

# Лекция 6: Структурные шаблоны

Юрий Литвинов  
yurii.litvinov@gmail.com

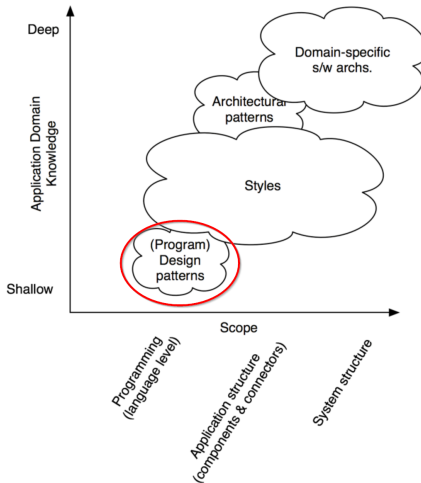
06.10.2020г

# 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

# Паттерны и архитектурные стили



© N. Medvidovic

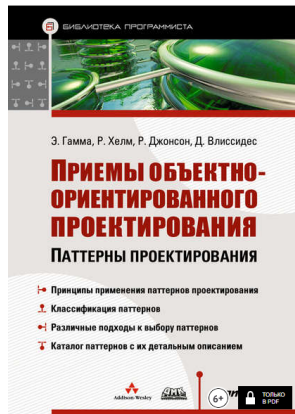
# Книжка про паттерны

Must read!

Приемы объектно-ориентированного проектирования. Паттерны проектирования

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

Design Patterns: Elements of Reusable Object-Oriented Software



# Начнём с примера

## Текстовый редактор

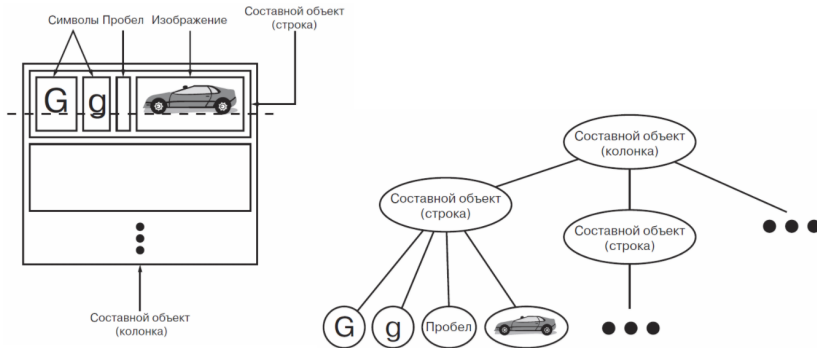
WYSIWYG-редактор, основные вопросы:

- ▶ Структура документа
- ▶ Форматирование
- ▶ Создание привлекательного интерфейса пользователя
- ▶ Поддержка стандартов внешнего облика программы
- ▶ Операции пользователя, undo/redo
- ▶ Проверка правописания и расстановка переносов

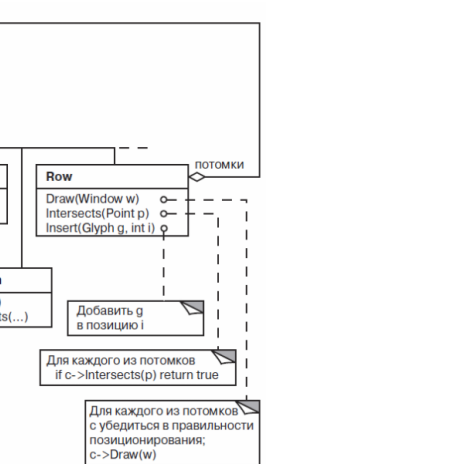
# Структура документа

- ▶ Документ — множество графических элементов
  - ▶ Организация в физическую структуру
  - ▶ Средства UI для манипулирования структурой
- ▶ Требования к внутреннему представлению
  - ▶ Отслеживание внутренней структуры документа
  - ▶ Генерирование визуального представления
  - ▶ Отображение позиций экрана на внутреннее представление
- ▶ Ограничения
  - ▶ Текст и графика едины
  - ▶ Простой и составной элементы едины

# Рекурсивная композиция



© Э. Гамма и др., Приемы объектно-ориентированного проектирования



ы объектно-ориентированного проектирования



# Паттерн "Компоновщик"

## Composite

- ▶ Представление иерархии объектов вида часть-целое
- ▶ Единообразная обработка простых и составных объектов
- ▶ Простота добавления новых компонентов
- ▶ Пример:
  - ▶ Синтаксические деревья



## “Компоновщик” (Composite), детали реализации

- ▶ Ссылка на родителя
  - ▶ Может быть полезна для простоты обхода
  - ▶ “Цепочка обязанностей”
  - ▶ Но дополнительный инвариант
  - ▶ Обычно реализуется в Component
- ▶ Разделяемые поддеревья и листья
  - ▶ Позволяют сильно экономить память
  - ▶ Проблемы с навигацией к родителям и разделяемым состоянием
  - ▶ Паттерн “Приспособленец”
- ▶ Идеологические проблемы с операциями для работы с потомками
  - ▶ Не имеют смысла для листа
    - ▶ Можно считать Leaf Composite-ом, у которого всегда 0 потомков
  - ▶ Операции add и remove можно объявить и в Composite, тогда придётся делать cast
    - ▶ Иначе надо бросать исключения в add и remove

## “Компоновщик”, детали реализации (2)

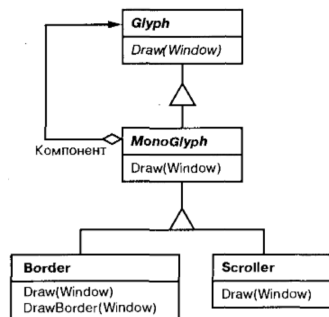
- ▶ Операция `getComposite()` – более аккуратный аналог `cast-a`
- ▶ Где определять список потомков
  - ▶ В `Composite`, экономия памяти
  - ▶ В `Component`, единообразие операций
  - ▶ “Список” вполне может быть хеш-таблицей, деревом или чем угодно
- ▶ Порядок потомков может быть важен, может нет
- ▶ Кеширование информации для обхода или поиска
  - ▶ Например, кеширование ограничивающих прямоугольников для фрагментов картинки
  - ▶ Инвалидация кеша
- ▶ Удаление потомков
  - ▶ Если нет сборки мусора, то лучше в `Composite`
  - ▶ Следует опасаться разделяемых листьев/поддеревьев

# Усовершенствование UI

- ▶ Хотим сделать рамку вокруг текста и полосы прокрутки, отключаемые по опции
- ▶ Желательно убирать и добавлять элементы оформления так, чтобы другие объекты даже не знали, что они есть
- ▶ Хотим менять во время выполнения — наследование не подойдёт
  - ▶ Наш выбор — композиция
  - ▶ Прозрачное оформление

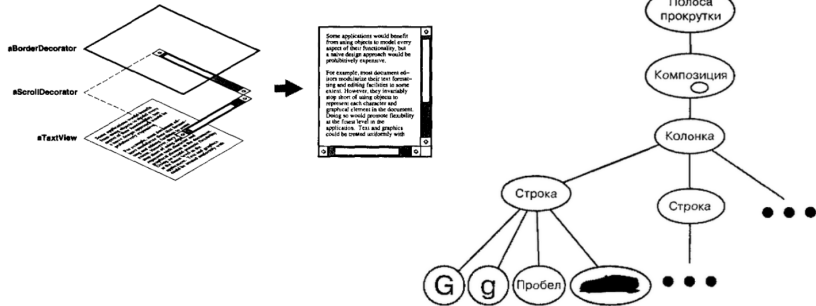
# Моноглиф

- ▶ Абстрактный класс с ровно одним сыном
  - ▶ Вырожденный случай компоновщика
- ▶ “Обрамляет” сына, добавляя новую функциональность



© Э. Гамма и др., Приемы  
объектно-ориентированного  
проектирования

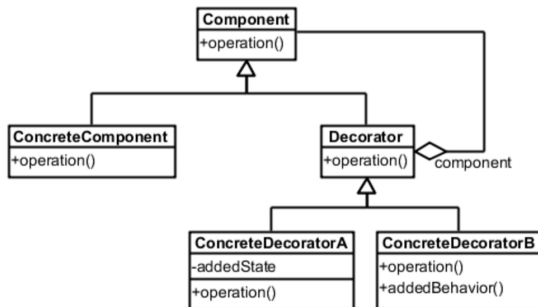
# Структура глифов



© Э. Гамма и др., Приемы объектно-ориентированного проектирования

# Паттерн “Декоратор”

Decorator



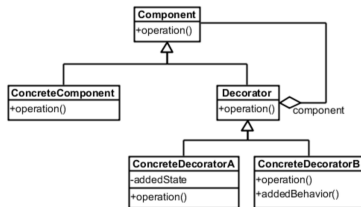
## Декоратор, особенности

- ▶ Динамическое добавление (и удаление) обязанностей объектов
  - ▶ Большая гибкость, чем у наследования
- ▶ Позволяет избежать перегруженных функциональностью базовых классов
- ▶ Много мелких объектов



# “Декоратор” (Decorator), детали реализации

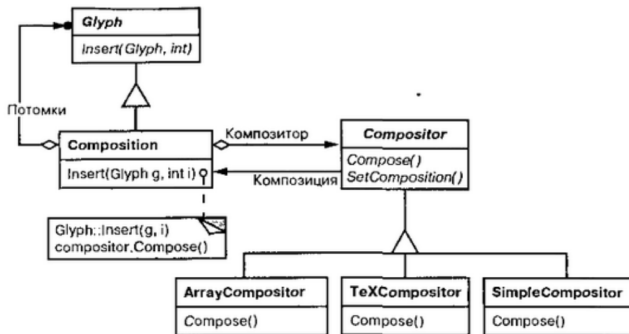
- ▶ Интерфейс декоратора должен соответствовать интерфейсу декорируемого объекта
  - ▶ Иначе получится “Адаптер”
- ▶ Если конкретный декоратор один, абстрактный класс можно не делать
- ▶ Component должен быть по возможности небольшим (в идеале, интерфейсом)
  - ▶ Иначе лучше паттерн “Стратегия”
  - ▶ Или самодельный аналог, например, список “расширений”, которые вызываются декорируемым объектом вручную перед операцией или после неё



# Форматирование текста

- ▶ Задача — разбиение текста на строки, колонки и т.д.
- ▶ Высокоуровневые параметры форматирования
  - ▶ Ширина полей, размер отступа, межстрочный интервал и т.д.
- ▶ Компромисс между качеством и скоростью работы
- ▶ Инкапсуляция алгоритма

# Compositor и Composition

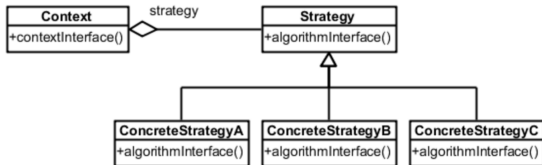


© Э. Гамма и др., Приемы объектно-ориентированного проектирования

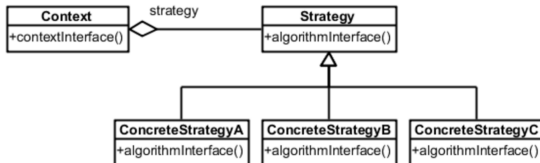
# Паттерн “Стратегия”

## Strategy

- ▶ Назначение — инкапсуляция алгоритма в объект
- ▶ Самое важное — спроектировать интерфейсы стратегии и контекста
  - ▶ Так, чтобы не менять их для каждой стратегии
- ▶ Применяется, если
  - ▶ Имеется много родственных классов с разным поведением
  - ▶ Нужно иметь несколько вариантов алгоритма
  - ▶ В алгоритме есть данные, про которые клиенту знать не надо
  - ▶ В коде много условных операторов



# “Стратегия” (Strategy), детали реализации



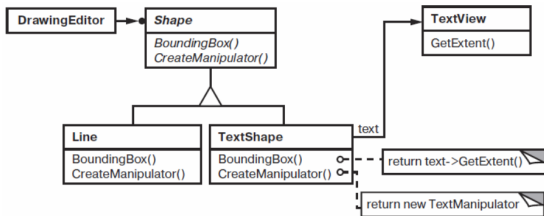
- ▶ Передача контекста вычислений в стратегию
  - ▶ Как параметры метода — уменьшает связность, но некоторые параметры могут быть стратегии не нужны
  - ▶ Передавать сам контекст в качестве аргумента — в **Context** интерфейс для доступа к данным

## “Стратегия” (Strategy), детали реализации (2)

- ▶ Стратегия может быть параметром шаблона
  - ▶ Если не надо её менять на лету
  - ▶ Не надо абстрактного класса и нет оверхеда на вызов виртуальных методов
- ▶ Стратегия по умолчанию
  - ▶ Или просто поведение по умолчанию, если стратегия не установлена
- ▶ Объект-стратегия может быть приспособленцем

# Проблема неподходящих интерфейсов

- ▶ Графический редактор
  - ▶ Shape, Line, Polygon, ...
- ▶ Сторонний класс TextView
  - ▶ Хотим его реализацию
  - ▶ Другой интерфейс

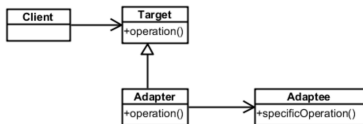


© Э. Гамма и др., Приемы объектно-ориентированного проектирования

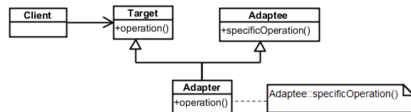
# Паттерн “Адаптер”

## Adapter

### ▶ Адаптер объекта:



### ▶ Адаптер класса:



### ▶ Нужно множественное наследование

#### ▶ private-наследование в C++

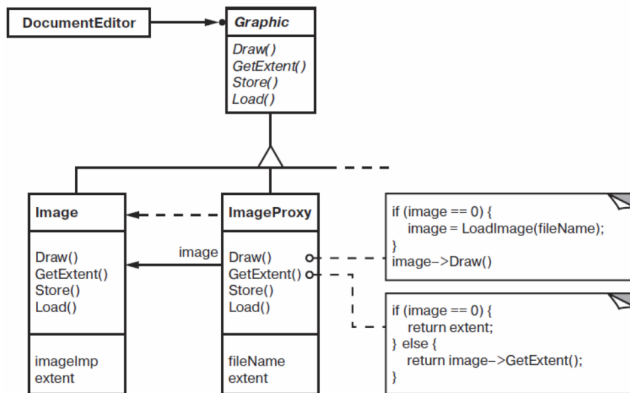


# Управление доступом к объектам

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

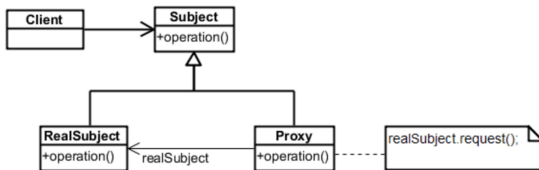


# Отложенная загрузка изображения



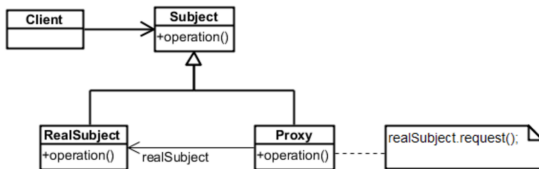
# Паттерн “Заместитель”

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

# Паттерн “Фасад”

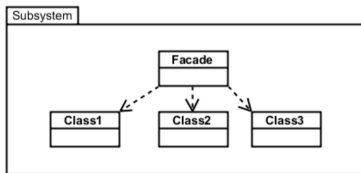
Facade

- ▶ Простой интерфейс к сложной системе
- ▶ Отделение подсистем от клиента и друг от друга
- ▶ Многоуровневая архитектура

# “Фасад” (Facade), детали реализации

- ▶ Абстрактный Facade

- ▶ Существенно снижает связность клиента с подсистемой



- ▶ Открытые и закрытые классы подсистемы

- ▶ Пространства имён и пакеты помогают, но требуют дополнительных соглашений
    - ▶ Пространство имён details
  - ▶ Инкапсуляция целой подсистемы — это хорошо

## Паттерн “Приспособленец” (Flyweight)

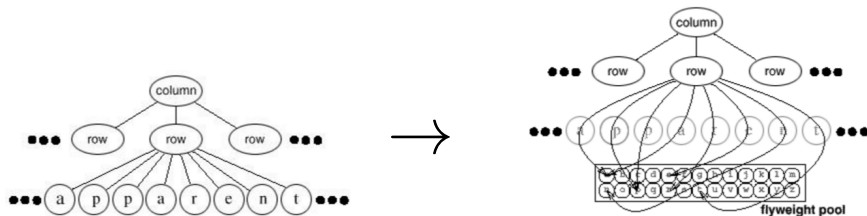
Предназначается для эффективной поддержки множества мелких объектов

Пример:

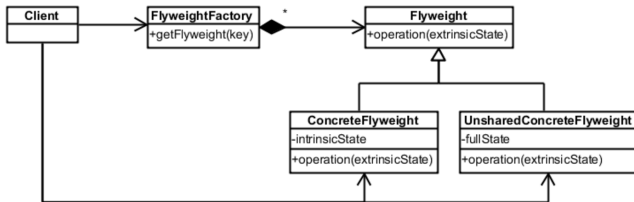
- ▶ Есть текстовый редактор
- ▶ Хочется работать с каждым символом как с объектом
  - ▶ Единообразие алгоритмов форматирования и внутренней структуры документа
  - ▶ Более красивая и ООПшная реализация
    - ▶ Паттерн “Компоновщик”, структура “Символ” → “Строка” → “Страница”
- ▶ Наивная реализация привела бы к чрезмерной расточительности по времени работы и по памяти, потому что документы с миллионами символов не редкость



# “Приспособленец”, пример

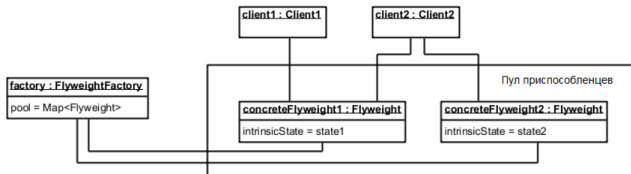


# “Приспособленец”, общая схема



- ▶ *Flyweight* — определяет интерфейс, через который приспособленцы могут получать внешнее состояние
- ▶ *ConcreteFlyweight* — реализует интерфейс *Flyweight* и может иметь внутреннее состояние, не зависит от контекста
- ▶ *UnsharedConcreteFlyweight* — неразделяемый “приспособленец”, хранящий всё состояние в себе, бывает нужен, чтобы собирать иерархические структуры из *Flyweight*-ов (“Компоновщик”)
- ▶ *FlyweightFactory* — содержит пул приспособленцев, создаёт их и управляет их жизнью

# “Приспособленец”, диаграмма объектов



- ▶ Клиенты могут быть разных типов
- ▶ Клиенты могут разделять приспособленцев
  - ▶ Один клиент может иметь несколько ссылок на одного приспособленца
- ▶ Во время выполнения клиенты имеют право не знать про фабрику

# Когда применять

- ▶ Когда в приложении используется много мелких объектов
- ▶ Они допускают разделение состояния на внутреннее и внешнее
  - ▶ Желательно, чтобы внешнее состояние было вычислимо
- ▶ Идентичность объектов не важна
  - ▶ Используется семантика Value Type
- ▶ Главное, когда от такого разделения можно получить ощутимый выигрыш

## Тонкости реализации

- ▶ Внешнее состояние — по сути, отдельный объект, поэтому если различных внешних состояний столько же, сколько приспособленцев, смысла нет
  - ▶ Один объект-состояние покрывает сразу несколько приспособленцев
    - ▶ Например, объект “Range” может хранить параметры форматирования для всех букв внутри фрагмента
- ▶ Клиенты не должны инстанцировать приспособленцев сами, иначе трудно обеспечить разделение
  - ▶ Имеет смысл иметь механизм для удаления неиспользуемых приспособленцев
    - ▶ Если их может быть много
- ▶ Приспособленцы немутабельны и Value Objects (с правильно переопределённой операцией сравнения)
  - ▶ Про hashCode() тоже надо не забыть