

Порождающие и поведенческие паттерны, детали реализации

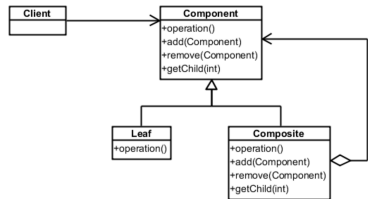
Юрий Литвинов
yurii.litvinov@gmail.com

19.04.2018г

“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

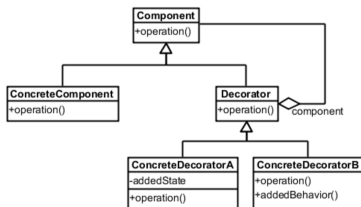
- ▶ Не имеют смысла для листа
 - ▶ Можно считать Leaf Composite-ом, у которого всегда 0 потомков
- ▶ Операции add и remove можно объявить и в Composite, тогда придётся делать cast
 - ▶ Иначе надо бросать исключения в add и remove

“Компоновщик”, детали реализации (2)

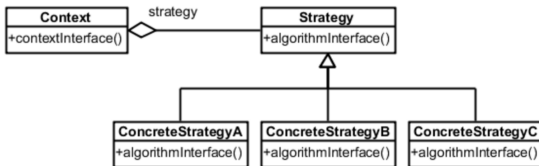
- ▶ Операция `getComposite()` – более аккуратный аналог `cast-a`
- ▶ Где определять список потомков
 - ▶ В `Composite`, экономия памяти
 - ▶ В `Component`, единообразие операций
 - ▶ “Список” вполне может быть хеш-таблицей, деревом или чем угодно
- ▶ Порядок потомков может быть важен, может нет
- ▶ Кеширование 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

“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



“Стратегия” (Strategy), детали реализации



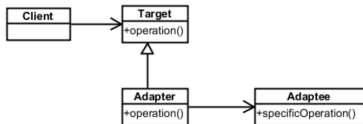
- ▶ Передача контекста вычислений в стратегию
 - ▶ Как параметры метода — уменьшает связность, но некоторые параметры могут быть стратегии не нужны
 - ▶ Передавать сам контекст в качестве аргумента — в Context интерфейс для доступа к данным

“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

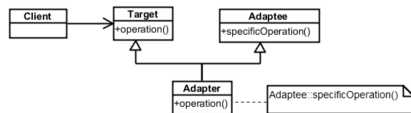
“Адаптер” (Adapter), детали реализации

▶ Адаптер объекта:



▶ Похоже на “Мост”

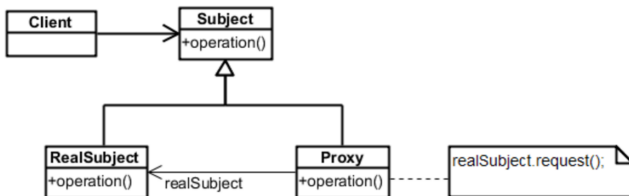
▶ Адаптер класса:



▶ Нужно множественное наследование

▶ private-наследование в C++

“Заместитель” (Proxy), 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

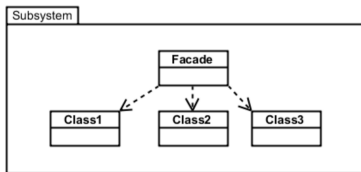
“Заместитель” (Proxy), детали реализации (2)

- ▶ 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

“Фасад” (Facade), детали реализации

▶ Абстрактный Facade

- ▶ Существенно снижает связность клиента с подсистемой

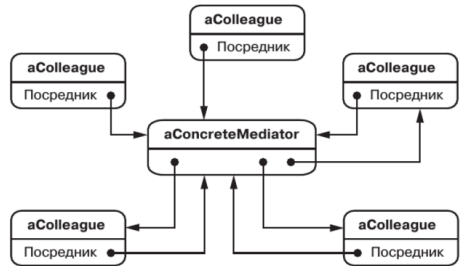


▶ Открытые и закрытые классы подсистемы

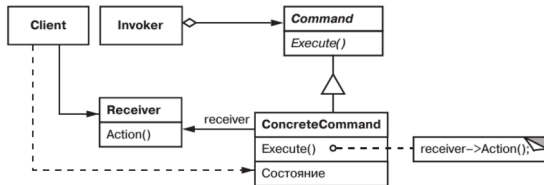
- ▶ Пространства имён и пакеты помогают, но требуют дополнительных соглашений
 - ▶ Пространство имён details
- ▶ Инкапсуляция целой подсистемы — это хорошо

“Посредник” (Mediator), детали реализации

- ▶ Абстрактный класс “Mediator” часто не нужен
- ▶ Паттерн “Наблюдатель”: медиатор подписывается на события в коллегах
- ▶ Наоборот: коллеги вызывают методы медиатора



“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
- ▶ Паттерн “Хранитель” для избежания ошибок восстановления

“Команда”, пример

- ▶ Qt, класс QAction:

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

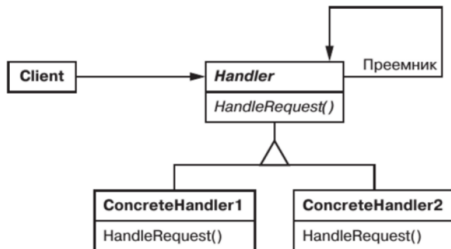
```
openAct->setShortcuts(QKeySequence::Open);  
openAct->setStatusTip(tr("Open an existing file"));
```

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

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

“Цепочка ответственности” (Chain of Responsibility), детали реализации

- ▶ Необязательно реализовывать связи в цепочке специально
 - ▶ На самом деле, чаще используются существующие связи



- ▶ По умолчанию в Handler передавать запрос дальше (если ссылки на преемника всё-таки есть)
- ▶ Если возможных запросов несколько, их надо как-то различать
 - ▶ Явно вызывать методы — нерасширяемо
 - ▶ Использовать объекты-запросы

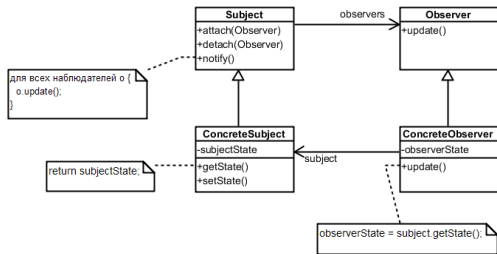
“Цепочка ответственности”, примеры

- ▶ Распространение исключений
- ▶ Распространение событий в оконных библиотеках:

```
void MyCheckBox::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton) {
        // handle left mouse button here
    } else {
        // pass on other buttons to base class
        QCheckBox::mousePressEvent(event);
    }
}
```

“Наблюдатель” (Observer), детали реализации

- ▶ В “нормальных” языках поддержан “из коробки” (через механизм событий)
- ▶ Могут использоваться хеш-таблицы для отображения субъектов и наблюдателей
 - ▶ Так делает WPF в .NET, есть даже языковая поддержка в C#
- ▶ Необходимость идентифицировать субъект
- ▶ Кто инициирует нотификацию
 - ▶ Операции, модифицирующие субъект
 - ▶ Клиент, после серии модификаций субъекта



“Наблюдатель” (Observer), детали реализации (2)

- ▶ Ссылки на субъектов и наблюдателей
 - ▶ Простой способ организовать утечку памяти в C# или грохнуть программу в C++
- ▶ Консистентность субъекта при отправке нотификации
 - ▶ Очевидно, но легко нарушить, вызвав метод предка в потомке
 - ▶ “Шаблонный метод”
 - ▶ Документировать, кто когда какие события бросает
- ▶ Передача сути изменений — pull vs push
- ▶ Фильтрация по типам событий
- ▶ Менеджер изменений (“Посредник”)

“Наблюдатель”, пример (1)

► События в C#:

```
internal class NewMessageEventArgs : EventArgs {  
    private readonly string message;  
  
    public NewMessageEventArgs(string message)  
        => this.message = message;  
  
    public string Message => message;  
}
```

“Наблюдатель”, пример (2)

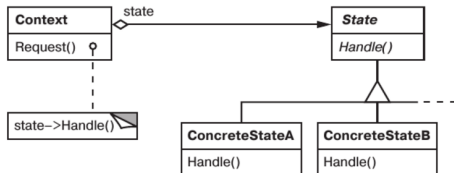
```
internal class Messenger {  
    public event EventHandler<NewMessageEventArgs> NewMessage;  
  
    protected virtual void OnMessage(NewMessageEventArgs e) {  
        EventHandler<NewMessageEventArgs> temp  
            = Volatile.Read(ref NewMessage);  
        if (temp != null)  
            temp(this, e);  
    }  
  
    public void SimulateMessage(String message) {  
        NewMessageEventArgs e = new NewMessageEventArgs(message);  
        OnMessage(e);  
    }  
}
```

“Наблюдатель”, пример (3)

```
internal sealed class Fax {  
    public Fax(Messenger mm) => mm.NewMessage += FaxMsg;  
  
    private void FaxMsg(object sender, NewMessageEventArgs e) {  
        Console.WriteLine("Faxing message:");  
        Console.WriteLine($"Message={e.Message}");  
    }  
  
    public void Unregister(Messenger mm)  
        => mm.NewMessage -= FaxMsg;  
}
```

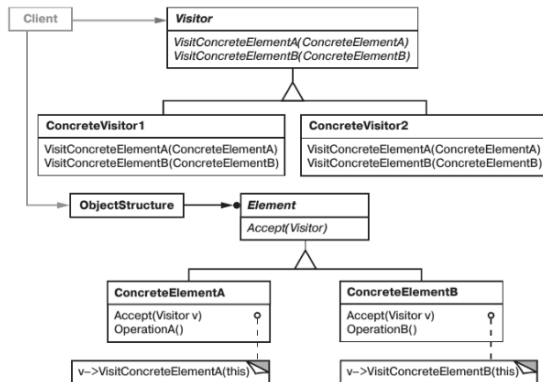
“Состояние” (State), детали реализации

- ▶ Переходы между состояниями — в Context или в State?
- ▶ Таблица переходов
 - ▶ Трудно добавить действия по переходу
- ▶ Создание и уничтожение состояний
 - ▶ Создать раз и навсегда
 - ▶ Создавать и удалять при переходах

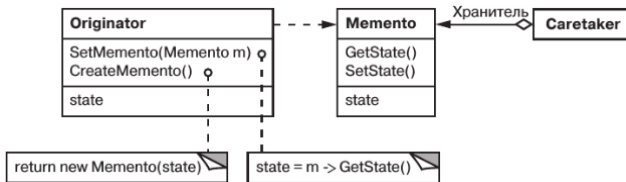


“Посетитель” (Visitor), детали реализации

- ▶ Использовать перегрузку методов Visit(...)
- ▶ Чаще всего сама коллекция отвечает за обход, но может быть итератор
- ▶ Может даже сам Visitor, если обход зависит от результата операции



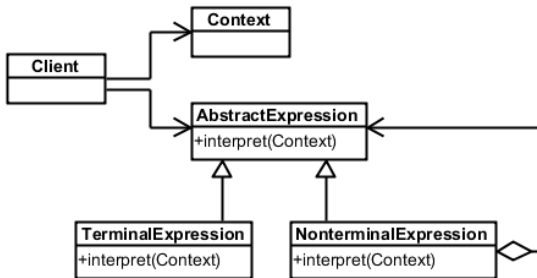
“Хранитель” (Memento), детали реализации



- ▶ Два интерфейса: “широкий” для хозяев и “узкий” для остальных объектов
 - ▶ Требуется языковая поддержка
- ▶ Можно хранить только дельты состояний

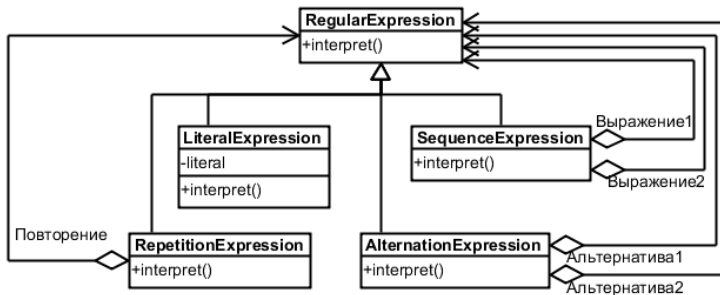
“Интерпретатор” (Interpreter)

Определяет представление грамматики и интерпретатор для заданного языка.



- ▶ Грамматика должна быть проста (иначе лучше “Visitor”)
- ▶ Эффективность не критична

"Интерпретатор", пример



“Интерпретатор”, детали реализации

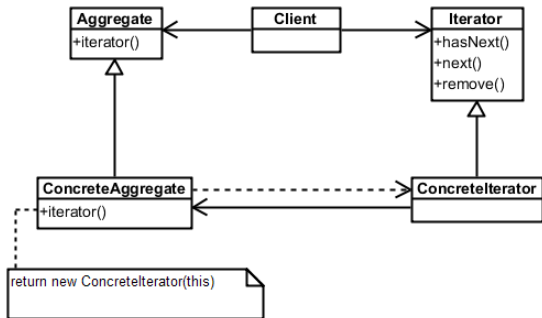
10-е правило Гринспена:

Любая достаточно сложная программа на Си или Фортране содержит заново написанную, неспецифицированную, глючную и медленную реализацию половины языка Common Lisp

- ▶ Построение дерева — отдельная задача
- ▶ Несколько разных операций над деревом — лучше “Visitor”
- ▶ Можно использовать “Приспособленец” для разделения терминальных символов

"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
- ▶ В C++ итераторы — это сложно