

Паттерны и архитектурные стили

Юрий Литвинов
y.litvinov@spbu.ru

14.07.2023

Паттерны проектирования

Шаблон проектирования — это повторяемая архитектурная конструкция, являющаяся решением некоторой типичной технической проблемы

- ▶ Подходит для класса проблем
- ▶ Обеспечивает переиспользуемость знаний
- ▶ Позволяет унифицировать терминологию
- ▶ В удобной для изучения форме
- ▶ НЕ конкретный рецепт или указания к действию

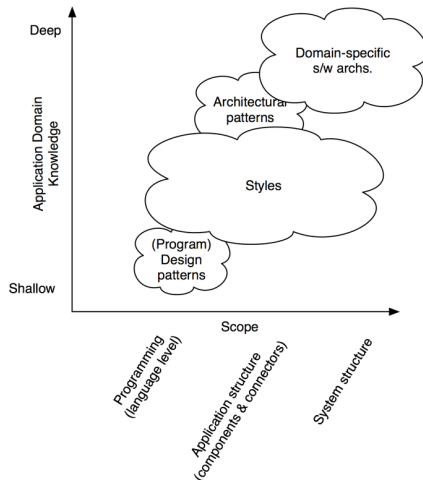
Архитектурные стили

Архитектурный стиль — набор решений, которые

1. применимы в выбранном контексте разработки,
2. задают ограничения на принимаемые архитектурные решения, специфичные для определённых систем в этом контексте,
3. приводят к желаемым положительным качествам получаемой системы.

Архитектурные шаблоны более «стратегичны» и более размыты, чем паттерны

Паттерны и архитектурные стили

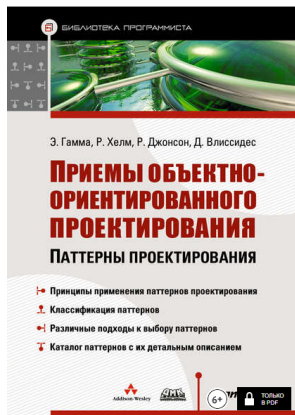


Книжка про паттерны

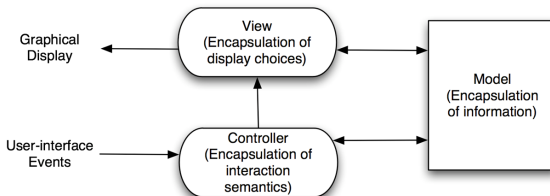
Приемы объектно-ориентированного проектирования. Паттерны проектирования

Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес

Design Patterns: Elements of Reusable Object-Oriented Software



Пример: Model-View-Controller



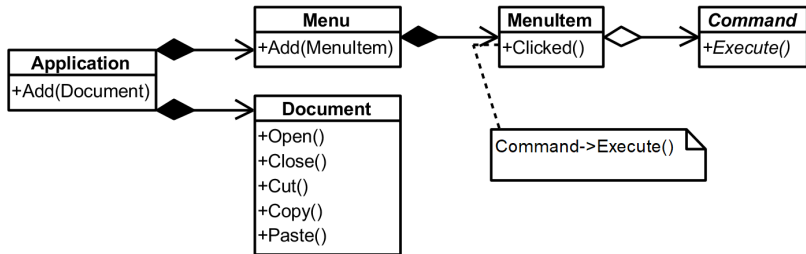
© N. Medvidovic

- ▶ Разделяет данные, представление и взаимодействие с пользователем
- ▶ Если в модели что-то меняется, она оповещает представление (представления)
- ▶ Через контроллер проходит всё взаимодействие с пользователем
 - ▶ Естественное место для паттерна «Команда» и Undo/Redo

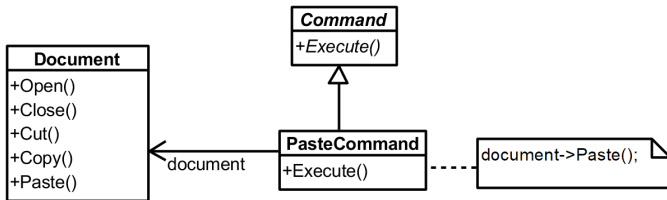
Паттерн «Команда», мотивация

- ▶ Хотим отделить инициацию запроса от его исполнения
- ▶ Хотим, чтобы тот, кто «активирует» запрос, не знал, как он исполняется
- ▶ При этом хотим, чтобы тот, кто знает, когда исполнится запрос, не знал, когда он будет активирован
- ▶ Но зачем?
 - ▶ Команды меню приложения
 - ▶ Палитры инструментов
 - ▶ ...
- ▶ Просто вызвать действие не получится, вызов функции жёстко свяжет инициатора и исполнителя

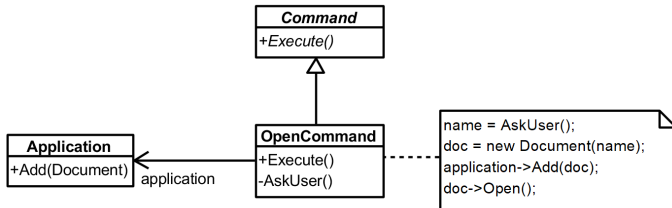
Решение: обернём действие в объект



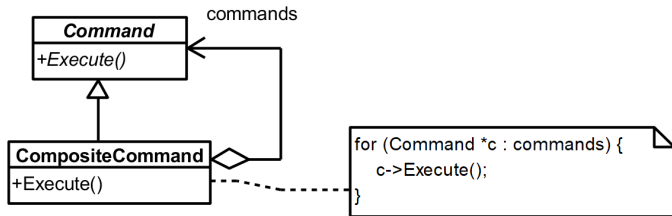
Команда вставки



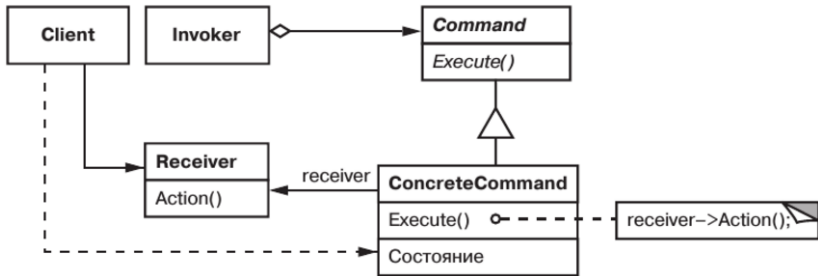
Команда открытия документа



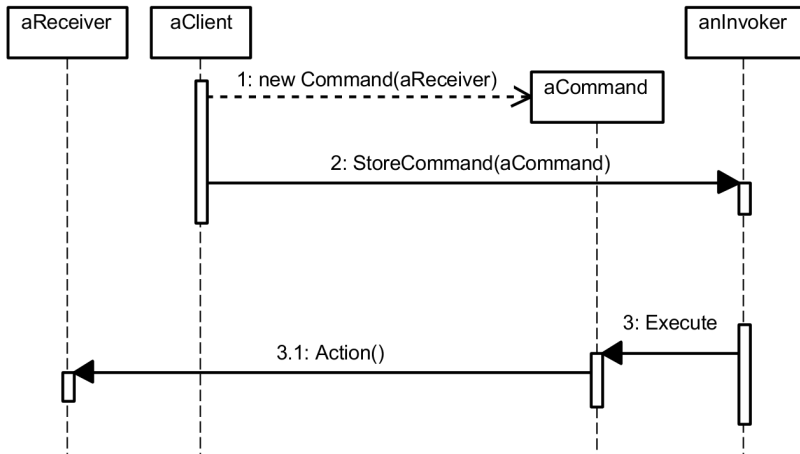
Составная команда



Паттерн «Команда»



Взаимодействие объектов



Команда, применимость

- ▶ Параметризовать объекты выполняемым действием
- ▶ Определять, ставить в очередь и выполнять запросы в разное время
- ▶ Поддерживать отмену операций
- ▶ Структурировать систему на основе высокоуровневых операций, построенных из примитивных
- ▶ Поддерживать протоколирование изменений

«Команда» (Command), детали реализации

- ▶ Насколько «умной» должна быть команда
- ▶ Отмена и повторение операций — тоже от хранения всего состояния в команде до «вычислимого» отката
 - ▶ Undo-стек и Redo-стек
 - ▶ Может потребоваться копировать команды
 - ▶ Искусственные команды
 - ▶ Композитные команды

«Команда», пример

- ▶ Qt, класс QAction:

```
const QIcon openIcon = QIcon(":/images/open.png");  
QAction *openAct = new QAction(openIcon, tr("&Open..."), this);
```

```
openAct->setShortcuts(QKeySequence::Open);  
openAct->setStatusTip(tr("Open an existing file"));
```

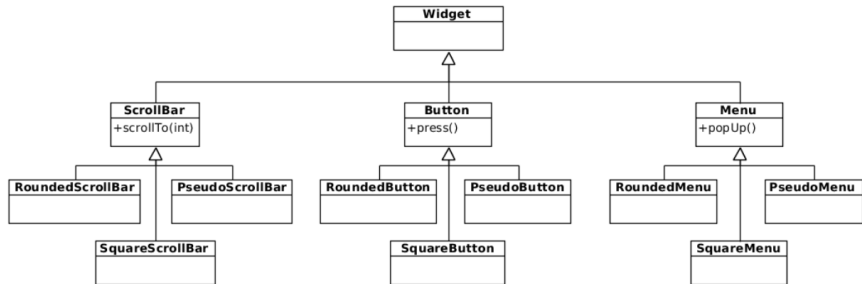
```
connect(openAct, &QAction::triggered, this, &MainWindow::open);
```

```
fileMenu->addAction(openAct);  
fileToolBar->addAction(openAct);
```

- ▶ ICommand в .NET

«Абстрактная фабрика», мотивация

- ▶ Хотим поддержать разные стили UI
 - ▶ Гибкая поддержка в архитектуре
 - ▶ Удобное добавление новых стилей



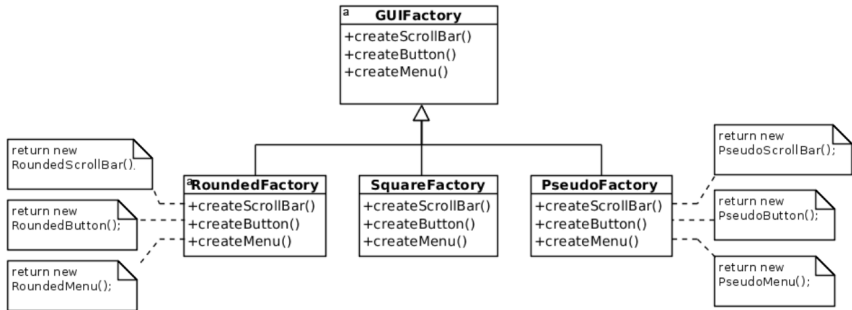
Создание виджетов

ScrollBar* bar = **new** RoundedScrollBar;

vs

ScrollBar* bar = guiFactory->createScrollBar();

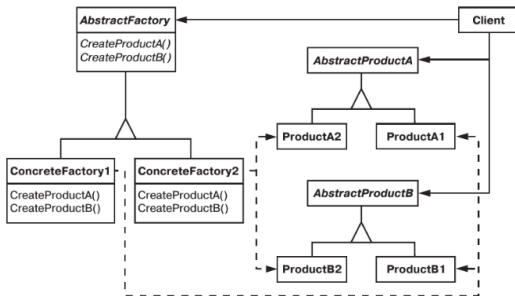
Фабрика виджетов



Паттерн «Абстрактная фабрика»

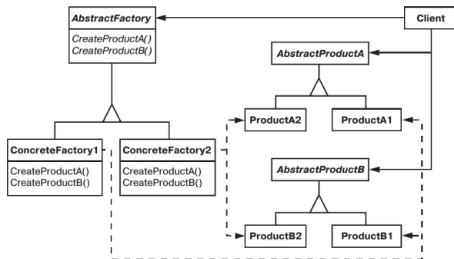
Abstract Factory

- ▶ Изолирует конкретные классы
- ▶ Упрощает замену семейств продуктов
- ▶ Гарантирует сочетаемость продуктов
- ▶ Поддержать новый вид продуктов непросто



«Абстрактная фабрика», детали реализации

- ▶ Хорошо комбинируются с паттерном «Одиночка»
- ▶ Если семейств продуктов много, то фабрика может инициализироваться *прототипами*, тогда не надо создавать сотню подклассов
- ▶ Прототип на самом деле может быть классом (например, Class в Java)
- ▶ Часто это просто лямбда, создающая объект
- ▶ Если виды объектов часто меняются, может помочь параметризация метода создания

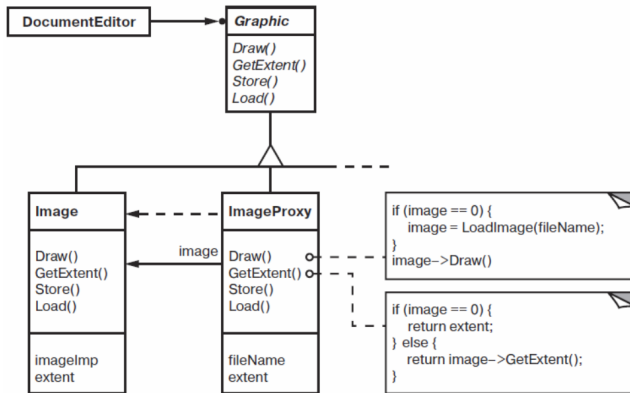


Управление доступом к объектам

- ▶ Встраивание в документ графических объектов
 - ▶ Затраты на создание могут быть значительными
 - ▶ Хотим отложить их на момент использования
- ▶ Использование заместителей объектов

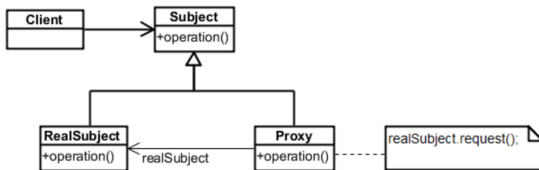


Отложенная загрузка изображения



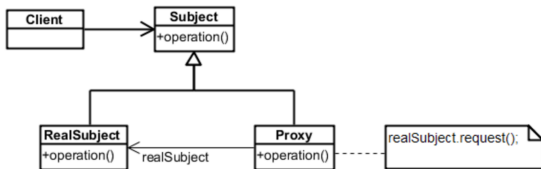
Паттерн «Заместитель»

Proxy



- ▶ Замещение удалённых объектов
- ▶ Создание «тяжёлых» объектов по требованию
- ▶ Контроль доступа
- ▶ Умные указатели
 - ▶ Подсчёт ссылок
 - ▶ Ленивая загрузка/инициализация
 - ▶ Работа с блокировками
 - ▶ Копирование при записи

«Заместитель», детали реализации



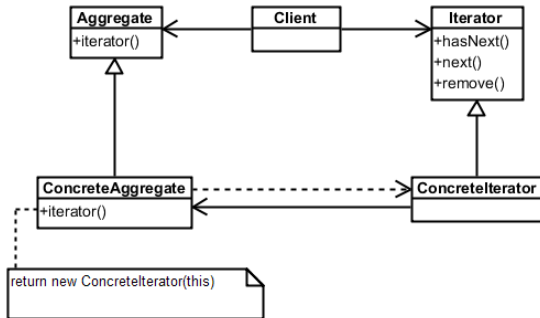
- ▶ Перегрузка оператора доступа к членам класса (для C++)
 - ▶ Умные указатели так устроены
 - ▶ C++ вызывает операторы `->` по цепочке
 - ▶ `object->do()` может быть хоть `((object.operator->()).operator->()).do()`
 - ▶ Не подходит, если надо различать операции

«Заместитель», детали реализации (2)

- ▶ Реализация «вручную» всех методов проксируемого объекта
 - ▶ Сотня методов по одной строчке каждый
 - ▶ C#/F#: **public void** do() => realSubject.do();
 - ▶ Препроцессор/генерация
 - ▶ Технологии наподобие WCF
- ▶ Проксируемого объекта может не быть в памяти

«Итератор» (Iterator)

Инкапсулирует способ обхода коллекции.



- ▶ Разные итераторы для разных способов обхода
- ▶ Можно обходить не только коллекции

«Итератор», примеры

► Java-стиль:

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

► .NET-стиль:

```
public interface IEnumerator<T>  
{  
    bool MoveNext();  
    T Current { get; }  
    void Reset();  
}
```

«Итератор», детали реализации (1)

- ▶ Внешние итераторы

foreach (Thing t **in** collection)

{

 Console.WriteLine(t);

}

- ▶ Внутренние итераторы

collection.ToList().ForEach(t => Console.WriteLine(t));

«Итератор», детали реализации (2)

- ▶ Итераторы и курсоры
- ▶ Устойчивые и неустойчивые итераторы
 - ▶ Паттерн «Наблюдатель»
 - ▶ Даже обнаружение модификации коллекции может быть не просто
- ▶ Дополнительные операции

Архитектурные стили

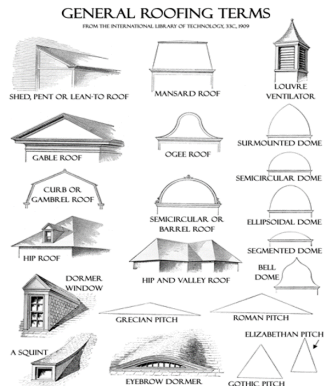
- ▶ Именованная коллекция архитектурных решений
- ▶ Менее узкоспециализированные, чем паттерны
- ▶ Определяют основные принципы построения системы в целом



© N. Medvidovic

Архитектурные стили

- ▶ Одна система может включать в себя несколько архитектурных стилей
- ▶ Понятие стиля применимо и к подсистемам



© N. Medvidovic

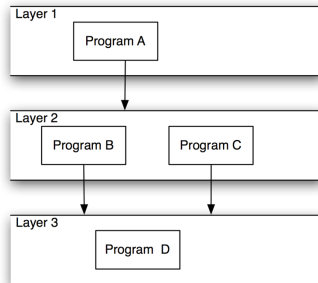
Слоистый стиль

Layered style

- ▶ Иерархическая организация системы
 - ▶ Многоуровневый «клиент-сервер»
 - ▶ Каждый слой предоставляет интерфейс для использования слоями выше
- ▶ Каждый слой работает как:
 - ▶ Сервер — предоставляет функциональность слоям выше
 - ▶ Клиент — использует функциональность слоёв ниже
- ▶ Пример — операционные системы, сетевые стеки протоколов

Слоистый стиль, подробности

- ▶ Преимущества:
 - ▶ Повышение уровня абстракции
 - ▶ Лёгкость в расширении
 - ▶ Изменения в каждом уровне затрагивают максимум два соседних
 - ▶ Возможны разные реализации уровня, если они удовлетворяют интерфейсу
- ▶ Недостатки:
 - ▶ Не всегда применим
 - ▶ Проблемы с производительностью



© N. Medvidovic

Каналы и фильтры

Pipes and filters

- ▶ Компоненты — это фильтры, преобразующие данные из входных каналов в данные в выходных каналах
- ▶ Инварианты:
 - ▶ Фильтры независимы (не имеют разделяемого состояния)
 - ▶ Фильтры не знают о фильтрах до или после них
- ▶ Вариации:
 - ▶ Конвейеры — линейные последовательности фильтров
 - ▶ Ограниченные каналы — где канал это очередь с ограниченным количеством элементов
 - ▶ Типизированные каналы — где каналы отличаются по типу передаваемых данных

Каналы и фильтры, подробности

- ▶ **Преимущества:**
 - ▶ Поведение системы — это просто последовательное применение поведений компонентов
 - ▶ Легко добавлять, заменять и переиспользовать фильтры
 - ▶ Любые два фильтра можно использовать вместе
 - ▶ Широкие возможности для анализа
 - ▶ Пропускная способность, задержки, deadlock-и
 - ▶ Широкие возможности для параллелизма
- ▶ **Недостатки:**
 - ▶ Последовательное исполнение
 - ▶ Проблемы с интерактивными приложениями
 - ▶ Пропускная способность определяется самым «узким» элементом

Стили с неявным вызовом

- ▶ Оповещение о событии вместо явного вызова метода
 - ▶ Слушатели могут подписаться на событие
 - ▶ Система при наступлении события сама вызывает все зарегистрированные методы слушателей
- ▶ Компоненты имеют два вида интерфейсов — методы и события
- ▶ Два типа соединителей:
 - ▶ Явный вызов метода
 - ▶ Неявный вызов по наступлению события
- ▶ Инварианты:
 - ▶ Те, кто производит события, не знают, кто и как на них отреагирует
 - ▶ Не делается никаких предположений о том, как событие будет обработано и будет ли вообще

Стили с неявным вызовом, преимущества и недостатки

- ▶ Преимущества:
 - ▶ Переиспользование компонентов
 - ▶ Очень низкая связность между компонентами
 - ▶ Лёгкость в конфигурировании системы
 - ▶ Как во время компиляции, так и во время выполнения
- ▶ Недостатки:
 - ▶ Зачастую неинтуитивная структура системы
 - ▶ Компоненты не управляют последовательностью вычислений
 - ▶ Непонятно, кто отреагирует на запрос и в каком порядке придут ответы
 - ▶ Тяжело отлаживаться
 - ▶ Гонки даже в однопоточном приложении

Peer-to-peer

- ▶ Состояние и поведение распределены между компонентами, которые могут выступать как клиенты и как серверы
- ▶ Компоненты: имеют своё состояние и свой поток управления
- ▶ Соединители: как правило, сетевые протоколы
- ▶ Элементы данных: сетевые сообщения
- ▶ Топология: сеть (возможно, с избыточными связями между компонентами), может динамически меняться
- ▶ Преимущества:
 - ▶ Хорош для распределённых вычислений
 - ▶ Устойчив к отказам
 - ▶ Если протокол взаимодействия позволяет, легко масштабируется