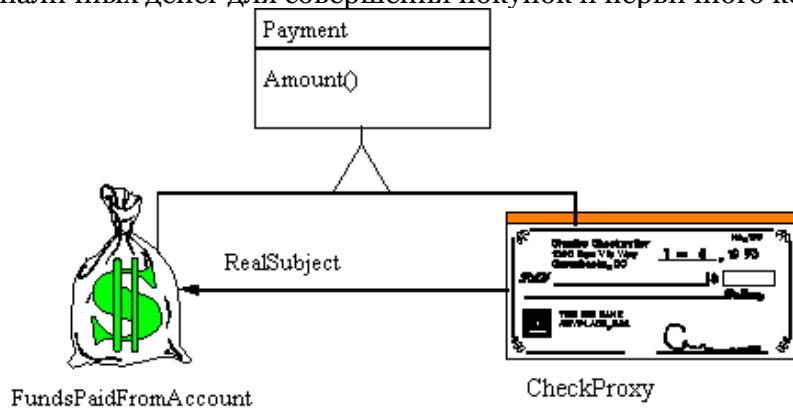
Программа, в которую заложена возможность распознавания и чтения документа в формате RTF (Rich Text Format), должна также "уметь" преобразовывать его во многие другие форматы, например в простой ASCII-текст или в представление, которое можно отобразить в виджете для ввода текста. Однако число вероятных преобразований заранее неизвестно. Поэтому должна быть обеспечена возможность без труда добавлять новый конвертор. Проблема решается с помощью паттерна **builder**.

Пример использования

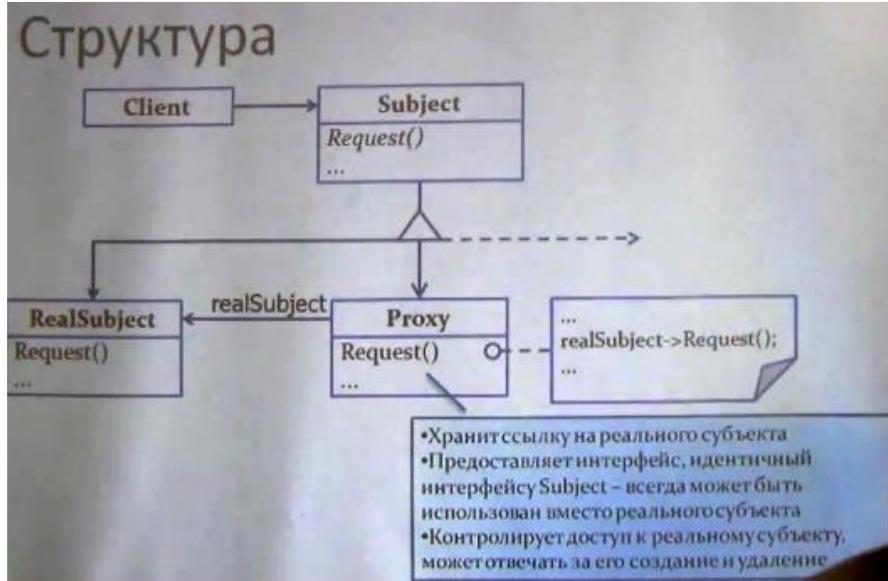
- В редакторе форматированного текстового документа необходимо реализовать возможность преобразования его в различные форматы
 - Plain text, HTML, RTF, PDF, DOC, DOCX
 - Список можно продолжить
- Задача решается путем введения сущностей
- Распорядитель - CFormattedTextReader
 - Строитель – CTextConverter
 - Конкретный Строитель – CHtmlConverter, CRtfConverter, CPlainTextConverter, CPdfConverter, ...
 - Продукт – CPlainTextDocument, CHtmlDocument, CRtfDocument, CPdfDocument, ...

Пример PROXY

"Заместитель" вводит заменителя обладающего полномочиями для обеспечения доступа к объекту. Чек или банковское поручение является заменителем денежных средств на счету. Чек может использоваться вместо наличных денег для совершения покупок и первичного контроля доступа к суммам на текущем счету.



Структура



Отношения

- Proxу при необходимости выполняет переадресацию запросов объекту RealSubject
 - Детали переадресации зависят от вида заместителя
- Клиенты взаимодействуют с субъектом только через интерфейс Subject

Удаленный – удаленный доступ к БД который находится в удаленном компе (в другом адресном пространстве)

Виртуальный – например загружать картинки не все сразу, а только те которые в данный момент нужны.

Применимость

- Удаленный заместитель (посол)
 - Предоставляет локального представителя вместо объекта в другом адресном пространстве
- Виртуальный заместитель
 - Создание «тяжелых» объектов по требованию
- Защищающий заместитель
 - Контроль доступа к исходному объекту
- Умный указатель
 - Подсчет ссылок, управление временем жизни
 - Загрузка объекта в память при первом обращении к нему
 - Блокировка доступа к объекту при обращении к нему

Результаты

- Для доступа к объекту вводится дополнительный уровень косвенности. Варианты:
 - Удаленный заместитель может скрыть факт, что объект находится в другом адресном пространстве
 - Виртуальный заместитель может выполнять оптимизацию, например, создавать объект по требованию
 - Защищающий заместитель и «умный» указатель решают дополнительные задачи при доступе к объекту

Разумно управлять доступом к объекту, поскольку тогда можно отложить расходы на создание и инициализацию до момента, когда объект действительно понадобится. Рассмотрим редактор документов, который допускает встраивание в документ графических объектов. Затраты на создание некоторых таких объектов, например больших растровых изображений, могут быть весьма значительны. Но документ должен открываться быстро, поэтому следует избегать создания всех "тяжелых" объектов на стадии открытия (да и вообще это излишне, поскольку не все они будут видны одновременно).

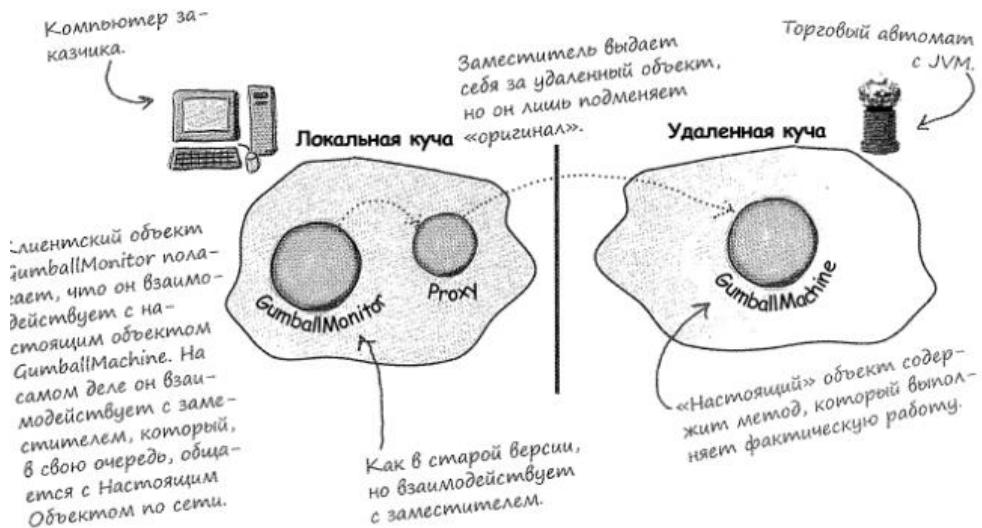
В связи с такими ограничениями кажется разумным создавать «тяжелые» объекты по требованию. Это означает «когда изображение становится видимым». Но что поместить в документ вместо изображения? И как, не усложняя реализации редактора, скрыть то, что изображение создается по требованию? Например, оптимизация не должна отражаться на коде, отвечающем за рисование и форматирование.

Решение состоит в том, чтобы использовать другой объект - заместитель изображения, который временно подставляется вместо реального изображения, заместитель ведет себя точно так же, как само изображение, и выполняет при необходимости его инстанцирование.

Предположим, что изображения хранятся в отдельных файлах. В таком случае мы можем использовать имя файла как ссылку на реальный объект. Заместитель хранит также размер изображения, то есть длину и ширину. "Зная" ее, заместитель может отвечать на запросы форматера о своем размере, не инстанцируя изображение.

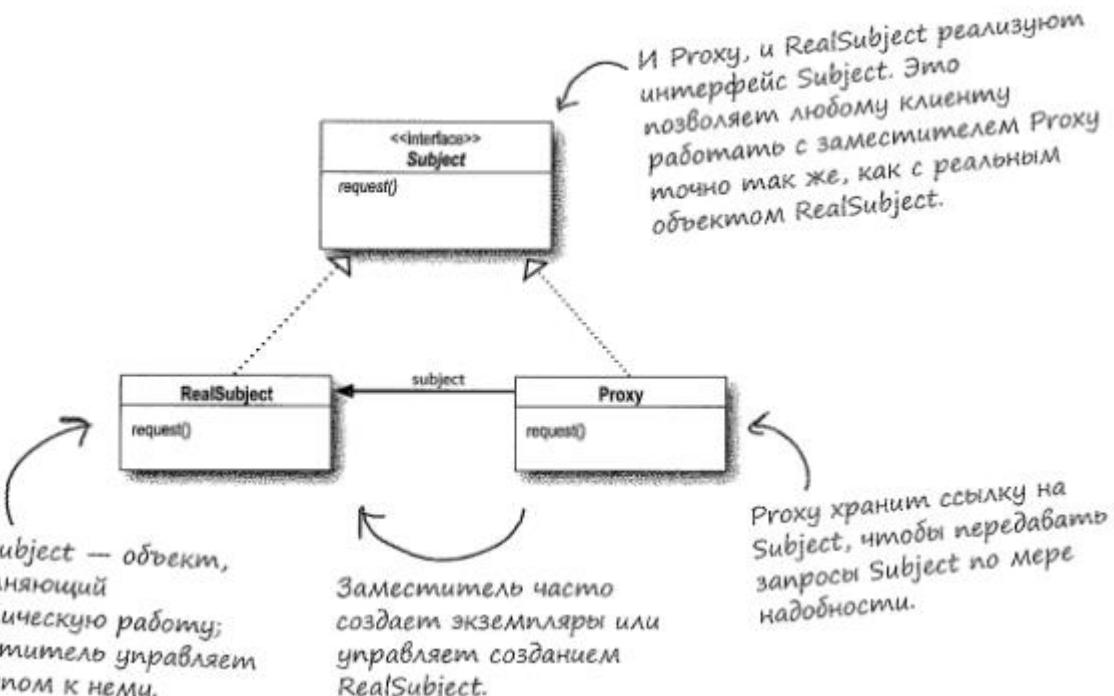
Роль «удаленного заместителя»

Удаленный заместитель действует как локальный представитель удаленного объекта. Что такое «удаленный объект»? Это объект, находящийся в куче другой виртуальной машины Java (или в более общем значении – удаленный объект, выполняемый в другом адресном пространстве). Что такое «локальный представитель»? Это объект, вызовы локальных методов которого перенаправляются удаленному объекту.



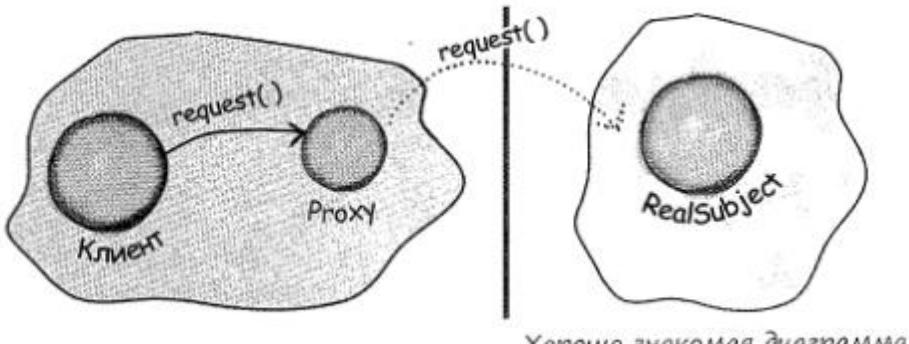
Клиентский объект работает так, словно он вызывает методы удаленного объекта. Но в действительности он вызывает методы объекта-заместителя, существующего в локальной куче, который берет на себя все низкоуровневые подробности сетевых взаимодействий.

- Удаленный заместитель управляет доступом к удаленному объекту.
- Виртуальный заместитель управляет доступом к ресурсу, создание которого требует больших затрат ресурсов.
- Защитный заместитель контролирует доступ к ресурсу в соответствии с системой привилегий.



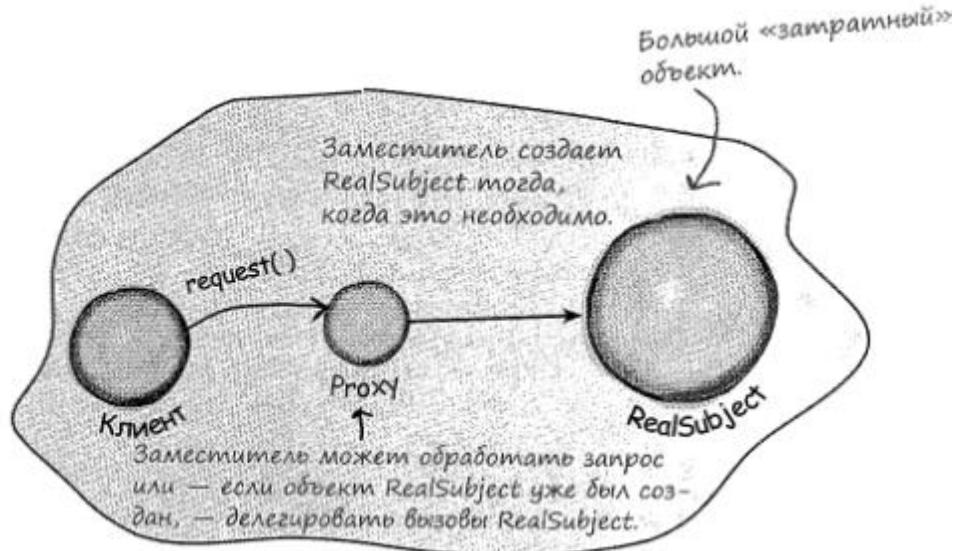
Удаленный Заместитель

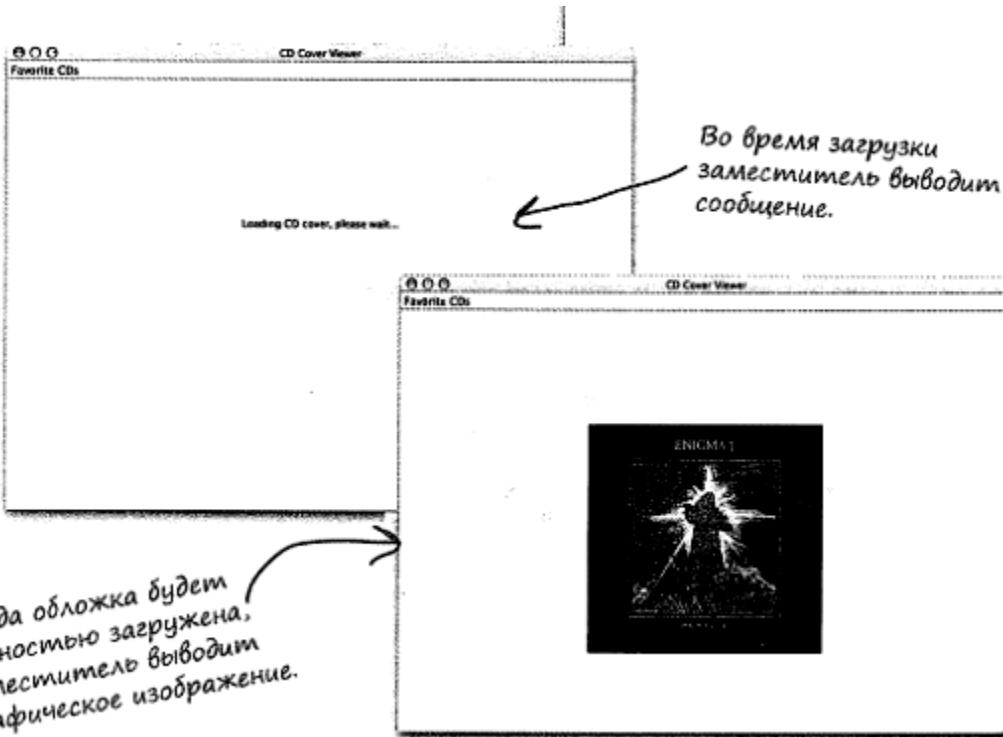
В этой разновидности заместитель выполняет функции локального представителя объекта, находящегося в другой виртуальной машине Java. Вызов метода заместителя передается по сети, метод вызывается на удаленном компьютере, результат возвращается заместителю, а затем и клиенту.



Виртуальный Заместитель

Виртуальный Заместитель представляет объект, создание которого сопряжено с большими затратами. Создание объекта часто откладывается до момента его непосредственного использования; Виртуальный Заместитель также выполняет функции суррогатного представителя объекта до и во время его создания. В дальнейшем заместитель делегирует запросы непосредственно RealSubject.





Защитный Заместитель может разрешить или запретить клиенту доступ к некоторым методам объекта в зависимости от роли клиента.

Перед нами идеальный пример ситуации, в которой применяется защитный заместитель. Что это такое? Заместитель, который управляет доступом к другому объекту в зависимости от прав пользователей. Например, для объекта работника защитный заместитель может разрешить самому работнику вызывать некоторые методы, начальнику — вызывать дополнительные методы (скажем, `setSalary()`), а отделу кадров — вызывать любые методы объекта.



**Фильтрующий
Заместитель** управляет доступом к группам сетевых ресурсов, защищая их от «недобросовестных» клиентов.

Среда обитания:
часто встречается
в корпоративных системах
защиты данных.



Кэширующий Заместитель
обеспечивает временное
хранение результатов
высокозатратных операций.

Также может обеспечивать
совместный доступ к результатам
для предотвращения лишних вычислений
или пересылки данных по сети.

Среда обитания: часто встречается
в веб-серверах, системах управления
контентом и публикаций.

- Паттерн Заместитель предоставляет «суррогат» для управления доступом к другому объекту.
- Удаленный заместитель управляет взаимодействием клиента с удаленным объектом.
- Виртуальный заместитель управляет доступом к объекту, создание которого сопряжено с большими затратами.
- Защитный заместитель управляет доступом к методам объекта в зависимости от привилегий вызывающей стороны.

"КОМПОНОВЩИК"

"Компоновщик" сводит объекты в древовидную структуру, и позволяет клиентам обращаться к отдельным объектам и их группам одинаковым образом.



Участники

- *Component (Graphic)* - компонент:
 - объявляет интерфейс для компонуемых объектов;
 - предоставляет подходящую реализацию операций по умолчанию, общую для всех классов;
 - объявляет интерфейс для доступа к потомкам и управления ими;
 - определяет интерфейс для доступа к родителю компонента в рекурсивной структуре и при необходимости реализует его.Описанная возможность необязательна;
- *Leaf (Rectangle, Line, Text, и т.п.)* - лист;
 - представляет листовые узлы композиции и не имеет потомков;
 - определяет поведение примитивных объектов в композиции;
- *Composite (Picture)* - составной объект:
 - определяет поведение компонентов, у которых есть потомки;
 - хранит компоненты-потомки;
 - реализует относящиеся к управлению потомками операции в интерфейсе класса Component;
- *Client* - клиент:
 - манипулирует объектами композиции через интерфейс Component. То есть ему нет разницы манипулирует с составным объектом или листовым.

Отношения

- Клиенты используют интерфейс класса **Component** для взаимодействия с объектами в составной структуре
 - Если получателем запроса является объект **Leaf**, то он обрабатывает запрос
 - Если получателем запроса является объект **Composite**, то обычно он перенаправляет запрос своим потомкам

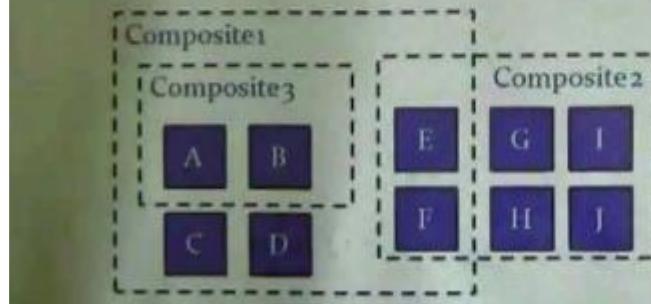
Паттерн компоновщик:

- упрощает архитектуру клиента. Клиенты могут единообразно работать с индивидуальными и объектами и с составными структурами. Обычно клиенту неизвестно, взаимодействует ли он с листовым или составным объектом. Это упрощает код клиента, поскольку нет необходимости писать функции, ветвящиеся в зависимости от того, с объектом какого класса они работают;
- облегчает добавление новых видов компонентов. Новые подклассы классов Composite или Leaf будут автоматически работать с уже существующими структурами и клиентским кодом. Изменять клиента при добавлении новых компонентов не нужно;
- Часто бывает полезно разделять компоненты, например для уменьшения объема занимаемой памяти. Но если у компонента может быть более одного родителя, то разделение становится проблемой. Возможное решение - позволить компонентам хранить ссылки на нескольких родителей. Однако в таком случае при распространении запроса по структуре могут возникнуть неоднозначности.

Элемент может входить в состав нескольких составных объектов (composite). Элементы В и D входят в состав Composite1 и composite3. Например, 1 человек может работать в нескольких организациях.

Разделение компонентов

- Для уменьшения потребления памяти возможно использование одного компонента в составе нескольких составных объектов
 - Возможная проблема – неоднозначность при определении родительского узла



Ограничение:

Если нужно ограничить объединение определенных компонентов (например не объединять в группы тексты, графические объекты можно объединять) тогда приходится прибегать к проверкам во время выполнения программы, что усложняет код компоновщика, который будет проверять какие элементы можно добавить в композицию, какие нет.

Пример использования

- В программах вроде Corel Draw, Visio или PowerPoint пользователь может создавать сложные объекты, группируя их из простых или сложных объектов
- Над полученными составными объектами возможно выполнять те же операции, что и над простыми
- Использование паттерна «Компоновщик» облегчает решение данной задачи

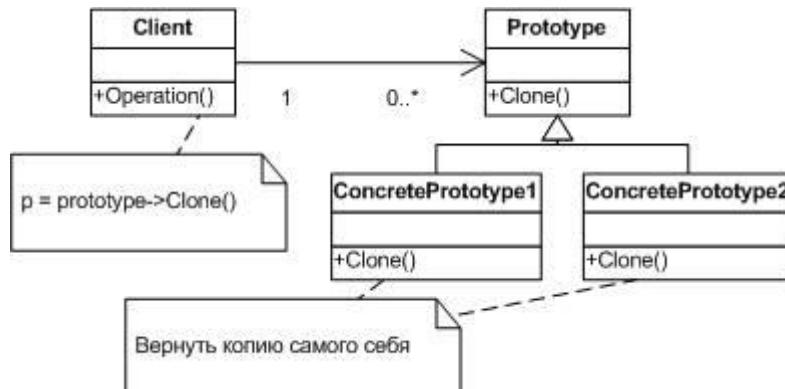
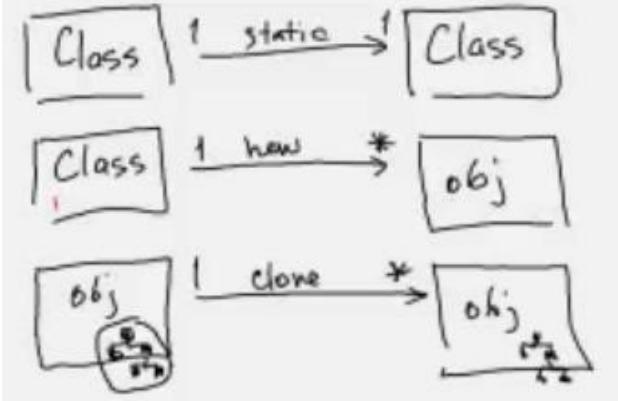
ПРОТОТИП

http://www.youtube.com/watch?v=8ryPDBRH0aE&list=UUj1txX_A5byHhrPKuGBKZIA&index=9&feature=plcp

Type VS Object

Паттерн прототип определяет тип который в дальнейшем позволяет порождать объекты. И создание этих объектов происходит клонированием прототипа.

Это паттерн создания объекта через клонирование другого объекта вместо создания через конструктор. Паттерн используется чтобы избежать дополнительных усилий по созданию объекта стандартным путем (имеется в виду использование ключевого слова 'new')



Участники

- Prototype (Graphic) - Прототип:
- объявляет интерфейс для клонирования самого себя;
- ConcretePrototype - конкретный прототип:
- реализует операцию клонирования себя;
- Client (GraphicTool) - клиент:
- создает новый объект, обращаясь к прототипу с запросом клонировать себя.

У прототипа те же самые результаты, что у абстрактной фабрики и строителя: он скрывает от клиента конкретные классы продуктов, уменьшая тем самым число известных клиенту имен.

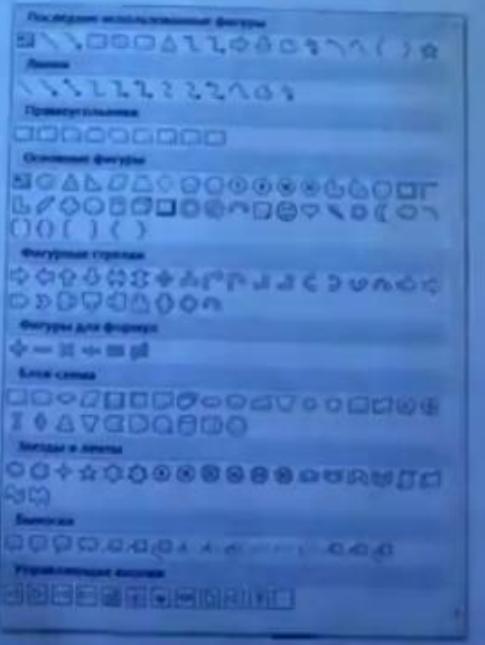
Преимущество: добавление и удаление продуктов во время выполнения. Прототип позволяет включать новый конкретный класс продуктов в систему, просто сообщив клиенту о новом экземпляре-прототипе. Это несколько более гибкое решение по сравнению с тем, что удастся сделать с помощью других порождающих паттернов, ибо клиент может устанавливать и удалять прототипы во время выполнения;

Основной **недостаток** паттерна прототип заключается в том, что каждый подкласс класса Prototype должен реализовывать операцию Clone, а это далеко не всегда просто. Например, сложно добавить операцию Clone, когда рассматриваемые классы уже существуют. Проблемы возникают и в случае, если во внутреннем представлении объекта есть другие объекты или наличествуют круговые ссылки.

Пользователь может создавать фигуры на основе имеющихся прототипов фигур.

Пример использования паттерна «Прототип»

- Палитра готовых фигур в программах вроде Microsoft PowerPoint
 - Пользователь может создавать фигуры на основе имеющихся прототипов фигур
- Количество прототипов фигур потенциально может быть неограниченным



SINGLETON

http://www.youtube.com/watch?v=oSBQM8ofV7k&list=UUj1txX_A5byHhrPKuGBKZIA&index=8&feature=plcp

Singleton

Ensure a class only has one instance, and provide a global point of access to it

```
public class Singleton
{
    private static Singleton instance;

    internal static Singleton GetInstance()
    {
        if (instance == null)
            instance = new Singleton();

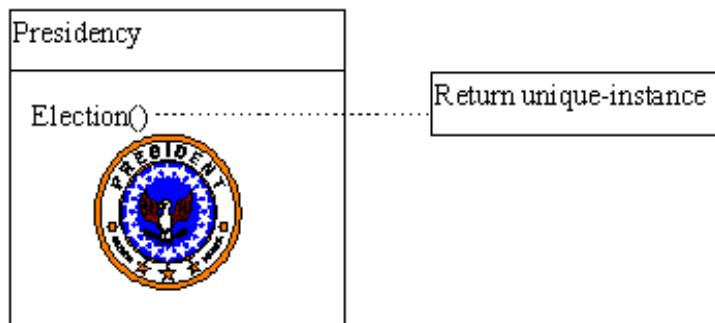
        return instance;
    }
}
```

Отношения

- Клиенты получают доступ к экземпляру класса Singleton только через его операцию Instance

- Большая гибкость, чем у статических функций класса
- В C++ статические функции не могут быть виртуальными => нельзя использовать полиморфизм

The office of the President of the United States is a *Singleton*. The United States Constitution specifies the means by which a president is elected and limits the term of office(ограничивает срок его пребывания в должности). As a result, there can be at most one active president at any given time. "The President of the United States" is a global point of access that identifies the person in the office.



Гуру: Паттерн Одиночка во многих отношениях представляет собой схему, которая гарантирует, что для данного класса может быть создан один и только один объект. Если кто-нибудь придумает более удачное решение, мир о нем услышит; а пока паттерн Одиночка, как и все паттерны, представляет собой проверенный временем механизм создания единственного объекта. Кроме того, Одиночка, как и глобальная переменная, предоставляет глобальную точку доступа к данным, но без ее недостатков.

Программист: Каких недостатков?

Гуру: Простейший пример: если объект присваивается глобальной переменной, он может быть создан в начале работы приложения. Верно? А если этот объект расходует много ресурсов, но никогда не будет использоваться приложением? Как вы увидите, паттерн Одиночка позволяет создавать объекты в тот момент, когда в них появится необходимость.

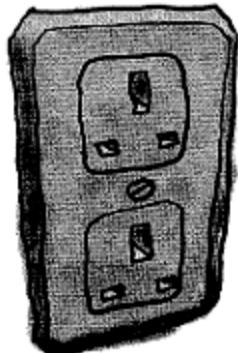
Объявлять статические переменные в качестве синглтона тоже неправильно, так как эти переменные нельзя использовать в наследовании. Статика и наследование друг с другом не дружат.

Адаптер (Adapter) - GoF

Проблема	Необходимо обеспечить взаимодействие несовместимых интерфейсов или как создать единый устойчивый интерфейс для нескольких компонентов с разными интерфейсами.
Решение	Конвертировать исходный интерфейс компонента к другому виду с помощью промежуточного объекта - адаптера, то есть, добавить специальный объект с общим интерфейсом в рамках данного приложения и перенаправить связи от внешних объектов к этому объекту - адаптеру.

An *adapter* allows classes to work together that normally could not because of incompatible interfaces

Европейская розетка



Европейская розетка
имеет один интерфейс
подключения устройств.



Адаптер

Стандартная вилка



Американский ноутбук
рассчитан на другой
интерфейс.

Адаптер обеспечивает
согласование этих двух
интерфейсов.

Адаптер включается между вилкой ноутбука и розеткой европейского стандарта; он адаптирует европейскую розетку, чтобы вы могли подключить к ней свое устройство и пользоваться ей. Или можно сказать иначе: адаптер приводит интерфейс розетки к интерфейсу, на который рассчитан ваш ноутбук.

Какие?

Обратите внимание: Клиент полностью изолирован от Адаптера; они ничего не знают друг о друге.

1 Клиент обращается с запросом к адаптеру, вызывая его метод через целевой интерфейс.

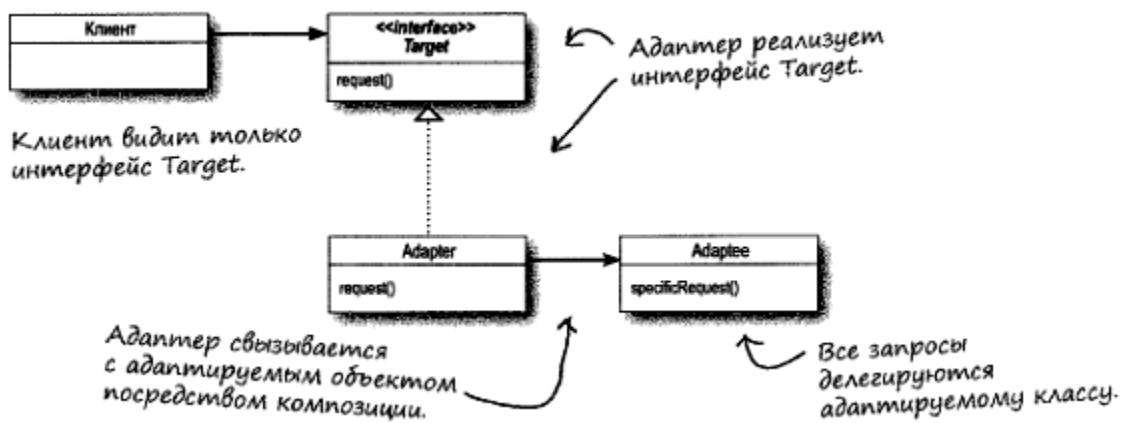
2 Адаптер преобразует запрос в один или несколько вызовов к адаптируемому объекту (в интерфейсе по-следнего).

3 Клиент получает результаты вызова, даже не подозревая о преобразованиях, выполненных адаптером.

- Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты
- Обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна

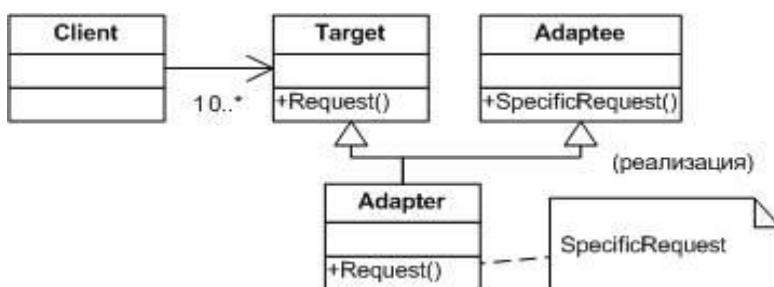
Применимость

- Необходимо использовать существующий класс, но его интерфейс не соответствует заданным требованиям
- Создание повторно используемого класса, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы

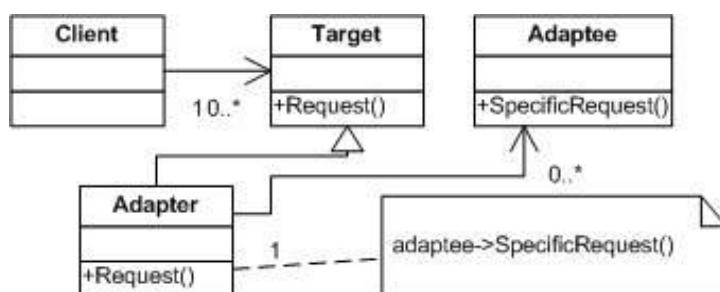


Принцип действия:

Адаптер класса использует множественное наследование для адаптации одного интерфейса к другому.



Адаптер объекта применяет композицию объектов.



- *Target (Shape)* - целевой:
 - определяет зависящий от предметной области интерфейс, которым пользуется Client;
- *Client (DrawingEditor)* - клиент:
 - вступает во взаимоотношения с объектами, удовлетворяющими интерфейсу Target;
- *Adaptee (TextView)* - адаптируемый:
 - определяет существующий интерфейс, который нуждается в адаптации;
- *Adapter (TextShape)* - адаптер:
 - адаптирует интерфейс Adaptee к интерфейсу Target.

Отношения (адаптер класса)

- Адаптер класса адаптирует Adaptee к Target, перепоручая действия конкретному классу Adaptee
 - Этот паттерн не будет работать, если мы захотим одновременно адаптировать класс и его подклассы
- Позволяет адаптеру Adapter заменить некоторые операции адаптируемого класса Adaptee

Адаптер объектов

Использование композиции дает мне немалые преимущества. Я могу адаптировать не только отдельный класс, но и все его субклассы.

В моих краях рекомендуется отдавать предпочтение композиции перед наследованием; возможно, получается на несколько строк больше, но мой код просто делегирует вызовы адаптируемому объекту. Мы выбираем гибкость.

Кого волнует один крошечный объект? Ты позволяешь быстро переопределить метод, но поведение, которое я добавляю в код адаптера, работает с моим адаптируемым классом и всеми его субклассами.

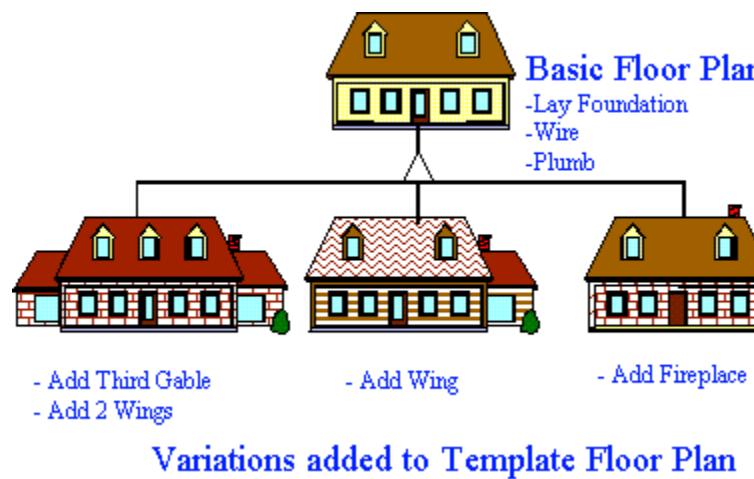
Адаптер классов

Верно, у меня с этим проблемы — я рассчитан на один адаптируемый класс, но зато мне не приходится реализовать его заново. А если потребуется, я могу переопределить поведение адаптируемого класса, ведь речь идет о простом субклассировании.

Гибкость — возможно. Эффективность? Нет. Для моей работы необходим лишь один экземпляр меня самого, а лишние экземпляры адаптера и адаптируемого класса не нужны.

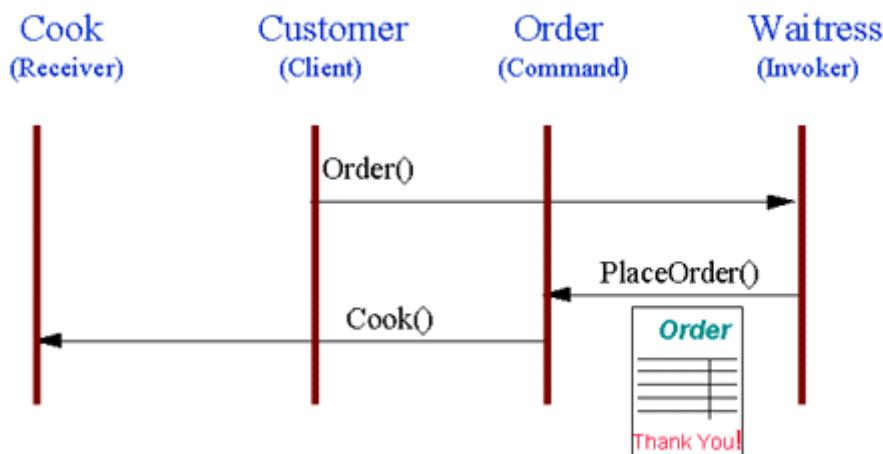
Template Method Example

The *Template Method* defines a skeleton of an algorithm in an operation, and defers some steps to subclasses. В жилищном строительстве "Шаблонный метод" используется при разработке новых типовых проектов. Фундаменты, окна, сантехника и проводка будут для каждого дома идентичными. Вариации вводятся на последних стадиях разработки, для производства более широкого спектра моделей.



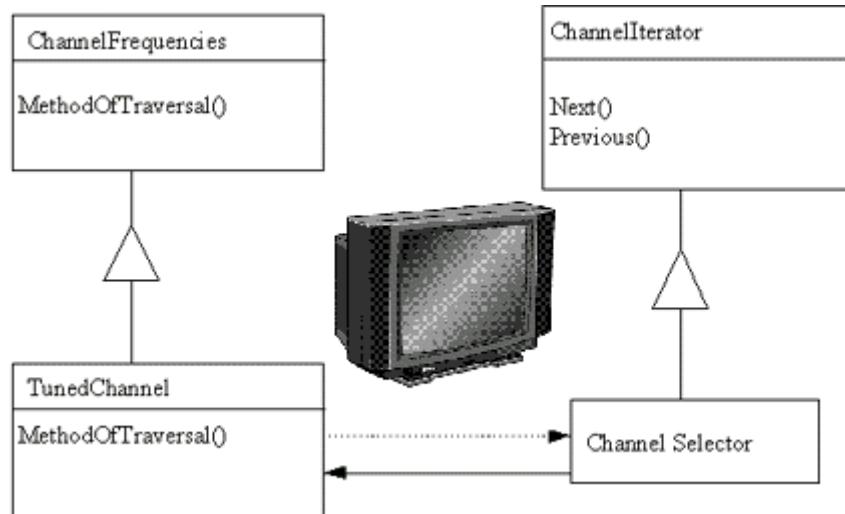
Command Example

The *Command* pattern allows requests to be encapsulated as objects, thereby allowing clients to be parameterized with different requests. The "check" at a diner is an example of a *Command* pattern. The waiter or waitress takes an order, or command from a customer, and encapsulates that order by writing it on the check. The order is then queued for a short order cook. Note that the pad of "checks" used by different diners is not dependent on the menu, and therefore they can support commands to cook many different items.



Iterator Example

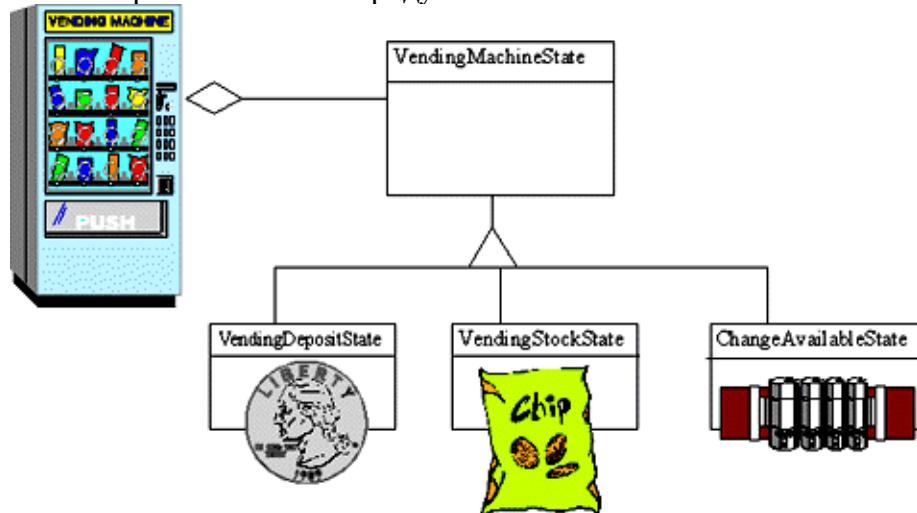
The *Iterator* provides ways to access elements of an aggregate object sequentially without exposing the underlying structure of the object. On early television sets, a dial was used to change channels. When channel surfing, the viewer was required to move the dial through each channel position, regardless of whether or not that channel had reception. On modern television sets, a next and previous button are used. When the viewer selects the "next" button, the next tuned channel will be displayed. Consider watching television in a hotel room in a strange city. When surfing through channels, the channel number is not important, but the programming is. If the programming on one channel is not of interest, the viewer can request the next channel, without knowing its number.



Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

STATE PATTERN

Шаблон "Состояние" позволяет объекту менять свое поведение, когда его внутреннее состояние изменилось . Этот шаблон можно наблюдать в торговом автомате. Торговые автоматы определяют свое состояние основываясь на реестре товаров, количестве полученных денег, возможности вносить изменения и т.д. Когда деньги приняты и выбор сделан, торговый автомат либо выдаст товар и не изменит свое состояние, либо выдаст продукт и изменит свое состояние, либо не выдаст продукт из-за недостаточного количества денег или исчерпания запасов продукта.

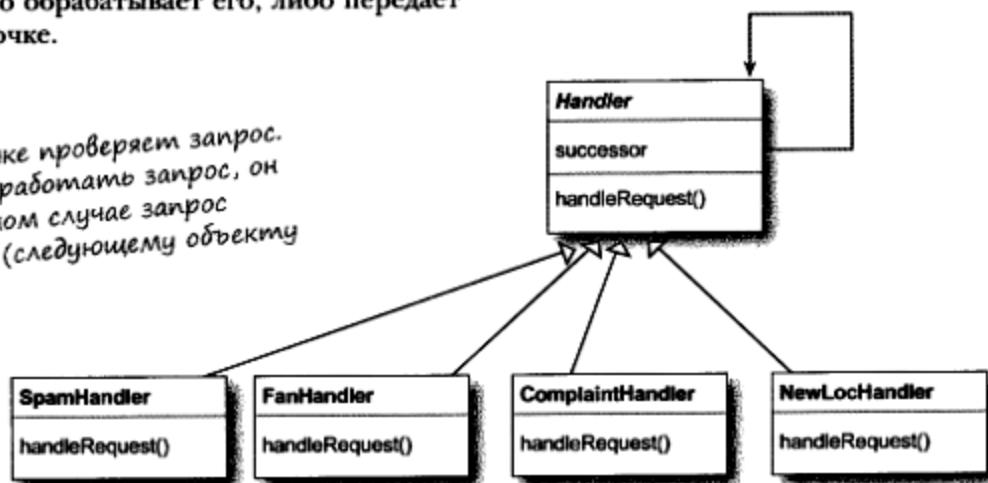


CHAIN OF RESPONSIBILITY

Как использовать паттерн Цепочка Обязанностей

В паттерне Цепочка Обязанностей создается цепочка объектов, последовательно анализирующих запрос. Каждый объект получает запрос и либо обрабатывает его, либо передает следующему объекту в цепочке.

Каждый объект в цепочке проверяет запрос. Если объект может обработать запрос, он это делает; в противном случае запрос передается преемнику (следующему объекту в цепочке).



Полученное сообщение передается первому обработчику, SpamHandler. Если SpamHandler не может обработать запрос, то последний передается FanHandler. И так далее...

Каждое сообщение передается первому обработчику.



Сообщение, добралось до конца цепочки, остается необработанным хотя вы всегда можете реализовать обработчик по умолчанию.

Chain of Responsibility Benefits

- Decouples the sender of the request and its receivers.
- Simplifies your object because it doesn't have to know the chain's structure and keep direct references to its members.
- Allows you to add or remove responsibilities dynamically by changing the members or order of the chain.

Chain of Responsibility Uses and Drawbacks

- Commonly used in windows systems to handle events like mouse clicks and keyboard events.
- Execution of the request isn't guaranteed; it may fall off the end of the chain if no object handles it (this can be an advantage or a disadvantage).
- Can be hard to observe the runtime characteristics and debug.

ROLES

A collaboration-based design decomposes an application into a set of collaborations and a set of objects. A collaboration specifies a set of allowed behaviors for a group of objects cooperating to perform a specific activity. The part of an object that fulfills its responsibilities within a collaboration is referred to as its role. A collaboration is a co-operating collection of roles. An object can participate in a collaboration if it supports the required role. Objects normally participate in more than one collaboration and hence comprise of several roles. It should be noted that an object does not need to play a role in each collaboration in the application.

Role diagrams can be more easily composed than class diagrams. They are more abstract. The role diagram notation looks like the notation from the design pattern catalog, based on roles rather than classes.

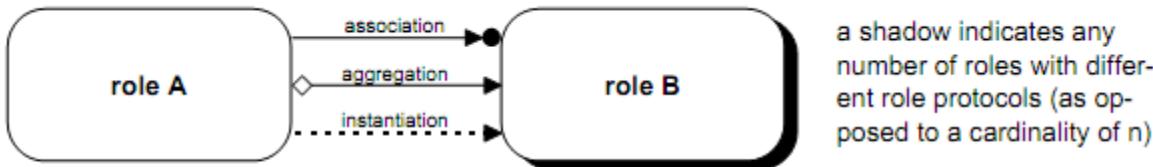


Figure 2-1: Roles and relationships

A role diagram describes the roles objects play in a collaboration. A role defines the abstract state and behavior of an object in the collaboration. Objects play roles. Thus, a single object can play several roles, and several objects can play the same role. Playing a role means entering a state.

Role A may imply role B ($A \Rightarrow B$), two roles A and B may prohibit each other ($\neg(A \wedge B)$)

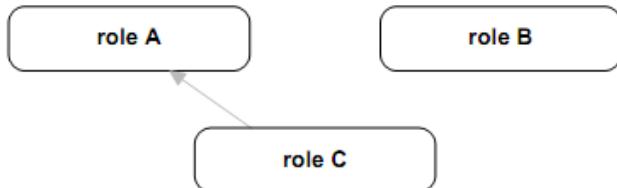


Figure 2-2: Implication between roles in role diagrams

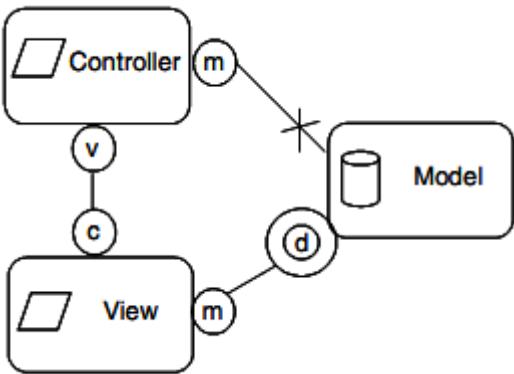
Role A implies role B, if an object playing role A always also plays role B. If a role A prohibits its role B, an object playing role A never plays role B within the same collaboration; this is symmetric. If nothing can be said, any combination may occur. Implication is usually implemented by inheritance on the class level.

Role diagrams can be more easily composed than class diagrams. They are more abstract.

Despite of its importance, the class dimension is limited in its ability to describe objects. First, the class view of objects is static. A class is basically a mold for making objects [26] and classifying objects [41]. Once an object is created for a class, it belongs to that class forever, hence “Once an engineer, forever an engineer.” Although an object may appear to be able to change its type in a polymorphic type hierarchy, it still cannot change its base class and behavior. Second, the class centered view tends

to project collaborations. A role model consists of a set of roles and their interactions. Fig. 2 shows a role model consisting of three roles: Model, View and Controller. Wirfs-

An object may play several different roles and a given role may be played by different objects.



ing the dynamics of a person or an object. For example, an academic through the eyes of her students is a lecturer; a traveler in the eyes of a travel agent and an author by her readers. Lecturer, traveler and author are the three roles played by the same person. Each role characterizes the person's position in a context meaningful from a particular viewpoint. Such a viewpoint is an abstraction, which selects the detail of

Roles describe an object from different viewpoints.

Often a role needs to cooperate with other roles to perform some task.

relations. Each role focuses on the relevant aspect of an object and filters out the irrelevant information. When an academic takes on the role of traveler, the focus is on the responsibilities and collaborations of the traveler. Similarly, by focusing on the Author role, the non-author aspects are ignored. The role concept is therefore about separation of concerns.

Role is dynamic and flexible. An object can play different roles, change roles and take on or off roles. An academic can take on an additional role as a year tutor or change a year tutor role to an examination officer role. Each of these roles can be considered and designed independently of others. For example, the role of an academic as Traveler is independent of the role of Year Tutor. These two roles live in two different contexts and interact with different other roles; the designs of these two roles can be modified separately without affecting one another. Hence when an object

Role is reusable and adaptable. A role is an extrinsic property of an object and may be played by different objects; likewise, an object may play several different roles. An academic may play several roles; an accountant is a role played by more than one administrative member; a travel agent is a role for all the staff working in a travel agent. Roles are therefore reusable abstractions for objects. Role collaborations

3.3 Designing a Network Point Model with Roles

A public transport network is a network of bus or train services. Such a network is made of *points* and *links* between points. A point in a network can represent a whole town, a place within a town, an individual vehicle stop, or the bus bay station at a stop – depending on the level of granularity of interest to an application (See Fig. 4). Therefore, a point in a network is not simply the smallest entity in space. Rather, it is a complex entity that may contain other smaller points and links between them and that play many roles. A point is thus a complex real world entity, spatially limited in a

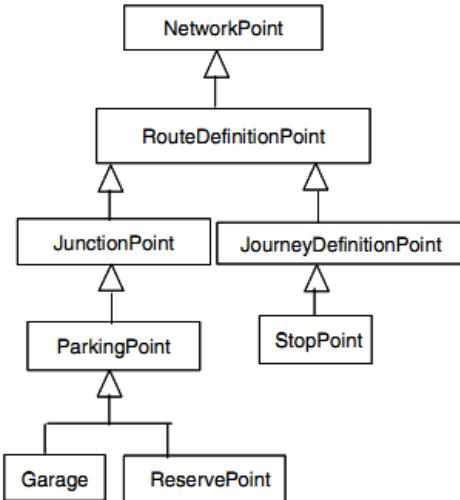


Fig. 5. A class hierarchy for point types in a public transport network. All points are by definition network points. There are already five levels of inheritance even with just seven point types. This design is inflexible because inserting a new point type will affect the application that uses it.

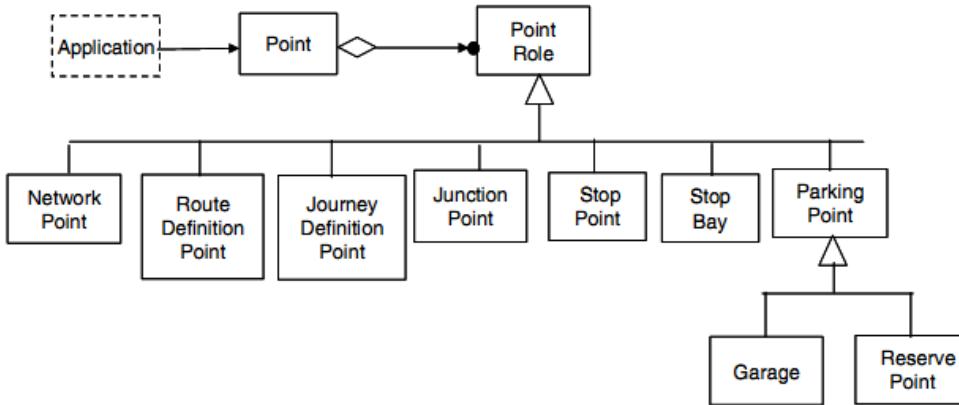


Fig. 6. Redesigning points in Fig. 5 into roles in a public transport network. A point can be attached to a specific role depending on its use in the application. This design is flexible because inserting a new point will not affect the point and the application that uses the point.

The early design of the point model was class-based; points are represented as classes and organized into a hierarchy of point types using inheritance (Fig. 5). All

Fig. 5 does not show the overlapping point types. Clearly, such a representation is inflexible because inserting a new type of point in the network affects the entire network structure. Maintaining evolving points becomes a difficult task.

can be specialized into either a garage or a reserve point. In contrast to the class-based model (Fig. 5), the new point model is flexible because inserting a new point role in a network will not affect the entire network structure. Maintaining evolving points

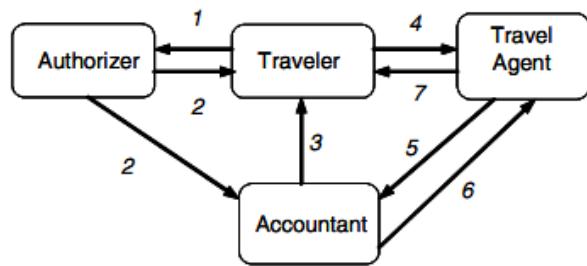


Fig. 7. A role model represents the collaboration context for the academic flight booking task. Arrows represent interactions between roles; an interaction starts from one role and ends in another; numbers show the sequence of interactions.

A **responsibility** is an action taken by a role. Traveler takes the travel request responsibility which will result in collaborating with Authorizer

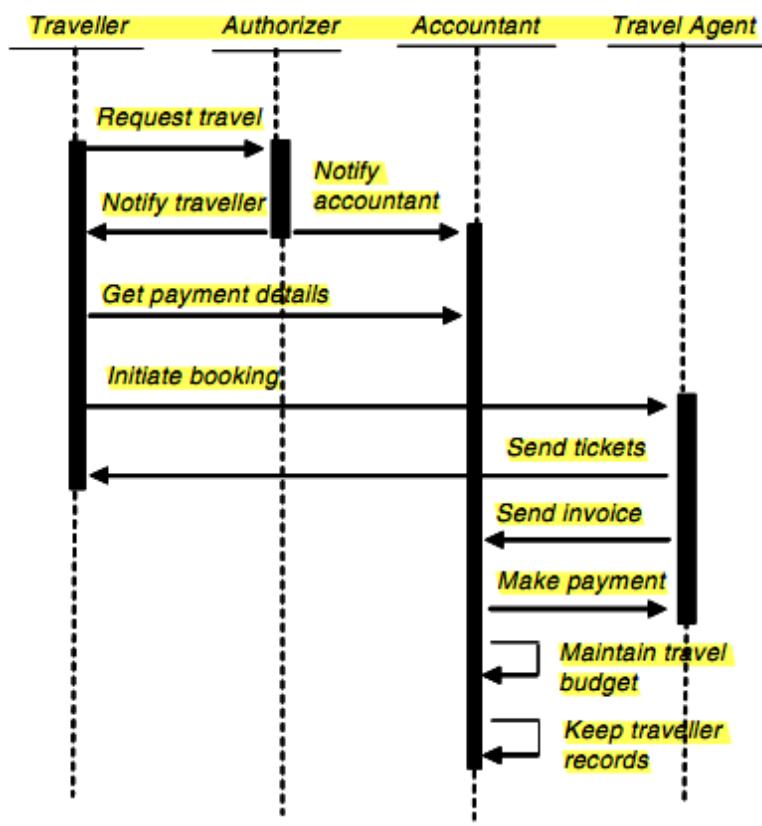


Fig. 9. An interaction diagram showing the ordering of responsibilities and collaborations, where solid bars representing the period in which roles are active and dotted lines representing the period when roles are inactive

holistic view of the roles in a particular role model. It shows both internal and external views of a role. The internal view of a role is characterized by its responsibilities and the external view is characterized by its collaborations with other roles.

5 Conclusion

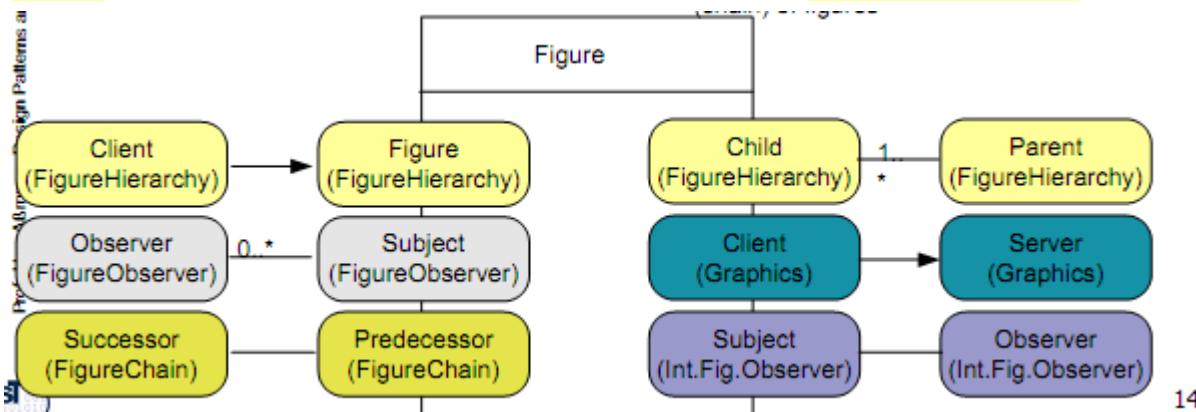
This article attempts to make a claim that the class concept is static, inflexible and not suitable for object design; in contrast the role concept is dynamic, flexible and central

A **role** is a dynamic view onto an object.

Roles are *services of an object in a context*

- Roles can be connected to each other, just as services are connected to client requests

Roles are *founded*, i.e., tied to *collaborations* and form *role models*



A role model specifies roles of objects in context, i.e., in a specific scenario.

A **role type (ability)** is a *service type of an object*

A role model is more *platform independent* than a class model

- The decision which **roles** are merged into which **classes** has not been taken and can be reversed
- We say: **roles are logical**, **classes are physical**

In MDA, role models are found on a more platform independent level than class models

- First design a set of role models
- Then find a class model by mapping roles into classes
- Respect role constraints

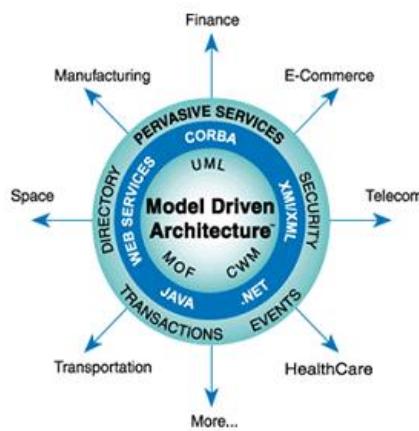
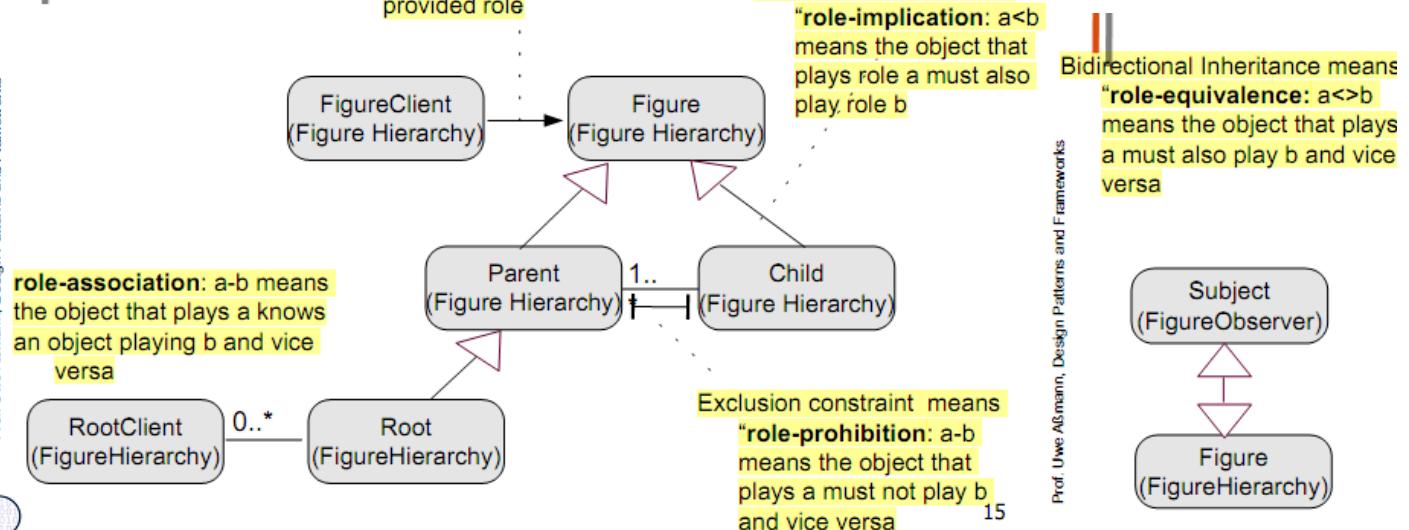


Fig. 3. Model Drive Architecture (from www.omg.com) consists of models organized into three layers. The outer layer is made of computation independent models (CIM); the middle layer is platform independent models (PIM); the inner layer is platform specific models (PSM).

Role Constraints in Role Models

- Arrows denote constraints between roles (role constraints)



Classes combine roles

- Classes are composed of role types
- Roles are dynamic items; classes are static items
- So, classes group roles to form objects

Class models combine role models

- Class models are composed of role models
- The sub-role-models of a composed role model are called its dimensions

Steps In Role-Based Design

- ▶ First, do role models
 - Roles are all kept distinct
 - Find out about role constraints that constraint which objects execute which roles
- ▶ Secondly, compose (merge) them
 - And set up new constraints between roles of different models
- ▶ Thirdly, map role models to class diagram

Class Model



The Difference of Roles and Facets

Facad (не может меняться - мужчина, женщина) vs role (может меняться)

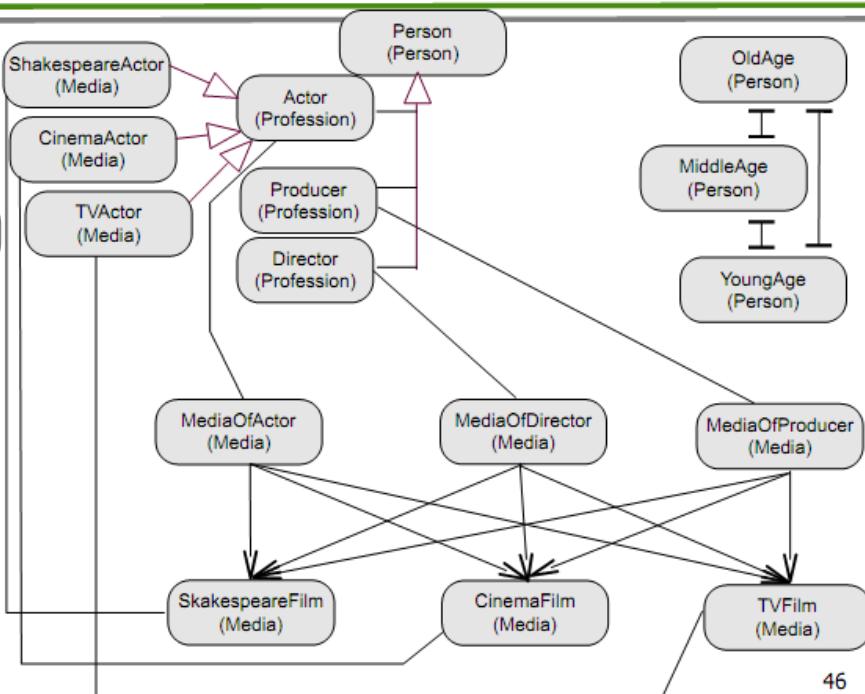
- Each facet is a role type
- Role types are independent of each other
- However, the role type is static, not dynamic: facets are lasting

Actors, Films, and Directors

- ▶ We model actors, directors, producers, and their films
- ▶ Actors have a genre (lover, serious, comedian) and play on a certain media (TV, cinema, Shakespeare)
- ▶ Directors and producers have similar attributes
- ▶ Films also
- ▶ Actors have an age (young, medium, old)

Example Role Model for Actors

Prof. Uwe Aschenbrenner

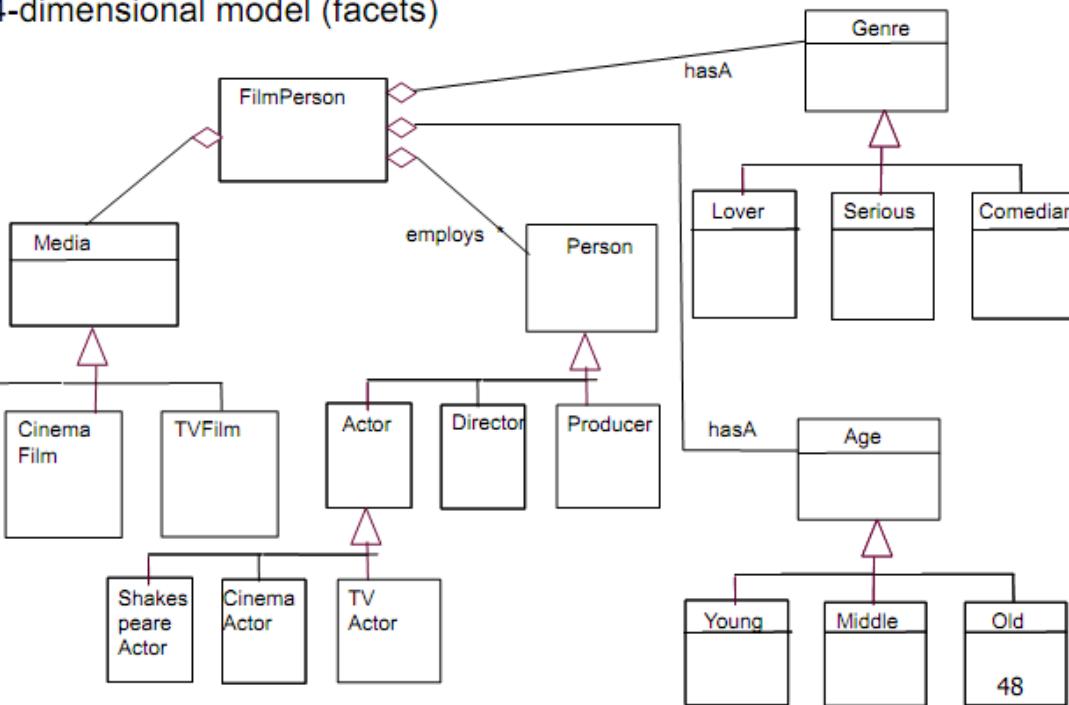


46

Very Simple Class Model for Actors and Films

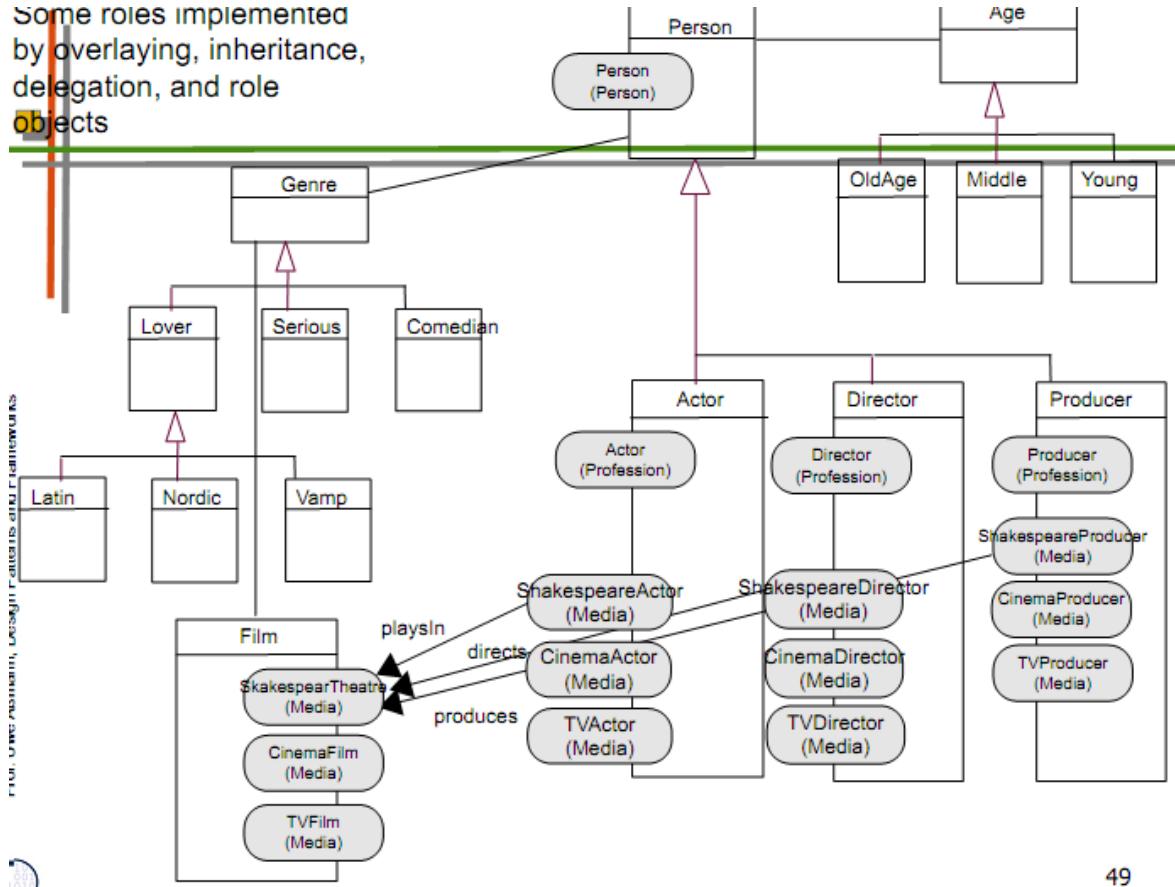
- ▶ 4-dimensional model (facets)

Prof. Uwe Aschenbrenner
Design Patterns and Frameworks



48

Some roles implemented by overlaying, inheritance, delegation, and role objects



49

Design Patterns have Role Models

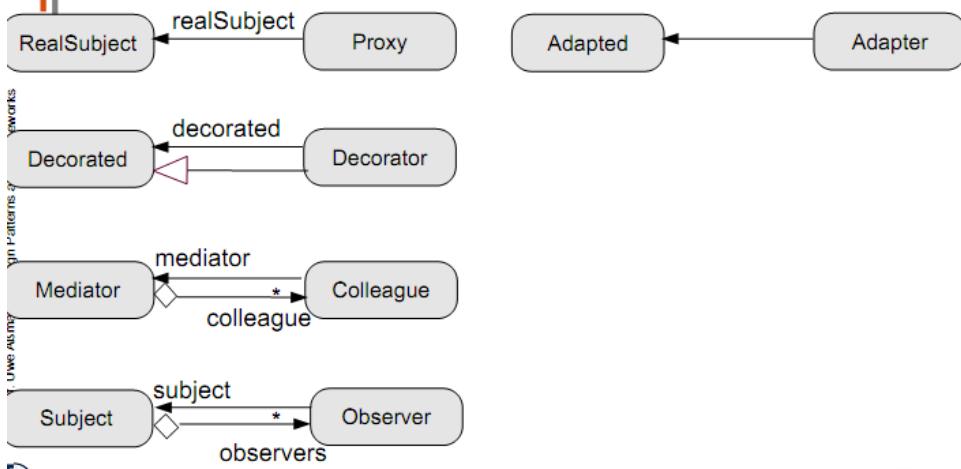
Observer role model



Patterns

- ▶ Many of them are quite similar

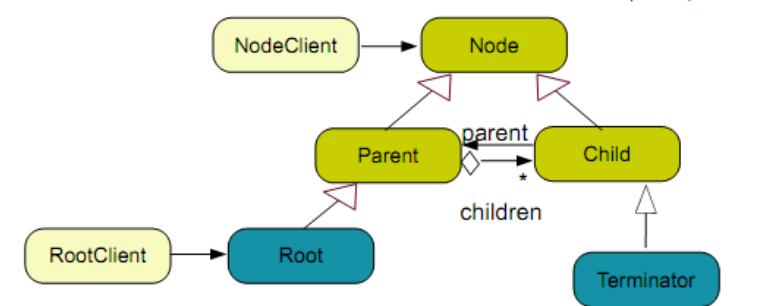
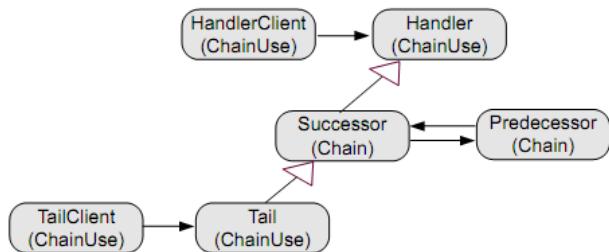
Uwe Aspray
Java Patterns



55

- ▶ Roles of Chain of Responsibility:

- Chain: (successor, predecessor)
- ChainUse: (Handler, HandlerClient, Tail, TailClient)



Roles vs Facets

- ▶ A facet is concerned always with one logical object
 - A facet classification is a *product lattice*
- ▶ Role models may *crosscut many objects*
 - They are concerned with collaboration of at least 2 objects

Rigid Types

If an object that has a (*semantically*) *rigid* type, it cannot stop being of the type without losing its identity

- ▶ Example:
 - A Book is a rigid type.
 - A Reader is a non-rigid type
 - A Reader can stop reading, but a Book stays a Book
- ▶ Semantically rigid types are *tied to the identity of objects*

Founded Types

- ▶ A *founded type* is a type if an object of the type is always in collaboration (association) with another object.
 - Example: Reader is a founded type because for being a reader, one has to have a book.

Natural types are non-founded and semantically rigid.

Book is a natural type.

A natural type is *independent* of a relationship

The objects cannot leave it

Law of Optimization for Design Patterns

Whenever you need a variant of a design pattern that is more efficient, investigate its role model and try to merge the classes of the roles

- ▶ Effect:
 - Less variability
 - Less runtime objects
 - Less delegations

The End: Summary

- ▶ Roles are important for design patterns
 - If a design pattern occurs in an application, some class of the application plays the role of a class in the pattern
 - Roles are dynamic classes: they change over time
 - ▶ Role-based modelling is more general and finer-grained than class-based modelling
 - ▶ Role mapping is the process of allocating roles to concrete implementation classes
-

Субъектно-ориентированное программирование

Субъектно-ориентированное программирование — метод построения **объектно-ориентированных систем**, как **композиции субъектов**. В целом СОП включает:

- разбиение системы на субъекты;
- написание правил для их правильной композиции.

Субъект в СОП — это коллекция классов или фрагментов классов, представляющих свою (субъективную) иерархию классов. Субъектом может быть само приложение, либо часть приложения, объединение которой с другими субъектами даёт приложение целиком. Композиция субъектов комбинирует иерархию классов так, что получаются новые субъекты, включающие функциональность существующих субъектов.

Сравнение с ООП

- **Объекту** необходимо конкретно указать, какие он должен выполнить методы, чтобы достичь результата.
- Субъекту необходимо конкретно указать какого результата необходимо ему достичь, а субъект сам выбирает методы, позволяющие это сделать.

Итак, Субъектом называется приложение, способное в рамках системы самостоятельно реализовывать задачу, имеющую несколько путей решения. В отличие от "неразумного" Объекта, Субъект наделен возможностью выбирать этот путь, то есть он способен сам корректировать последовательности своих действий для достижения поставленной цели

OBJECT RECURSION

Object Recursion transparently enables a request to be repeatedly broken into smaller parts that are easier to handle. Also Known As **Recursive Delegation**.

Motivation

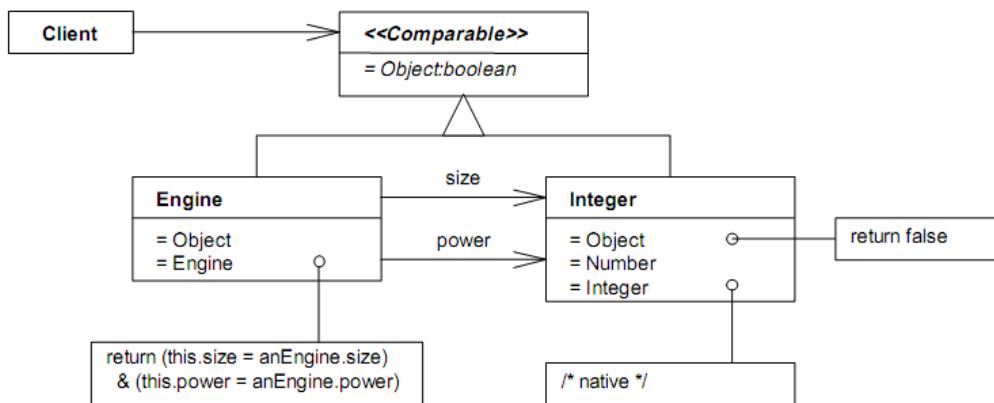
Consider the need to determine if two objects are equivalent. Simple objects and primitives are easy to compare; just use native operations. The difficulty lies in comparing arbitrarily complex objects.

One approach is to employ a Comparer object that accepts any two arbitrarily complex objects and answers whether the subjects are equivalent. The Comparer takes the subjects, breaks each one into pieces, and compares the pieces to determine if they're equivalent. If any of the pieces are too complex to compare, the Comparer repeats the process by breaking it into pieces, and so on until all of the pieces are simple enough to compare.

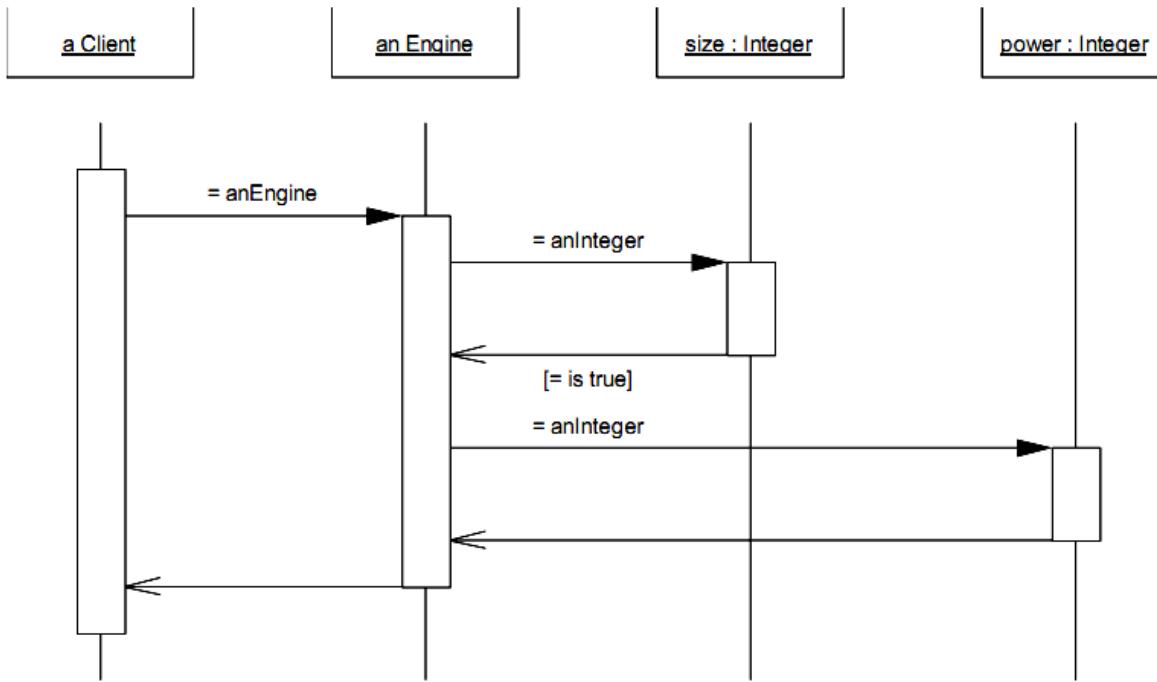
A more object-oriented approach is for the Comparer to tell the subjects to compare themselves and let them decide how to do that. This way, the Comparer is telling the subjects *what* to do, but not *how* to do it. With this approach, the Comparer object isn't even needed because any Client can simply ask one subject to compare itself with another.

So how do the subjects compare themselves? One determines if the other is equivalent to itself. It considers which parts of its state must be equivalent, then compares those. Each of those parts considers which of its parts must be equivalent and compares those, and so on until all of the parts are simple objects. At each step, the comparison process is relatively simple. Even a highly complex object doesn't need to know how to compare itself entirely; it just needs to know what its parts are and they need to know how to compare themselves.

For example, consider an `Engine` object that knows its internal displacement and total horsepower. To compare it to another engine, a client must verify that both engines are the same size and same power. This diagram shows the classes involved and how to implement =:



The Client sends = to the first Engine with the second Engine as an argument. The first Engine compares itself to the second one by comparing their size and power. The parts are Integers, simple objects that the native system can compare. If the parts were complex objects, they would continue the comparison process by comparing their parts. Eventually, the simplest parts are either equal or they're not.



Keys

A system that incorporates the Object Recursion pattern has the following features:

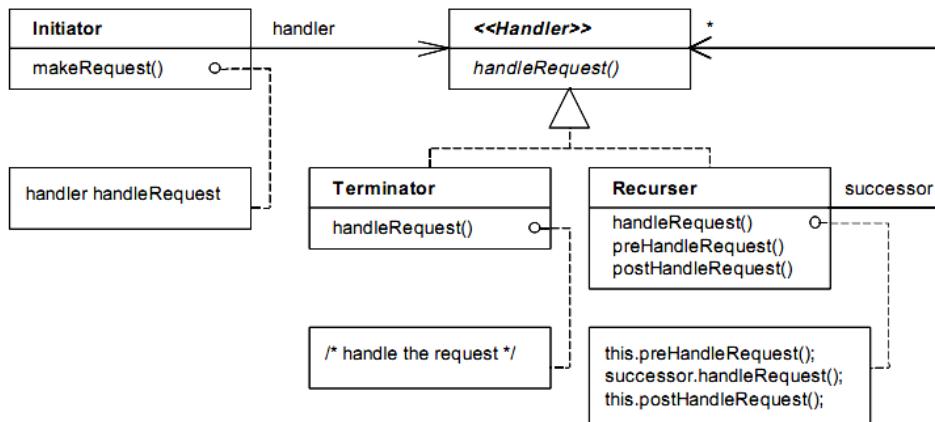
- Two polymorphic classes, one of which handles a request recursively and another which simply handles the request without recursing.

Applicability

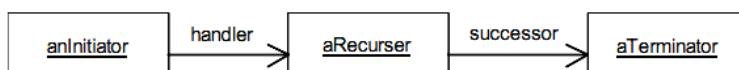
- passing a message through a linked structure where the ultimate destination is unknown.
- broadcasting a message to all nodes in part of a linked structure.
- distributing a behavior's responsibility throughout a linked structure.

Structure

The following class diagram shows the roles of the participants:



A typical object structure might look like this:



Participants

- **Initiator (Client)**
 - initiates the request.
- **Handler (Comparable)**
 - defines a type that can handle requests that initiators make.
- **Recruser (Engine)**
 - defines the successor link.
 - handles a request by delegating it to its successors.
- **Terminator (Integer)**
 - finishes the request by implementing it completely

Collaboration

- The Initiator needs to make a request. It asks its Handler to handle the request.
- When the Handler is a Recruser, it does whatever work it needs to do, asks its successor—another Handler—to handle the request, and returns a result based on the successor's result. The extra work can be done before and/or after delegating to the successor. If the Recruser has multiple successors, it delegates to each of them in turn, perhaps asynchronously.
- When the Handler is a Terminator, it handles the request without delegating the request to any other successors and returns the result (if any).

The advantages of the Object Recursion pattern are:

- *Distributed processing.* The processing of the request is distributed across a structure of handlers that can be as numerous and arranged as complexly as necessary to best complete the task.

Sample Code

Let's look at how to implement equals recursively.

All objects, no matter how complex, are ultimately composed of simple objects (i.e., primitives) such as integers, floats, booleans, and characters. Determining the equality of two simple objects (of the same type) is a trivial task performed natively by the operating system or CPU. For example:

```
5 == 5          // integer comparison (true)
5.25 == 5.15   // float comparison (false)
true == false   // boolean comparison (false)
'a' == 'b'     // character comparison (false)
```

There's no recursion here, but these simple comparisons form the terminating case for recursion.

Comparing two ordered collections is nearly as simple: Are each of the elements equal? Thus two strings are equal if each of their characters are equal. For example, look at the implementation of

Thus if each of the characters in the strings are equal, the strings are equal. For the purposes of implementing equality recursively, a string is a terminating object.

Object Recursion vs. Composite and Decorator

When a Decorator [GHJV95, p. 175] delegates to its component or a Composite [GHJV95, p. 163] delegates to its children, this might be considered an example of Object Recursion. However, Composite and Decorator are structural (data structure) patterns, whereas Object Recursion is a behavioral (algorithm) pattern. If the structural patterns do embody recursion, it is only explicitly one

Object Recursion vs. Chain of Responsibility

Chain of Responsibility [GHJV95, p. 223] contains the Object Recursion pattern. Chain of Responsibility uses a linked-list or tree organized by specialization or priority. When a request is made, the structure uses Object Recursion to find an appropriate handler.

Schizophrenia

“Object Schizophrenia results when the state and/or behavior of what is intended to appear as a single object are actually broken into several objects (each of which has its own object identity).”

[Subject-oriented programming](#) as a solution, which by static composition avoids any issues of object schizophrenia. Role object pattern also avoids object schizophrenia problem.

An Object is said to be under the **Schizophrenia** State if it has one or more of the following:

- 1) Its **Interface** is an aggregate of several objects interfaces; known as Broken Interface
- 2) Its **Identity** is an aggregate of several objects Identities (mainly by Inheritance)
- 3) **Broken State**, which happened when the state of an object is composed from different objects states.

Instead of using one class to refine another, delegation allows one object to override the behavior of another. The original object *a* (analogous to the base class behaviors) can *delegate* some of its methods to another object *b* (analogous to the derived class behaviors). If *a* delegates its *foo* method to the *bar* method of *b*, then any invocation of *foo* on *a* will cause *b*'s *bar* method to execute. However, *bar* executes in the context of the *a* object — for example, its *self* identifier refers to *a* rather than to *b*.

When delegation is used, the question arises: What is the identity of the object *a*? The identity is split. There are two potentially meaningful *self* values when executing a method of *a*.

~~item or object-oriented programming. Object schizophrenia appears together with internal delegation, i.e., when a logical object is split into two physical ones. Such a split destroys the uniqueness of self/this, and if clients of the logical object rely on the fact that the object should be unique and complete, calls to the object may fail [Szyperski, 1998]. Hence, object~~

```

class A {
    void foo() {
        // "this" also known under the names "current", "me" and "self" in other languages
        this.bar();
    }

    void bar() {
        print("a.bar");
    }
};

class B {
    private A a; // delegation link
    public B(A a)
    {
        this.a = a;
    }
    void foo() {
        a.foo(); // call foo() on the a-instance
    }
    void bar() {
        print("b.bar");
    }
};
a = new A();
b = new B(a); // establish delegation between two objects
b.foo()

```

Patterns and Frameworks

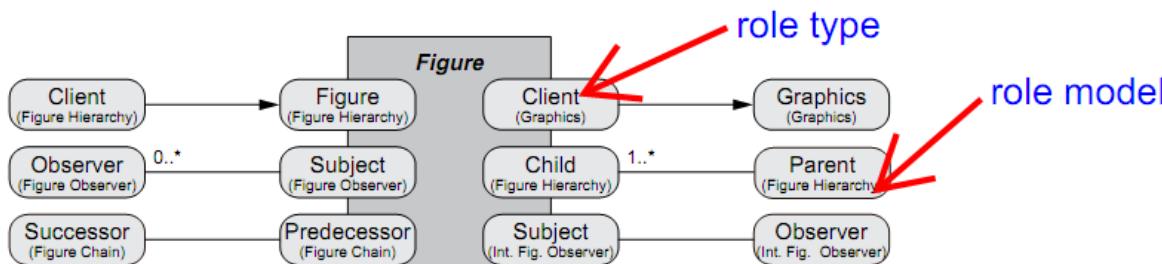
Design patterns are building blocks of frameworks

Today, any large object-oriented software system is built using frameworks.

- *Unanticipated use-contexts.* Sometimes, a framework must be prepared for extension into unforeseen contexts, e.g., because client requirements cannot be determined in advance. The ability of a framework to be extended in such a case is crucial for its successful reuse. Thus, a

An oval depicts a role type,
with the role model in which it is
defined set below in a small font.

A rectangle depicts a class,
with its role type set defined by
the role types put on top of it.



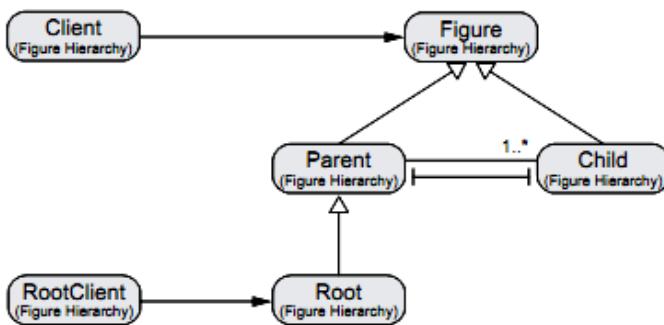
An arrow depicts
a directed use-
relationship.

A star depicts
an unlimited
cardinality.

A line depicts
a bidirectional
use-relationship.

Role model-concern. It focuses on a single purpose of object collaboration:

the role type Figure from the FigureHierarchy role model is
fully qualified as FigureHierarchy.Figure, thereby distinguishing it from the class Figure. Every role type is unique.



An object playing the Parent role may maintain several objects, each playing the Child role. (A Parent object maintains several Child objects). An object playing the Root role also always the Parent role.

- For a role type pair (A, B), a *role-dontcare* value states that there are no constraints on an object playing any of the roles defined by the role types A and B.
- For a role type pair (A, B), a *role-implied* value states that an object playing a role defined by role type A must always be able to play a role defined by role type B (but not necessarily the other way).
- For a role type pair (A, B), a *role-equivalent* value well as for (B, A). (I.e., both roles are always available together.)
- For a role type pair (A, B), a *role-prohibited* value states that an object playing a role defined by role type A never plays a role defined by role type B and vice versa.

Role constraints are constraints on object collaborations.

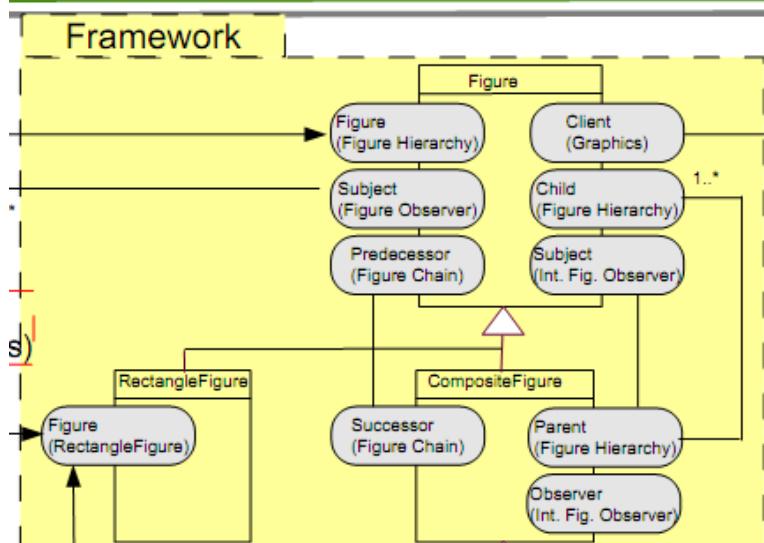
Role constraints are always scoped by the role model, for

A **class model** is a set of classes that relate to each other using class inheritance. Classes combine role types, and class models combine role models.

A **framework** is a class model, together with an integration role type set, and a class set. An **integration role type set** contains those role types of the class model, which have not been assigned to classes.

Framework Instantiation with Open Roles

Open role hooks (free, unbound abilities) are role types that have not yet been assigned to classes. A framework is instantiated by binding its integration repertoire (role type set) to classes.



Merging of Frameworks

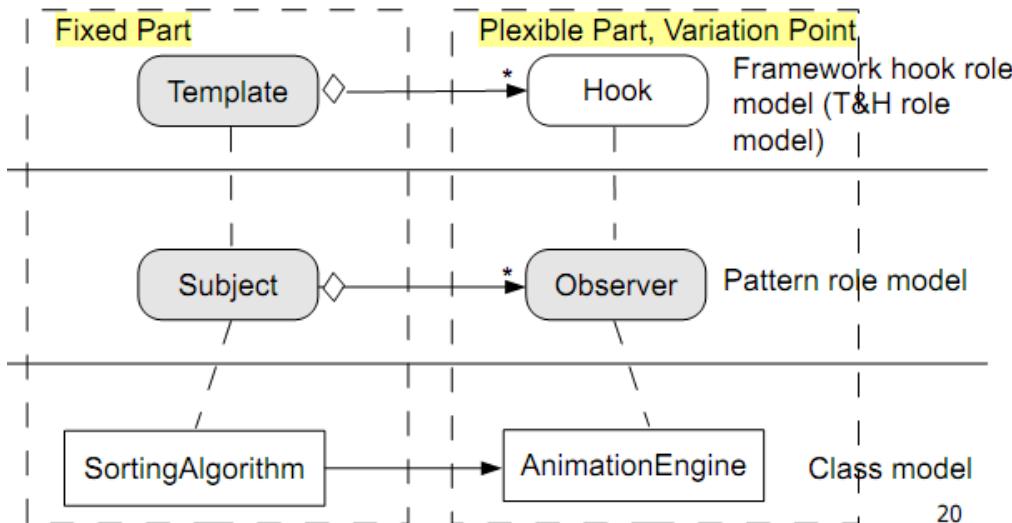
- ▶ Two frameworks are *merged* by binding the integration abilities of A to classes of B
 - Role constraints have to be respected

[Riehle/Gross] role-based framework instantiation relies on simple role binding, with role constraints

- ▶ A T&H role model has 2 parts:
 - A template class (or *template role type*), which gives the skeleton algorithm of the framework: Fix, grasps commonalities
 - A hook class, which can be exchanged (or: a *hook role type* which can be bound to a client class): Variable, even extensible, grasps variability and extension



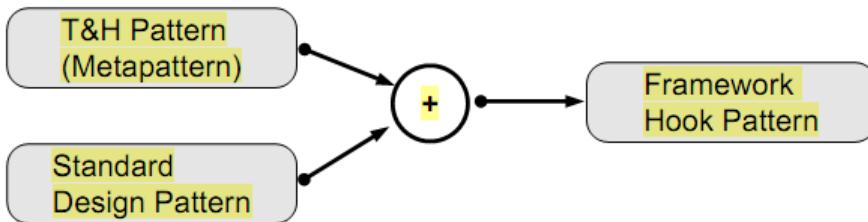
- The template part fixes parts of the pattern
- The hook part keeps parts of the pattern variable, i.e., open for binding.



20

Metapatterns are very general role models that can be mixed into every design pattern

As design patterns describe application models, metapatterns describe design patterns



Framework Hook Patterns

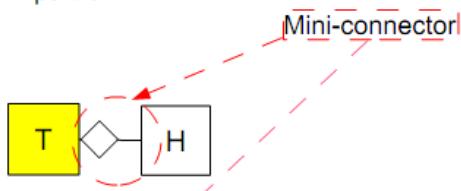
- ▶ The template-hook role model
 - adds more pragmatics to a standard design pattern, information about commonality and variability. Hence, framework variation points are described
 - The template-hook role model adds more *constraints* to a standard design pattern. Some things can no longer be exchanged

Differences between Standard Patterns and Framework Hook Patterns

- | | |
|---|---|
| <ul style="list-style-type: none"> ▶ Standard design pattern <ul style="list-style-type: none"> ▪ Often, no template parts; everything flows (exception: TemplateClass and -Method) ▪ Rich pattern and role model ▪ Applicable everywhere in the framework ▪ No T&H metapattern overlayed | <ul style="list-style-type: none"> ▶ Framework hook pattern <ul style="list-style-type: none"> ▪ Fixed and variable part ▪ Elementary pattern and role model ▪ Applicable only <i>at the border</i> of the framework, ▪ or at the border of a component, i.e., in an "interface" ▪ One T&H metapattern overlayed |
|---|---|

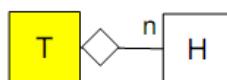
1-T—H (open role hook)

H part of T

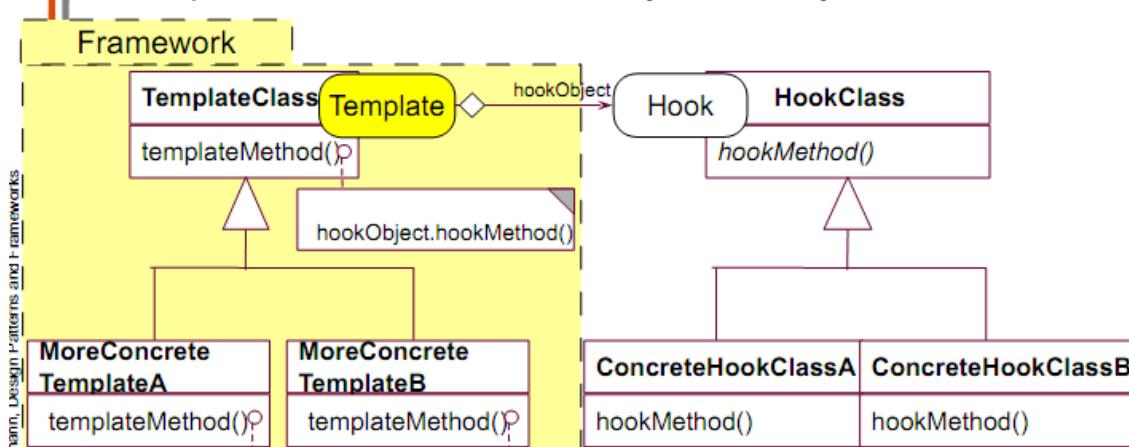


n-T—H (flat extension)

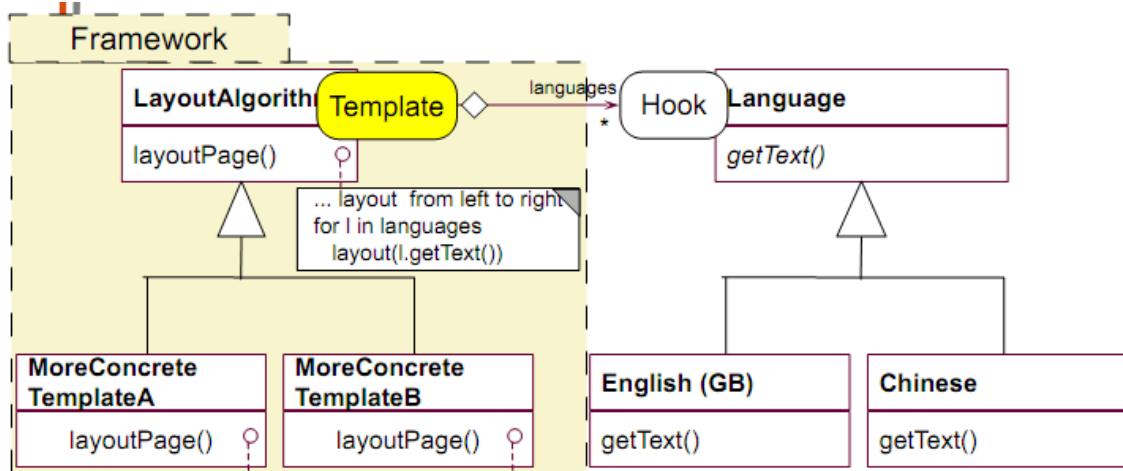
T has n H parts, n is dynamic



Bridge with 1-T--H



n-T—H is based on *-Bridge pattern

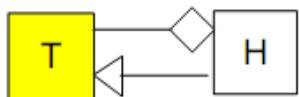


n-T—H Makes Bridge Frameworks Extensible

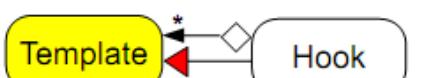
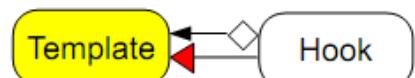
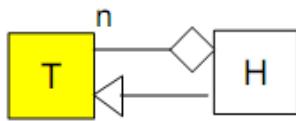
- An n-T—H framework hook makes dimensional bridge frameworks extensible with new dimensions *at run time*
- New extensions in new dimensions can be added and removed on-the-fly
- 1-T—H gives variability
- n-T—H gives extensibility

11.4 The H<=T Recursion Metapattern

decorator
H<=T (deep list extension)
 T part of H
 H inherit from T
 1-ObjectRecursion/Decorator

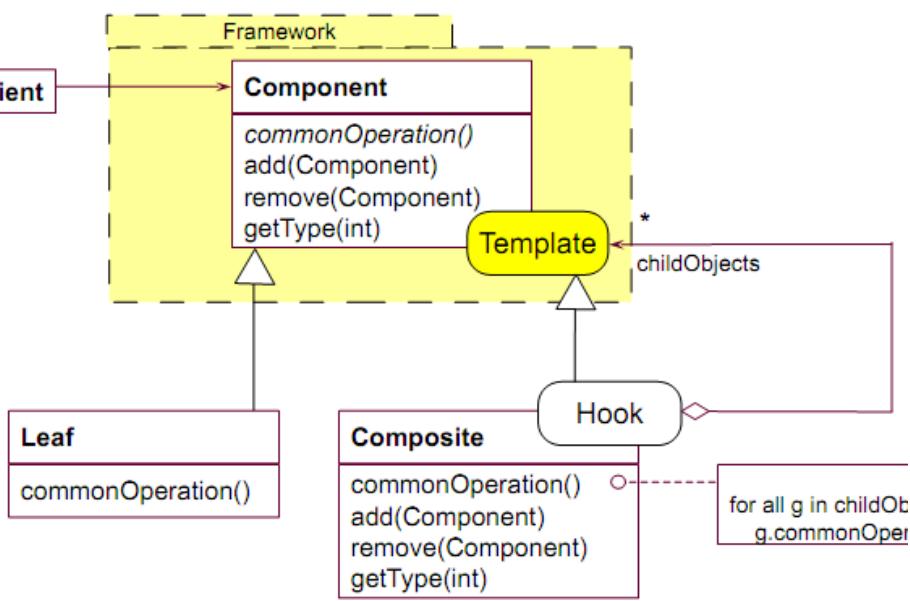


composite
n-H<=T (deep graph extension)
 H has n T parts
 T inherit from H
 n-ObjectRecursion/Composite



48

Composite as n-H<=T

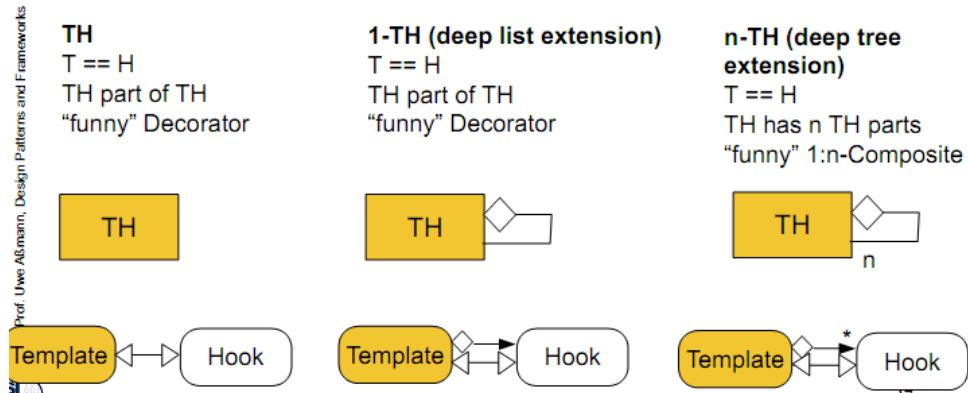


H<=T framework hooks result in frameworks between black-box and white-box

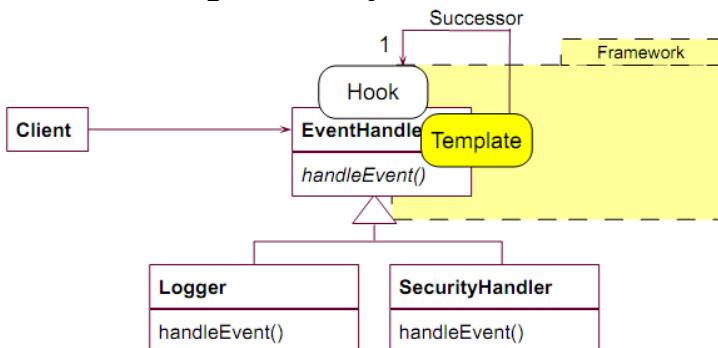
to есть gray-box

Unified T&H pattern (TH framework hook)

T-class == H-class



ChainOfResponsibility as 1-TH



A H<T framework hook means whitebox framework



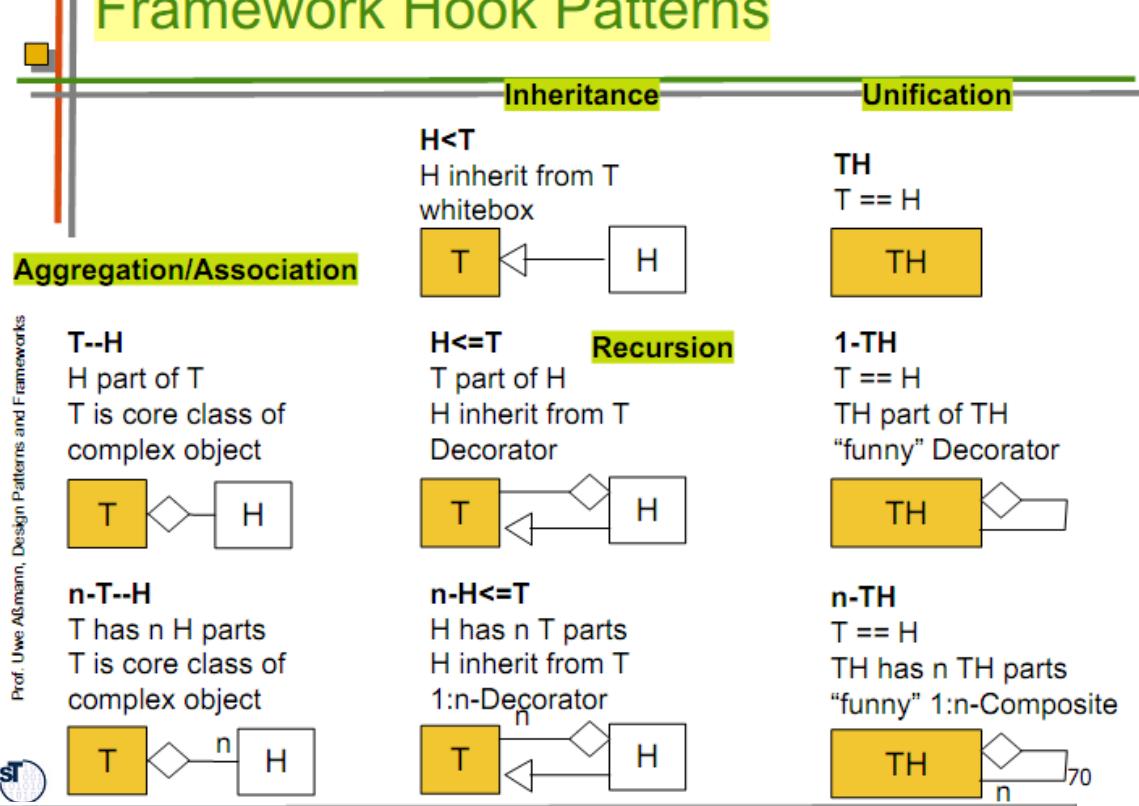
1:1 – T and H correspond 1:1

- T has 1 H part
- Hooks are not extensible at runtime
- 1:1 T&H framework hooks should be used when the behavior of the framework should be varied, but not extended at the variation point

1:n – T and H correspond 1:n

- T has n H parts
- Hooks are extensible, also dynamically
- 1:n T&H framework hooks should be used when the behavior of the framework should not only be varied, but also *extended* dynamically at

Framework Hook Patterns

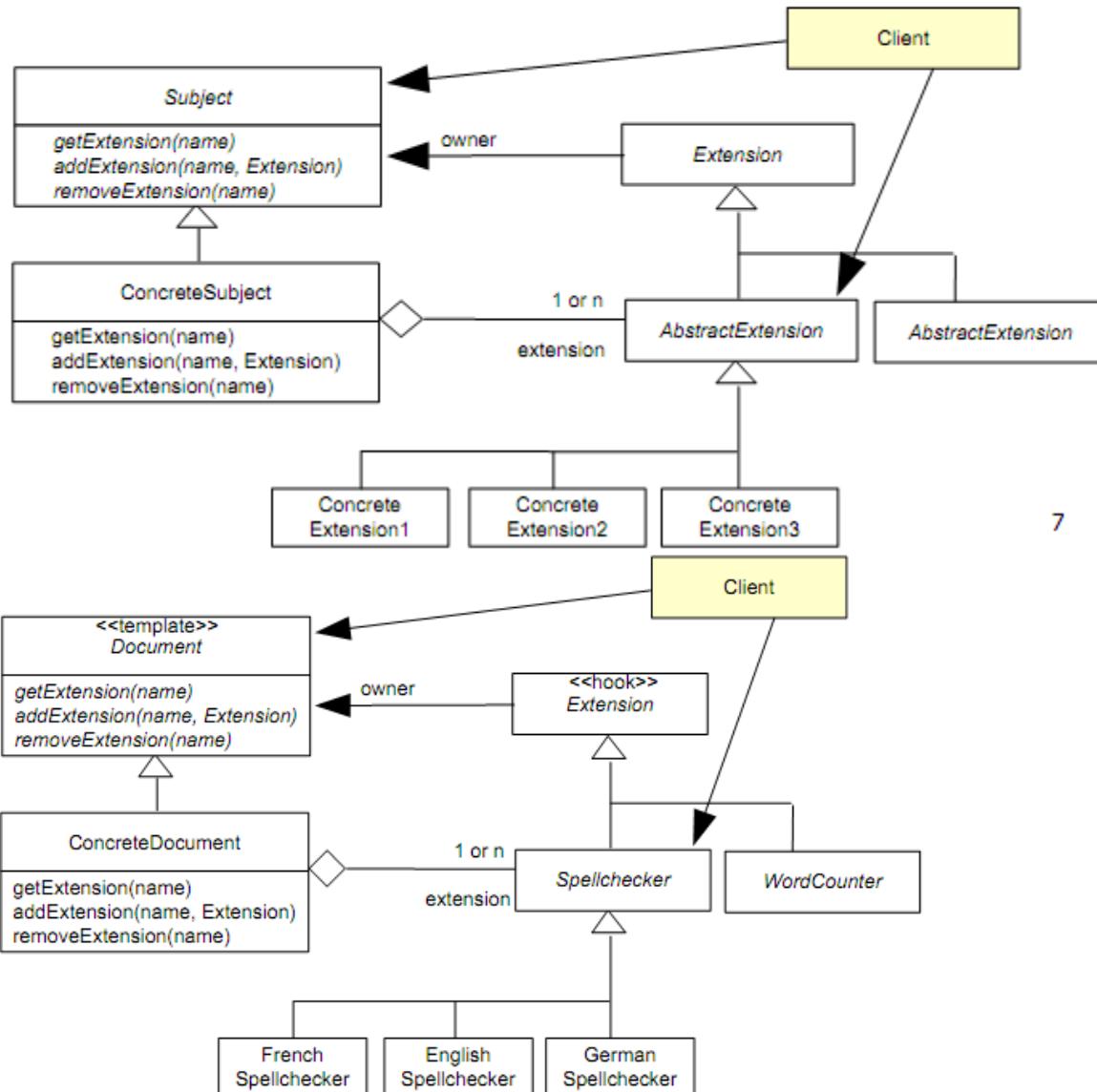


Today, frameworks are the most important software technology for product lines in large companies

Instantiating big frameworks is very hard

Extension Object Pattern

- Extension is the base class of all extensions
- AbstractExtension defines an interface for a concrete hierarchy of extension objects
- Extensions can be added, retrieved, and removed by clients



7

8

Advantages:

Extensions can be added dynamically and unforeseen

Disadvantage

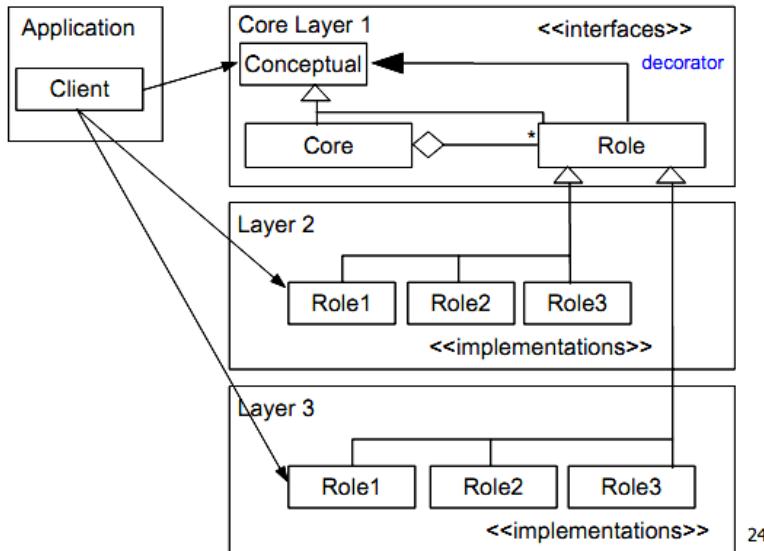
- Clients have to manage extension objects themselves, and hence, are more complex
- Extension objects suffer from the *object schizophrenia* problem: the

The Role-Object Pattern (ROP)

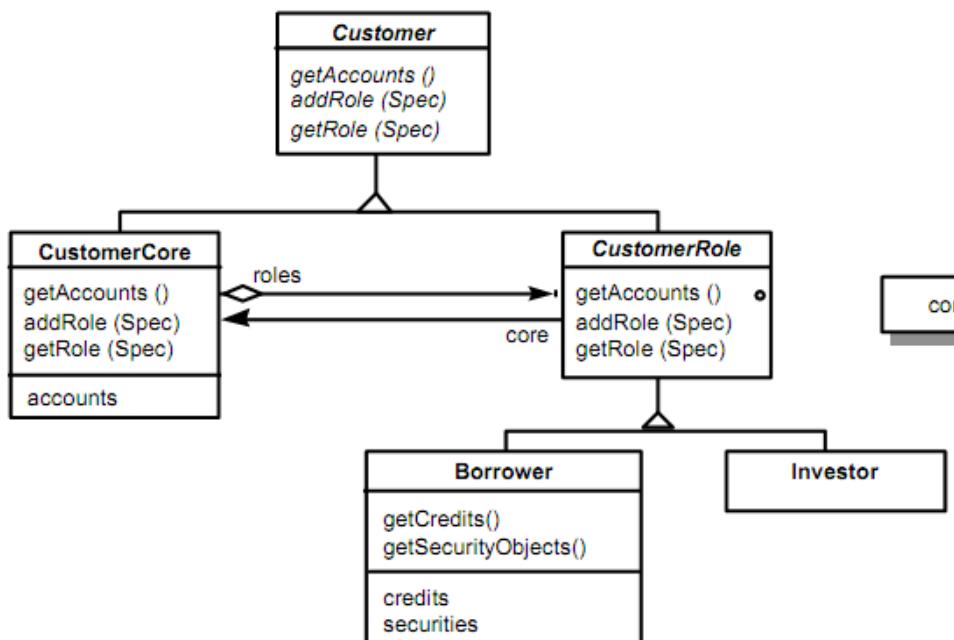
The Role-Object Pattern (ROP) is both a variability and extensibility pattern.

- Can easily be extended with new layers
- A **conceptual object** - complex object, **subject**
- **Core and role objects** conceptually belong together to the conceptual object
- The core knows all RoleObjects, and distributes requests (Mediator)
- Role objects decorate the core object, and pass requests on to it

A **subject** often plays several roles and the same role is likely to be played by different subjects. As an example consider two different customers playing the role of borrower and investor, respectively. Both roles could as well be played by a single Customer object.



24



- **Customer** is defined as an abstract superclass. It serves as a pure interface.
 - The **CustomerCore** subclass implements the Customer interface.
 - **CustomerRole** also supports the Customer interface. The **CustomerRole** class is an abstract class.
 - **Borrower class** defines additional operations to manage the customer's credits and securities.
- Similarly, the Investor class adds operations specific to the investment department's view of customers.

Don't use this pattern

- if your potential roles have strong interdependencies.

Participants:

- **Component (Customer)** - specifies the protocol for adding, removing, testing and querying for role objects.
- **ComponentCore (CustomerCore)**
 - implements the Component interface
 - creates ConcreteRole instances;
- **ComponentRole (CustomerRole)**
 - stores a reference to the decorated ComponentCore;
- **ConcreteRole (Investor, Borrower)**
 - models and implements a context-specific extension of the Component interface;

COLLABORATIONS

Core and role objects collaborate as follows:

- ComponentRole forwards requests to its ComponentCore object.
- ComponentCore instantiates and manages ConcreteRoles.

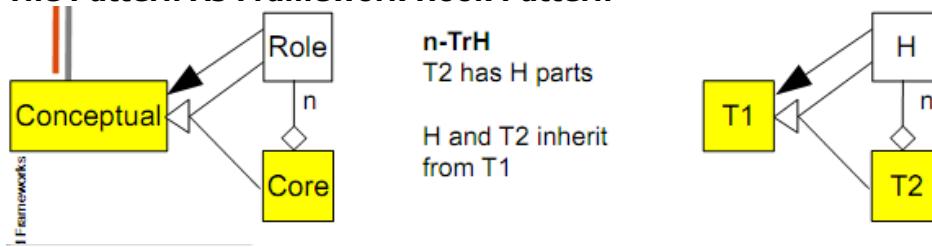
Advantages:

- Roles can be evolved easily and independently of each other. A ConcreteRole class lets you add new roles
- Roles objects can be added and removed dynamically. A role object can be added and removed at runtime, simply by attaching it to and detaching it from the core object.

Disadvantage:

Maintaining constraints between roles becomes difficult. Since a subject consists of several objects which are mutually dependent, maintaining constraints and preserving the overall subject consistency might become difficult.

The Pattern As Framework Hook Pattern



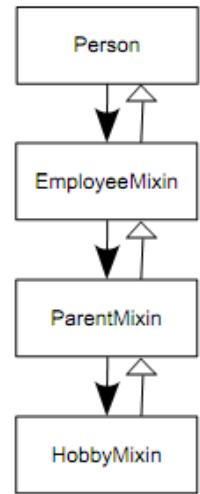
- ▶ The RoleObject pattern lends itself not only to variability, but also to static and dynamic extensibility
 - If a framework hook is a role object pattern, the hook can be extended in unforeseen ways **without** changing the framework!
 - New layers of the application or the framework can be added at design time or runtime
- ▶ Powerful extension concept
 - Whenever you have to design something complex which should be extensible in unforeseen ways, consider Role Object
- Multi-Bridge suffers from object schizophrenia, ROP can implement "this()" on itself without object schizophrenia

The GenVoca Pattern

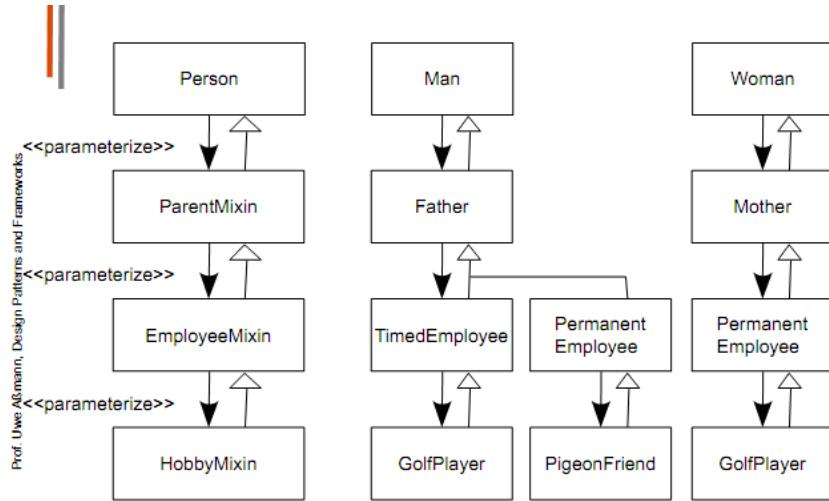
A Mixin is a partial class, for an extension of another class

If several mixin parameterizations are nested, the GenVoca pattern results [Batory]

```
// Persons composed with GenVoca pattern
HobbyMixin<ParentMixin<EmployeeMixin<Person>>> assmann;
EmployeeMixin<ParentMixin<HobbyMixin<Person>>> assmann2;
```



GenVoca Variations



```
// Variants
Person: Man, Woman
ParentMixin: FatherMixin, MotherMixin
EmployeeMixin: TimedEmployee, PermanentEmployee
HobbyMixin: PigeonFriend, Sportsman, GolfPlayer

// Compositions
GolfPlayer<TimedEmployee<Father<Man>>> assmann;
PigeonFriend<PermanentEmployee<Father<Man>>> miller;
GolfPlayer<PermanentEmployee<Mother<Woman>>> brown;
```

- GenVoca applications are more efficient, since they merge all roles together into one physical object (see the Aßmann's law on role merging)

How can we structure a Product Line as Layered Framework?

- ExtensionObjects is a simple extension mechanism for frameworks
- Layered frameworks provide variability and extensibility for thousands of different products in a product line

Process for layered frameworks:

- Identify concerns (abstraction layers), which crosscut all or many objects. These concerns are similar to facets, but not independent
- Sort them according to their (acyclic) dependencies
- Use ROP or GenVoca pattern for implementation
- Use framework role layers or mixin layers for a layered application

The Tools And Materials

- ▶ People use tools in their everyday work
 - Tools are *means of work*
- ▶ People use tools to work on material
- Tools and materials are in relation

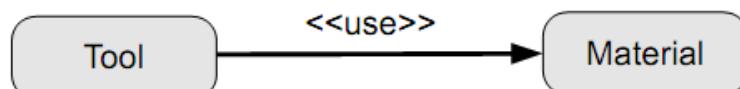
Material

- Passive entities (entries in a database, items in a worklist)
- Transformed and modified during the work
- Not directly accessible, only via tools

Tools

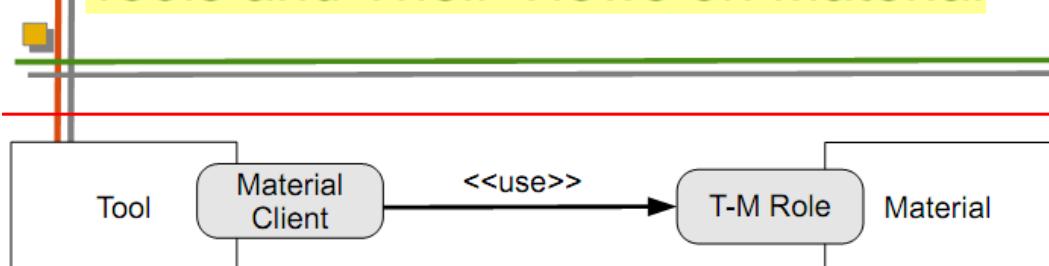
Active entities (Visible on the desktop as wizards, active forms)

- ▶ To say, what is a tool and what the material, depends a lot on the concrete task (interpretation freedom)
 - Pencil -- paper
 - Pencil sharpener - pencil



- ▶ A *tool-material collaboration* (T&M role model, T&M access aspect) expresses the relation of a tool and the material
- Roles of a material define the necessary operations on a material for one specific task
 - ! They reflect usability: how can a material be used?
 - ! Express a tool's individual needs on a material

Tools and Their Views on Material



- Tools have one functional part and one or several interaction part

Functional Part:

- Manipulation of the material
- Access to Material via material-roles

Interaction Part:

- Reactive on user inputs

The environment initializes everything, displays everything on the desktop, and waits for tool launch

T&M is a pattern language for constructing interactive applications

Eclipse

- ▶ Eclipse is a set of frameworks for development of
 - IDE applications
 - IDE (not only for Java)
 - GUI applications

Plugins (Extensions)

- ▶ Are classes that are dynamically loaded from a special directory `eclipse/plugins`
- ▶ Every plugin is represented by a *plugin class*,

San Francisco

What is San Francisco (SF)?

- ▶ Business framework of IBM, to support the building of business applications

Design goals

- flexibility by using object-oriented framework technology
- maximal reuse
- isolation from underlying technology
- focus on the core, provide the common tasks of every business application
- rapidly building quality applications
- integration with existing systems

Refactorings - Transform Antipatterns Into Design Patterns

- Refactoring based on Metaprogramming
- Role-based Generic Model Refactoring

Преимущества Посетителя

- Возможность добавления операций в составную структуру без изменения самой структуры.
- Добавление новых операций выполняется относительно просто.

Объектная шизофрения -- это когда 2 инстанса (объекта) работают вместе и по идеи должны представляться как один логический объект, т.е. указатель `this` должен был возвращать один и тот же объект (в C++ `this` должен возвращать указатель на один и тот же адрес в памяти). Но на самом деле так не происходит в основных языках программирования (C++, C#, Java), каждый объект имеет свой собственный `this` (по сути, идентификатор).

Возьми, например, Strategy паттерн. В нем один объект имеет ссылку на другой объект. Второй объект (подобъект) расширяет функции первого (основного) объекта, и по сути оба объекта представляют один логический.

Но когда ты в коде считаешь значение `this`, то у тебя они представляются как разные физические объекты. Это и есть шизофрения.