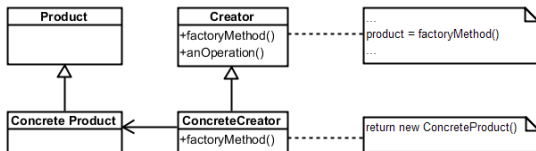


# Порождающие и поведенческие паттерны, детали реализации

Юрий Литвинов  
yurii.litvinov@gmail.com

10.04.2019г

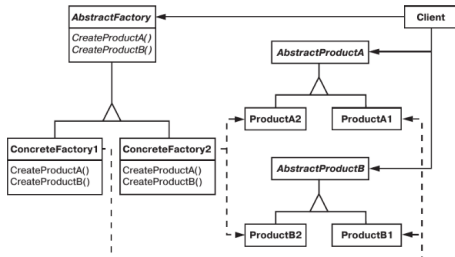
## “Фабричный метод” (Factory Method), детали реализации



- ▶ Абстрактный Creator или реализация по умолчанию
  - ▶ Второй вариант может быть полезен для расширяемости
- ▶ Параметризованные фабричные методы
- ▶ Если язык поддерживает инстанциацию по прототипу (JavaScript, Smalltalk), можно хранить порождаемый объект
- ▶ Creator не может вызывать фабричный метод в конструкторе
- ▶ Можно сделать шаблонный Creator

# “Абстрактная фабрика” (Abstract Factory), детали реализации

- ▶ Хорошо комбинируются с паттерном “Одиночка”
- ▶ Если семейств продуктов много, то фабрика может инициализироваться *прототипами*, тогда не надо создавать сотню подклассов

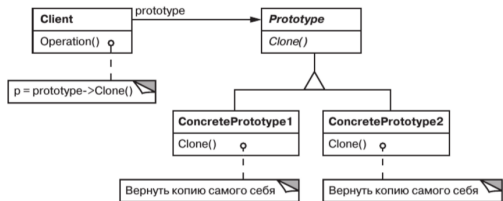


- ▶ Прототип на самом деле может быть классом (например, `Class` в Java)
- ▶ Если виды объектов часто меняются, может помочь параметризация метода создания
  - ▶ Может пострадать типобезопасность

# “Прототип” (Prototype), детали реализации

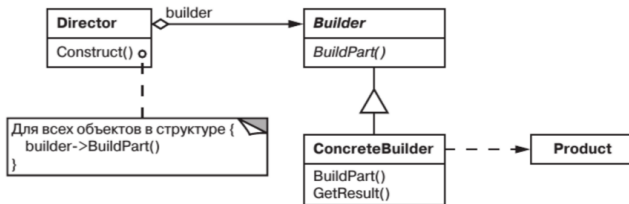
- ▶ Паттерн интересен только для языков, где мало runtime-информации о типе (C++)

- ▶ Реестр прототипов, обычно ассоциативное хранилище



- ▶ Операция Clone
  - ▶ Глубокое и мелкое копирование
  - ▶ В случае, если могут быть круговые ссылки
  - ▶ Сериализовать/десериализовать объект (но помнить про идентичность)
- ▶ Инициализация клона
  - ▶ Передавать параметры в Clone — плохая идея

## “Строитель” (Builder), детали реализации



- ▶ Абстрактные и конкретные строители
  - ▶ Достаточно общий интерфейс
- ▶ Общий интерфейс для продуктов не требуется
  - ▶ Клиент конфигурирует распорядителя конкретным строителем, он же и забирает результат
- ▶ Пустые методы по умолчанию

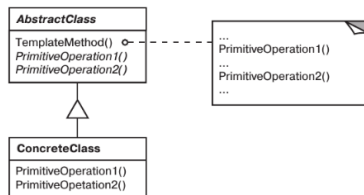
## “Строитель”, примеры

- ▶ StringBuilder
- ▶ Guava, подсистема работы с графами

```
MutableNetwork<Webpage, Link> webSnapshot =  
    NetworkBuilder.directed()  
        .allowsParallelEdges(true)  
        .nodeOrder(ElementOrder.natural())  
        .expectedNodeCount(100000)  
        .expectedEdgeCount(1000000)  
        .build();
```

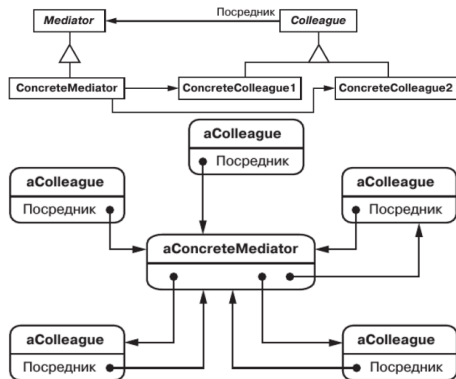
# “Шаблонный метод” (Template Method), детали реализации

- ▶ Сам шаблонный метод, как правило, не виртуальный
- ▶ Лучше использовать соглашения об именовании, например, называть операции с Do
- ▶ Примитивные операции могут быть виртуальными или чисто виртуальными
  - ▶ Лучше их делать protected
  - ▶ Чем их меньше, тем лучше



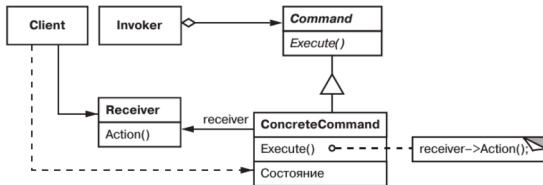
# “Посредник” (Mediator), детали реализации

- ▶ 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

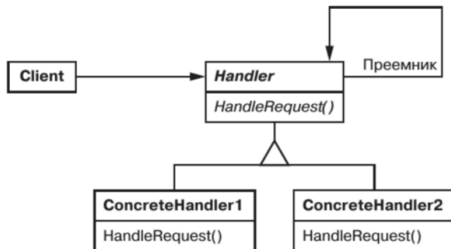
## “Команда”, пример

- ▶ 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

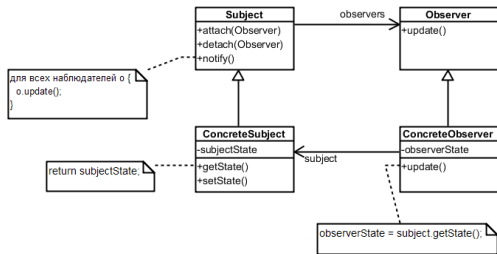
## “Цепочка ответственности”, примеры

- ▶ 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);  
    }  
}
```

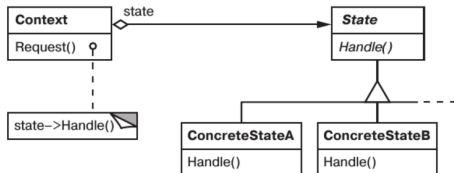


## “Наблюдатель”, пример (3)

```
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;  
}
```

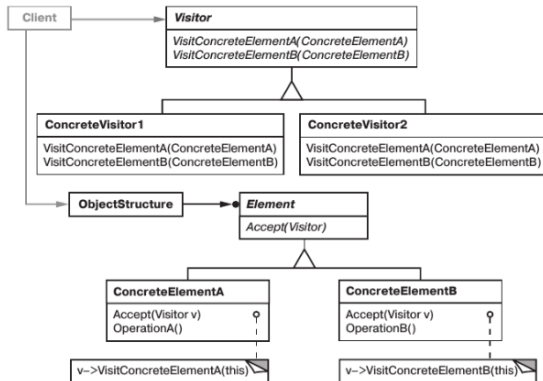
# “Состояние” (State), детали реализации

- ▶ 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

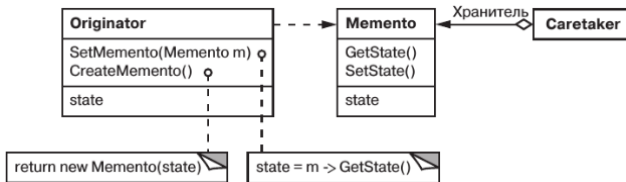


# “Посетитель” (Visitor), детали реализации

- ▶ 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



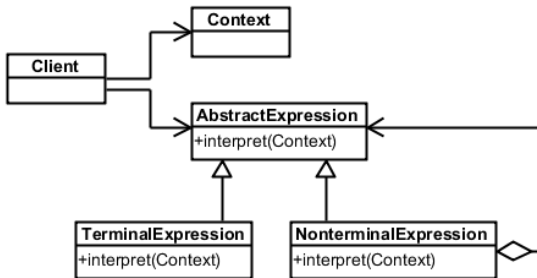
# “Хранитель” (Memento), детали реализации



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

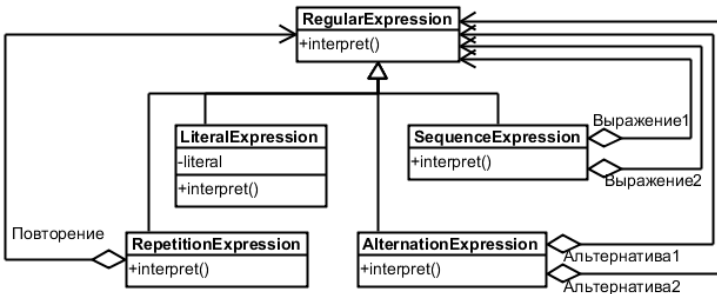
## “Интерпретатор” (Interpreter)

Определяет представление грамматики и интерпретатор для заданного языка.



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

# “Интерпретатор”, пример



# “Интерпретатор”, детали реализации

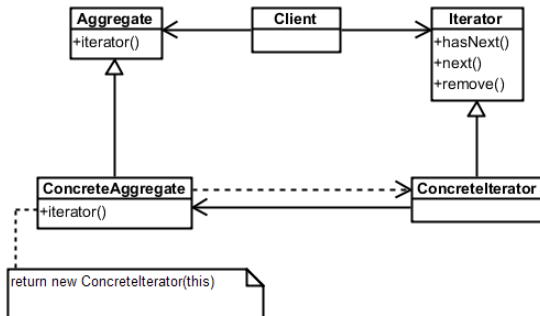
## 10-е правило Гринспена:

*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 непросто
- ▶ Дополнительные операции
- ▶ В C++ итераторы — это сложно