Печатная версия для ознакомления. Полная версия – на сайте:

http://cpp-reference.ru/patterns

http://sourcemaking.com/design_patterns

Сводная таблица паттернов проектирования

Оригинальное название	Русскоязычное название	Тип паттерна	Краткое описание
Abstarct Factory	Абстрактная фабрика	Порождающий	Создает семейство взаимосвязанных объектов
Adapter	Адаптер	Структурный	Преобразует интерфейс существующего класса к виду, подходящему для использования
Bridge	Мост	Структурный	Делает абстракцию и реализацию независимыми друг от друга
Builder	Строитель	Порождающий	Поэтапное создание сложного объекта
Chain of Responsibility	Цепочка обязанностей	Поведения	Предоставляет способ передачи запроса по цепочке получателей
Command	Команда	Поведения	Инкапсулирует запрос в виде объекта
Composite	Компоновщик	Структурный	Группирует схожие объекты в древовидные структуры
Decorator	Декоратор	Структурный	Динамически добавляет объекту новую функциональность
Facade	Фасад	Структурный	Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой системы
Factory Method	Фабричный метод	Порождающий	Определяет интерфейс для создания объекта, при этом его тип определяется подклассами
Flyweight	Приспособленец	Структурный	Использует разделение для поддержки множества мелких объектов
Interpreter	Интерпретатор	Поведения	Для языка определяет его грамматику и интепретатор, использующий эту грамматику
Iterator	Итератор	Поведения	Предоставляет механизм обхода элементов коллекции

Mediator	Посредник	Поведения	Инкапсулирует взаимодействие между множеством объектов в объект-посредник
Memento	Хранитель	Поведения	Сохраняет и восстанавливает состояние объекта
Object Pool	Пул объектов	Порождающий	Создание "затратных" объектов за счет их многократного использования
Observer	Наблюдатель	Поведения	При изменении объекта извещает всех зависимые объекты для их обновления
Prototype	Прототип	Порождающий	Создание объектов на основе прототипов
Proxy	Заместитель	Структурный	Подменяет другой объект для контроля доступа к нему
Singleton	Одиночка	Порождающий	Создает единственный экземпляр некоторого класса и предоставляет к нему доступ
State	Состояние	Поведения	Изменяет поведение объекта при изменении его состояния
Strategy	Стратегия	Поведения	Переносит алгоритмы в отдельную иерархию классов, делая их взаимозаменяемыми
Template Method	Шаблонный метод	Поведения	Определяет шаги алгоритма, позволяя подклассам изменить некоторые из них
Visitor	Посетитель	Поведения	Определяет новую операцию в классе без его изменения

Паттерн Abstract Factory (абстрактная фабрика)

Назначение паттерна Abstract Factory

Используйте паттерн Abstract Factory (абстрактная фабрика) если:

- Система должна оставаться независимой как от процесса создания новых объектов, так и от типов порождаемых объектов. Непосредственное использование выражения new в коде приложения нежелательно (подробнее об этом в разделе Порождающие паттерны).
- Необходимо создавать группы или семейства взаимосвязанных объектов, исключая возможность одновременного использования объектов из разных семейств в одном контексте.

Приведем примеры групп взаимосвязанных объектов.

Пусть некоторое приложение с поддержкой графического интерфейса пользователя рассчитано

на использование на различных платформах, при этом внешний вид этого интерфейса должен соответствовать принятому стилю для той или иной платформы. Например, если это приложение установлено на Windows-платформу, то его кнопки, меню, полосы прокрутки должны отображаться в стиле, принятом для Windows. Группой взаимосвязанных объектов в этом случае будут элементы графического интерфейса пользователя для конкретной платформы.

Другой пример. Рассмотрим текстовый редактор с многоязычной поддержкой, у которого имеются функциональные модули, отвечающие за расстановку переносов слов и проверку орфографии. Если, скажем, открыт документ на русском языке, то должны быть подключены соответствующие модули, учитывающие специфику русского языка. Ситуация, когда для такого документа одновременно используются модуль расстановки переносов для русского языка и модуль проверки орфографии для немецкого языка, исключается. Здесь группой взаимосвязанных объектов будут соответствующие модули, учитывающие специфику некоторого языка.

И последний пример. В разделе Порождающие паттерны говорилось об игре-стратегии, в которой описывается военное противостояние между армиями Рима и Карфагена. Очевидно, что внешний вид, боевые порядки и характеристики для разных родов войск (пехота, лучники, конница) в каждой армии будут своими. В данном случае семейством взаимосвязанных объектов будут все виды воинов для той или иной противоборствующей стороны, при этом должна исключаться, например, такая ситуация, когда римская конница воюет на стороне Карфагена.

Описание паттерна Abstract Factory

Паттерн Abstract Factory реализуется на основе фабричных методов (см. паттерн Factory Method).

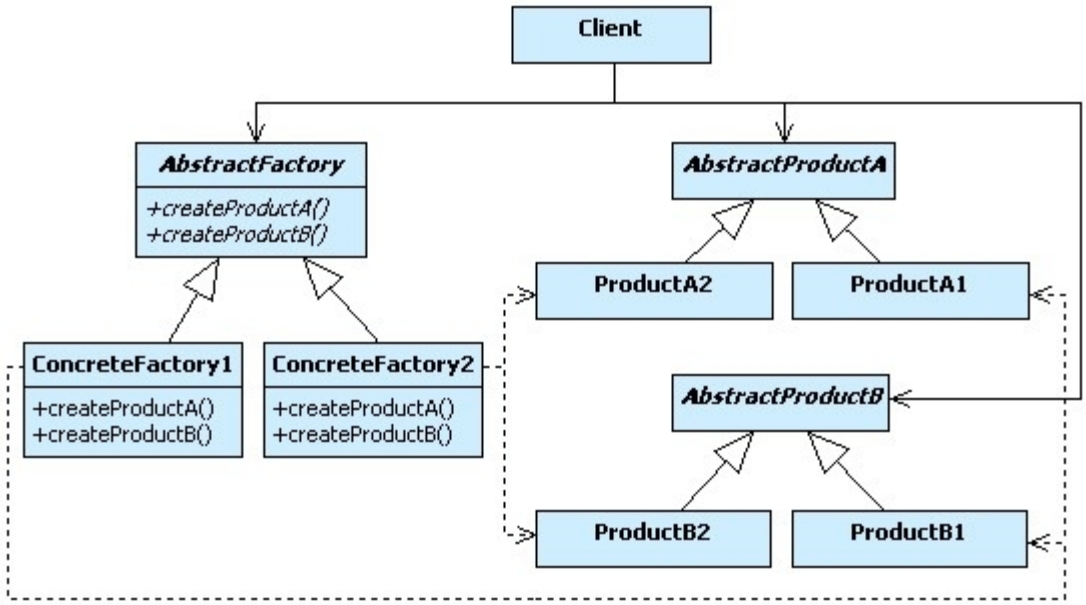
Любое семейство или группа взаимосвязанных объектов характеризуется несколькими общими типами создаваемых продуктов, при этом сами продукты таких типов будут различными для разных семейств. Например, для случая стратегической игры общими типами создаваемых продуктов будут пехота, лучники и конница, при этом каждый из этих родов войск римской армии может существенно отличаться по внешнему виду и боевым характеристикам от соответствуюших родов войск армии Карфагена.

Для того чтобы система оставалась независимой от специфики того или иного семейства продуктов необходимо использовать общие интерфейсы для всех основных типов продуктов. В случае стратегической игры это означает, что необходимо использовать три абстрактных базовых класса для каждого типа воинов: пехоты, лучников и конницы. Производные от них классы будут реализовывать специфику соответствующего типа воинов той или иной армии.

Для решения задачи по созданию семейств взаимосвязанных объектов паттерн Abstract Factory вводит понятие абстрактной фабрики. Абстрактная фабрика представляет собой некоторый полиморфный базовый класс, назначением которого является объявление интерфейсов фабричных методов, служащих для создания продуктов всех основных типов (один фабричный

метод на каждый тип продукта). Производные от него классы, реализующие эти интерфейсы, предназначены для создания продуктов всех типов внутри семейства или группы. В случае нашей игры базовый класс абстрактной фабрики должен определять интерфейс фабричных методов для создания пехотинцев, лучников и конницы, а два производных от него класса будут реализовывать этот интерфейс, создавая воинов всех родов войск для той или иной армии.

UML-диаграмма классов паттерна Abstract Factory



Реализация паттерна Abstract Factory

Приведем реализацию паттерна Abstract Factory для военной стратегии "Пунические войны". При этом предполагается, что число и типы создаваемых в начале игры боевых единиц идентичны для обеих армий. Подробное описание этой игры можно найти в разделе Порождающие паттерны.

```
#include <iostream>
#include <vector>

// Абстрактные базовые классы всех возможных видов воинов
class Infantryman
{
  public:
    virtual void info() = 0;
    virtual ~Infantryman() {}
};
```

```
class Archer
{
  public:
    virtual void info() = 0;
    virtual ~Archer() {}
};

class Horseman
{
  public:
    virtual void info() = 0;
    virtual ~Horseman() {}
};


// Классы всех видов воинов Римской армии
class RomanInfantryman: public Infantryman
{
  public:
    void info() {
      cout << "RomanInfantryman" << endl;
    }
};

class RomanArcher: public Archer
{
  public:
    void info() {
      cout << "RomanArcher" << endl;
    }
};

class RomanHorseman: public Horseman
{
  public:
    void info() {
      cout << "RomanHorseman" << endl;
    }
};


// Классы всех видов воинов армии Карфагена
class CarthaginianInfantryman: public Infantryman
{
  public:
    void info() {
      cout << "CarthaginianInfantryman" << endl;
```

```
    }
};

class CarthaginianArcher: public Archer
{
  public:
    void info() {
      cout << "CarthaginianArcher" << endl;
    }
};

class CarthaginianHorseman: public Horseman
{
  public:
    void info() {
      cout << "CarthaginianHorseman" << endl;
    }
};


// Абстрактная фабрика для производства воинов
class ArmyFactory
{
  public:
    virtual Infantryman* createInfantryman() = 0;
    virtual Archer* createArcher() = 0;
    virtual Horseman* createHorseman() = 0;
    virtual ~ArmyFactory() {}
};


// Фабрика для создания воинов Римской армии
class RomanArmyFactory: public ArmyFactory
{
  public:
    Infantryman* createInfantryman() {
      return new RomanInfantryman;
    }
    Archer* createArcher() {
      return new RomanArcher;
    }
    Horseman* createHorseman() {
      return new RomanHorseman;
    }
};
```

```
// Фабрика для создания воинов армии Карфагена
class CarthaginianArmyFactory: public ArmyFactory
{
  public:
    Infantryman* createInfantryman() {
      return new CarthaginianInfantryman;
    }
    Archer* createArcher() {
      return new CarthaginianArcher;
    }
    Horseman* createHorseman() {
      return new CarthaginianHorseman;
    }
};


// Класс, содержащий всех воинов той или иной армии
class Army
{
  public:
   ~Army() {
      int i;
      for(i=0; i<vi.size(); ++i)  delete vi[i];
      for(i=0; i<va.size(); ++i)  delete va[i];
      for(i=0; i<vh.size(); ++i)  delete vh[i];
    }
    void info() {
      int i;
      for(i=0; i<vi.size(); ++i)  vi[i]->info();
      for(i=0; i<va.size(); ++i)  va[i]->info();
      for(i=0; i<vh.size(); ++i)  vh[i]->info();
    }
    vector<Infantryman*> vi;
    vector<Archer*> va;
    vector<Horseman*> vh;
};


// Здесь создается армия той или иной стороны
class Game
{
  public:
    Army* createArmy( ArmyFactory& factory ) {
      Army* p = new Army;
      p->vi.push_back( factory.createInfantryman());
      p->va.push_back( factory.createArcher());
      p->vh.push_back( factory.createHorseman());
      return p;
```

```
    }
};


int main()
{
    Game game;
    RomanArmyFactory ra_factory;
    CarthaginianArmyFactory ca_factory;

    Army * ra = game.createArmy( ra_factory);
    Army * ca = game.createArmy( ca_factory);
    cout << "Roman army:" << endl;
    ra->info();
    cout << "\nCarthaginian army:" << endl;
    ca->info();
    // ...
}


Roman army:
RomanInfantryman
RomanArcher
RomanHorseman

Carthaginian army:
CarthaginianInfantryman
CarthaginianArcher
CarthaginianHorseman
```

Достоинства паттерна Abstract Factory

- Скрывает сам процесс порождения объектов, а также делает систему независимой от типов создаваемых объектов, специфичных для различных семейств или групп (пользователи оперируют этими объектами через соответствующие абстрактные интерфейсы).
- Позволяет быстро настраивать систему на нужное семейство создаваемых объектов. В случае многоплатформенного графического приложения для перехода на новую платформу, то есть для замены графических элементов (кнопок, меню, полос прокрутки) одного стиля другим достаточно создать нужный подкласс абстрактной фабрики. При этом условие невозможности одновременного использования элементов разных стилей для некоторой платформы будет выполнено автоматически.

Недостатки паттерна Abstract Factory

- Трудно добавлять новые типы создаваемых продуктов или заменять существующие, так как интерфейс базового класса абстрактной фабрики фиксирован. Например, если для

нашей стратегической игры нужно будет ввести новый вид военной единицы - осадные орудия, то надо будет добавить новый фабричный метод, объявив его интерфейс в полиморфном базовом классе AbstractFactory и реализовав во всех подклассах. Снять это ограничение можно следующим образом. Все создаваемые объекты должны наследовать от общего абстрактного базового класса, а в единственный фабричный метод в качестве параметра необходимо передавать идентификатор типа объекта, который нужно создать. Однако в этом случае необходимо учитывать следующий момент. Фабричный метод создает объект запрошенного подкласса, но при этом возвращает его с интерфейсом общего абстрактного класса в виде ссылки или указателя, поэтому для такого объекта будет затруднительно выполнить какую-либо операцию, специфичную для подкласса.

Паттерн Adapter (адаптер, wrapper, обертка)

Назначение паттерна Adapter

Часто в новом программном проекте не удается повторно использовать уже существующий код. Например, имеющиеся классы могут обладать нужной функциональностью, но иметь при этом несовместимые интерфейсы. В таких случаях следует использовать паттерн Adapter (адаптер).

Паттерн Adapter, представляющий собой программную обертку над существующими классами, преобразует их интерфейсы к виду, пригодному для последующего использования.

Рассмотрим простой пример, когда следует применять паттерн Adapter. Пусть мы разрабатываем систему климат-контроля, предназначенной для автоматического поддержания температуры окружающего пространства в заданных пределах. Важным компонентом такой системы является температурный датчик, с помощью которого измеряют температуру окружающей среды для последующего анализа. Для этого датчика уже имеется готовое программное обеспечение от сторонних разработчиков, представляющее собой некоторый класс с соответствующим интерфейсом. Однако использовать этот класс непосредственно не удастся, так как показания датчика снимаются в градусах Фаренгейта. Нужен адаптер, преобразующий температуру в шкалу Цельсия.

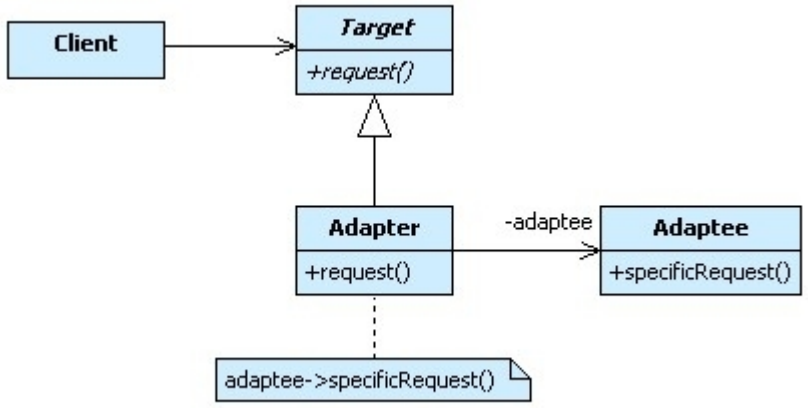
Контейнеры queue, priority_queue и stack библиотеки стандартных шаблонов STL реализованы на базе последовательных контейнеров list, deque и vector, адаптируя их интерфейсы к нужному виду. Именно поэтому эти контейнеры называют контейнерами-адаптерами.

Описание паттерна Adapter

Пусть класс, интерфейс которого нужно адаптировать к нужному виду, имеет имя Adaptee. Для решения задачи преобразования его интерфейса паттерн Adapter вводит следующую иерархию классов:

- Виртуальный базовый класс Target. Здесь объявляется пользовательский интерфейс подходящего вида. Только этот интерфейс доступен для пользователя.
- Производный класс Adapter, реализующий интерфейс Target. В этом классе также имеется указатель или ссылка на экземпляр Adaptee. Паттерн Adapter использует этот указатель для перенаправления клиентских вызовов в Adaptee. Так как интерфейсы Adaptee и Target несовместимы между собой, то эти вызовы обычно требуют преобразования.

UML-диаграмма классов паттерна Adapter



Реализация паттерна Adapter

Классическая реализация паттерна Adapter

Приведем реализацию паттерна Adapter. Для примера выше адаптируем показания температурного датчика системы климат-контроля, переведя их из градусов Фаренгейта в градусы Цельсия (предполагается, что код этого датчика недоступен для модификации).

```
#include <iostream>

// Уже существующий класс температурного датчика окружающей среды
class FahrenheitSensor
{
  public:
    // Получить показания температуры в градусах Фаренгейта
    float getFahrenheitTemp() {
      float t = 32.0;
      // ... какой то код
      return t;
    }
};

class Sensor
{
```

```
  public:
    virtual ~Sensor() {}
    virtual float getTemperature() = 0;
};

class Adapter : public Sensor
{
  public:
    Adapter( FahrenheitSensor* p ) : p_fsensor(p) {
    }
   ~Adapter() {
      delete p_fsensor;
    }
    float getTemperature() {
      return (p_fsensor->getFahrenheitTemp()-32.0)*5.0/9.0;
    }
  private:
    FahrenheitSensor* p_fsensor;
};

int main()
{
  Sensor* p = new Adapter( new FahrenheitSensor);
  cout << "Celsius temperature = " << p->getTemperature() << endl;
  delete p;
  return 0;
}
```

Реализация паттерна Adapter на основе закрытого наследования

Пусть наш температурный датчик системы климат-контроля поддерживает функцию юстировки для получения более точных показаний. Эта функция не является обязательной для использования, возможно, поэтому соответствующий метод adjust() объявлен разработчиками защищенным в существующем классе FahrenheitSensor.

Разрабатываемая нами система должна поддерживать настройку измерений. Так как доступ к защищенному методу через указатель или ссылку запрещен, то классическая реализация паттерна Adapter здесь уже не подходит. Единственное решение - наследовать от класса FahrenheitSensor. Интерфейс этого класса должен оставаться недоступным пользователю, поэтому наследование должно быть закрытым.

Цели, преследуемые при использовании открытого и закрытого наследования различны. Если открытое наследование применяется для наследования интерфейса и реализации, то закрытое наследование - только для наследования реализации.

```
#include <iostream>
```

```
class FahrenheitSensor
{
  public:
    float getFahrenheitTemp() {
      float t = 32.0;
      // ...
      return t;
    }
  protected:
    void adjust() {} // Настройка датчика (защищенный метод)
};

class Sensor
{
  public:
    virtual ~Sensor() {}
    virtual float getTemperature() = 0;
    virtual void adjust() = 0;
};

class Adapter : public Sensor, private FahrenheitSensor
{
  public:
    Adapter() { }
    float getTemperature() {
      return (getFahrenheitTemp()-32.0)*5.0/9.0;
    }
    void adjust() {
      FahrenheitSensor::adjust();
    }
};

int main()
{
  Sensor * p = new Adapter();
  p->adjust();
  cout << "Celsius temperature = " << p->getTemperature() << endl;
  delete p;
  return 0;
}
```

Результаты применения паттерна Adapter

Достоинства паттерна Adapter

- Паттерн Adapter позволяет повторно использовать уже имеющийся код, адаптируя его

несовместимый интерфейс к виду, пригодному для использования.

Недостатки паттерна Adapter

- Задача преобразования интерфейсов может оказаться непростой в случае, если клиентские вызовы и (или) передаваемые параметры не имеют функционального соответствия в адаптируемом объекте.

Паттерн Bridge (мост, идиома "Handle/Body")

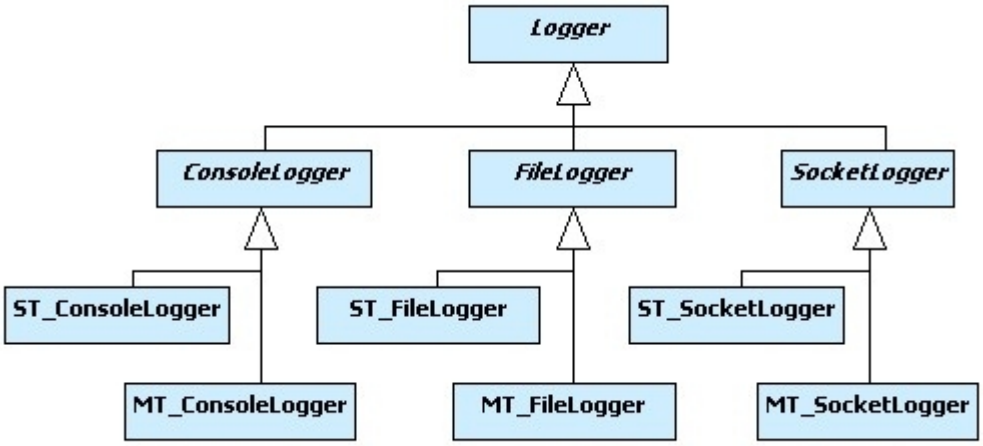
Назначение паттерна Bridge

В системе могут существовать классы, отношения между которыми строятся в соответствии со следующей объектно-ориентированной иерархией: абстрактный базовый класс объявляет интерфейс, а конкретные подклассы реализуют его нужным образом. Такой подход является стандартным в ООП, однако, ему свойственны следующие недостатки:

1. Система, построенная на основе наследования, является статичной. Реализация жестко привязана к интерфейсу. Изменить реализацию объекта некоторого типа в процессе выполнения программы уже невозможно.
2. Система становится трудно поддерживаемой, если число родственных производных классов становится большим.

Поясним сложности расширения системы новыми типами на примере разработки логгера. Логгер это система протоколирования сообщений, позволяющая фиксировать ошибки, отладочную и другую информацию в процессе выполнения программы. Разрабатываемый нами логгер может использоваться в одном из трех режимов: выводить сообщения на экран, в файл или отсылать их на удаленный компьютер. Кроме того, необходимо обеспечить возможность его применения в одно- и многопоточной средах.

Стандартный подход на основе полиморфизма использует следующую иерархию классов.



Видно, что число родственных подклассов в системе равно 6. Добавление еще одного вида логгера увеличит его до 8, двух - до 10 и так далее. Система становится трудно управляемой.

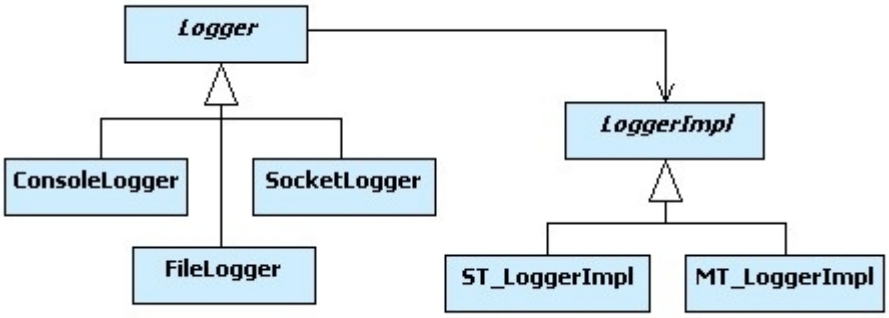
Указанные недостатки отсутствует в системе, спроектированной с применением паттерна Bridge.

Описание паттерна Bridge

Паттерн Bridge разделяет абстракцию и реализацию на две отдельные иерархии классов так, что их можно изменять независимо друг от друга.

Первая иерархия определяет интерфейс абстракции, доступный пользователю. Для случая проектируемого нами логгера абстрактный базовый класс Logger мог бы объявить интерфейс метода log() для вывода сообщений. Класс Logger также содержит указатель на реализацию pimpl, который инициализируется должным образом при создании логгера конкретного типа. Этот указатель используется для перенаправления пользовательских запросов в реализацию. Заметим, в общем случае подклассы ConsoleLogger, FileLogger и SocketLogger могут расширять интерфейс класса Logger.

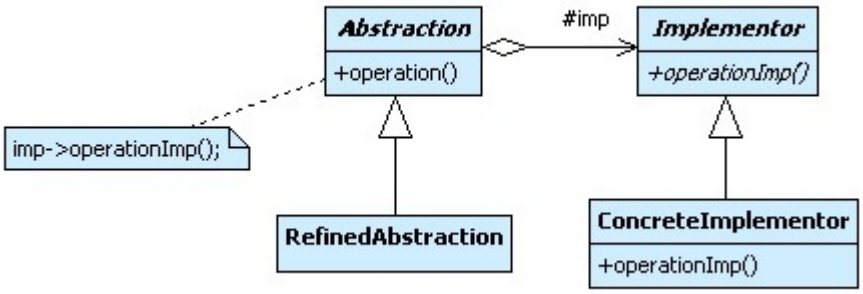
Все детали реализации, связанные с особенностями среды скрываются во второй иерархии. Базовый класс LoggerImpl объявляет интерфейс операций, предназначенных для отправки сообщений на экран, файл и удаленный компьютер, а подклассы ST_LoggerImpl и MT_LoggerImpl его реализуют для однопоточной и многопоточной среды соответственно. В общем случае, интерфейс LoggerImpl необязательно должен в точности соответствовать интерфейсу абстракции. Часто он выглядит как набор низкоуровневых примитивов.



Паттерн Bridge позволяет легко изменить реализацию во время выполнения программы. Для этого достаточно перенастроить указатель pimpl на объект-реализацию нужного типа. Применение паттерна Bridge также позволяет сократить общее число подклассов в системе, что делает ее более простой в поддержке.

Структура паттерна Bridge

UML-диаграмма классов паттерна Bridge



Паттерны Bridge и Adapter имеют схожую структуру, однако, цели их использования различны. Если паттерн Adapter применяют для адаптации уже существующих классов в систему, то паттерн Bridge используется на стадии ее проектирования.

Реализация паттерна Bridge

Приведем реализацию логгера с применением паттерна Bridge.

```
// Logger.h - Абстракция
#include <string>

// Опережающее объявление
class LoggerImpl;

class Logger
{
```

```
  public:
    Logger( LoggerImpl* p );
    virtual ~Logger( );
    virtual void log( string & str ) = 0;
  protected:
    LoggerImpl * pimpl;
};

class ConsoleLogger : public Logger
{
  public:
    ConsoleLogger();
    void log( string & str );
};

class FileLogger : public Logger
{
  public:
    FileLogger( string & file_name );
    void log( string & str );
  private:
    string file;
};

class SocketLogger : public Logger
{
  public:
    SocketLogger( string & remote_host, int remote_port );
    void log( string & str );
  private:
    string host;
    int  port;
};


// Logger.cpp - Абстракция
#include "Logger.h"
#include "LoggerImpl.h"

Logger::Logger( LoggerImpl* p ) : pimpl(p)
{ }

Logger::~Logger( )
{
    delete pimpl;
}

ConsoleLogger::ConsoleLogger() : Logger(
```

```
      #ifdef MT
        new MT_LoggerImpl()
      #else
        new ST_LoggerImpl()
      #endif
    )
{ }

void ConsoleLogger::log( string & str )
{
    pimpl->console_log( str);
}

FileLogger::FileLogger( string & file_name ) : Logger(
      #ifdef MT
        new MT_LoggerImpl()
      #else
        new ST_LoggerImpl()
      #endif
    ), file(file_name)
{ }

void FileLogger::log( string & str )
{
    pimpl->file_log( file, str);
}

SocketLogger::SocketLogger( string & remote_host,
                            int remote_port ) : Logger(
      #ifdef MT
        new MT_LoggerImpl()
      #else
        new ST_LoggerImpl()
      #endif
    ), host(remote_host), port(remote_port)
{ }

void SocketLogger::log( string & str )
{
    pimpl->socket_log( host, port, str);
}


// LoggerImpl.h - Реализация
#include <string>

class LoggerImpl
{
```

```
  public:
    virtual ~LoggerImpl( ) {}
    virtual void console_log( string & str ) = 0;
    virtual void file_log(
                    string & file, string & str ) = 0;
    virtual void socket_log(
           tring & host, int port, string & str ) = 0;
};

class ST_LoggerImpl : public LoggerImpl
{
  public:
    void console_log( string & str );
    void file_log   ( string & file, string & str );
    void socket_log (
            string & host, int port, string & str );
};

class MT_LoggerImpl : public LoggerImpl
{
  public:
    void console_log( string & str );
    void file_log   ( string & file, string & str );
    void socket_log (
            string & host, int port, string & str );
};


// LoggerImpl.cpp - Реализация
#include <iostream>
#include "LoggerImpl.h"


void ST_LoggerImpl::console_log( string & str )
{
  cout << "Single-threaded console logger" << endl;
}

void ST_LoggerImpl::file_log( string & file, string & str )
{
  cout << "Single-threaded file logger" << endl;
}

void ST_LoggerImpl::socket_log(
                     string & host, int port, string & str )
{
  cout << "Single-threaded socket logger" << endl;
```

```
};

void MT_LoggerImpl::console_log( string & str )
{
  cout << "Multithreaded console logger" << endl;
}

void MT_LoggerImpl::file_log( string & file, string & str )
{
  cout << "Multithreaded file logger" << endl;
}

void MT_LoggerImpl::socket_log(
                    string & host, int port, string & str )
{
  cout << "Multithreaded socket logger" << endl;
}


// Main.cpp
#include <string>
#include "Logger.h"

int main()
{
  Logger * p = new FileLogger( string("log.txt"));
  p->log( string("message"));
  delete p;
  return 0;
}
```

Отметим несколько важных моментов приведенной реализации паттерна Bridge:

1. При модификации реализации клиентский код перекомпилировать не нужно. Использование в абстракции указателя на реализацию (**идиома pimpl**) позволяет заменить в файле Logger.h включение include "LoggerImpl.h" на опережающее объявление class LoggerImpl. Такой прием снимает зависимость времени компиляции файла Logger.h (и, соответственно, использующих его файлов клиента) от файла LoggerImpl.h.
2. Пользователь класса Logger не видит никаких деталей его реализации.

Результаты применения паттерна Bridge

Достоинства паттерна Bridge

- Проще расширять систему новыми типами за счет сокращения общего числа

родственных подклассов.
- Возможность динамического изменения реализации в процессе выполнения программы.
- Паттерн Bridge полностью скрывает реализацию от клиента. В случае модификации реализации пользовательский код не требует перекомпиляции.

Паттерн Builder (строитель)

Назначение паттерна Builder

Паттерн Builder может помочь в решении следующих задач:

- В системе могут существовать сложные объекты, создание которых за одну операцию затруднительно или невозможно. Требуется поэтапное построение объектов с контролем результатов выполнения каждого этапа.
- Данные должны иметь несколько представлений. Приведем классический пример. Пусть есть некоторый исходный документ в формате RTF (Rich Text Format), в общем случае содержащий текст, графические изображения и служебную информацию о форматировании (размер и тип шрифтов, отступы и др.). Если этот документ в формате RTF преобразовать в другие форматы (например, Microsoft Word или простой ASCII-текст), то полученные документы и будут представлениями исходных данных.

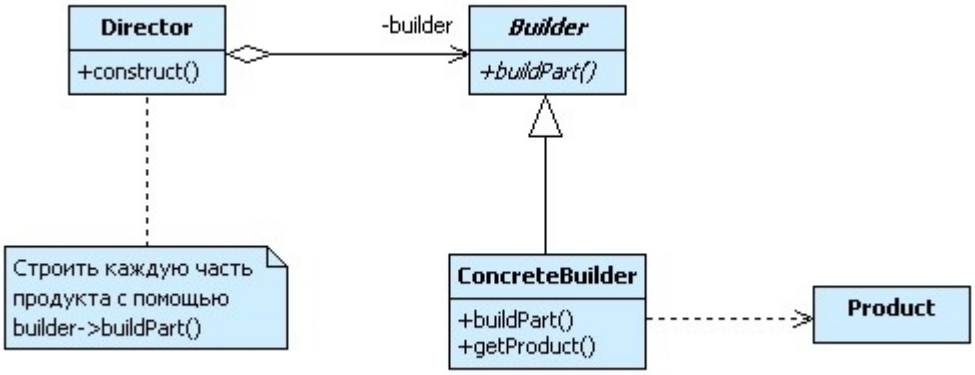
Описание паттерна Builder

Паттерн Builder отделяет алгоритм поэтапного конструирования сложного продукта (объекта) от его внешнего представления так, что с помощью одного и того же алгоритма можно получать разные представления этого продукта.

Поэтапное создание продукта означает его построение по частям. После того как построена последняя часть, продукт можно использовать.

Для этого паттерн Builder определяет алгоритм поэтапного создания продукта в специальном классе `Director` (распорядитель), а ответственность за координацию процесса сборки отдельных частей продукта возлагает на иерархию классов `Builder`. В этой иерархии базовый класс `Builder` объявляет интерфейсы для построения отдельных частей продукта, а соответствующие подклассы `ConcreteBuilder` их реализуют подходящим образом, например, создают или получают нужные ресурсы, сохраняют промежуточные результаты, контролируют результаты выполнения операций.

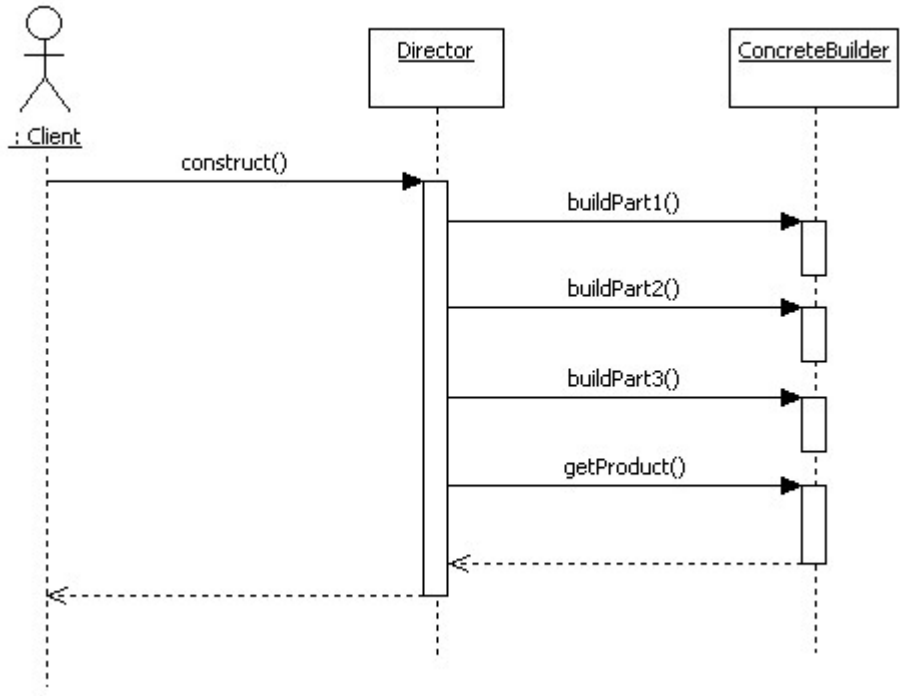
UML-диаграмма классов паттерна Builder



Класс `Director` содержит указатель или ссылку на `Builder`, который перед началом работы должен быть сконфигурирован экземпляром `ConcreteBuilder`, определяющим соответствующе представление. После этого `Director` может обрабатывать клиентские запросы на создание объекта. Получив такой запрос, с помощью имеющегося экземпляра строителя `Director` строит продукт по частям, а затем возвращает его пользователю.

Сказанное демонстрирует следующая диаграмма последовательностей.

UML-диаграмма последовательности паттерна Builder



Для получения разных представлений некоторых данных с помощью паттерна Builder распорядитель Director должен использовать соответствующие экземпляры ConcreteBuilder.

Ранее говорилось о задаче преобразования RTF-документов в документы различных форматов. Для ее решения класс Builder объявляет интерфейсы для преобразования отдельных частей исходного документа, таких как текст, графика и управляющая информация о форматировании, а производные классы WordBuilder, AsciiBuilder и другие их реализуют с учетом особенностей того или иного формата. Так, например, конвертор AsciiBuilder должен учитывать тот факт, что простой текст не может содержать изображений и управляющей информации о форматировании, поэтому соответствующие методы будут пустыми.

По запросу клиента распорядитель Director будет последовательно вычитывать данные из RTF-документа и передавать их в выбранный ранее конвертор, например, AsciiBuilder. После того как все данные прочитаны, полученный новый документ в виде ASCII-теста можно вернуть клиенту. Следует отметить, для того чтобы заменить формат исходных данных (здесь RTF) на другой, достаточно использовать другой класс распорядителя.

Реализация паттерна Builder

Приведем реализацию паттерна Builder на примере построения армий для военной стратегии "Пунические войны". Подробное описание этой игры можно найти в разделе Порождающие паттерны.

Для того чтобы не нагромождать код лишними подробностями будем полагать, что такие рода войск как пехота, лучники и конница для обеих армий идентичны. А с целью демонстрации возможностей паттерна Builder введем новые виды боевых единиц:

- Катапульты для армии Рима.
- Боевые слоны для армии Карфагена.

```
#include <iostream>
#include <vector>

// Классы всех возможных родов войск
class Infantryman
{
  public:
    void info() {
      cout << "Infantryman" << endl;
    }
};

class Archer
{
  public:
    void info() {
      cout << "Archer" << endl;
    }
};
```

```
class Horseman
{
  public:
    void info() {
      cout << "Horseman" << endl;
    }
};

class Catapult
{
  public:
    void info() {
      cout << "Catapult" << endl;
    }
};

class Elephant
{
  public:
    void info() {
      cout << "Elephant" << endl;
    }
};


// Класс "Армия", содержащий все типы боевых единиц
class Army
{
  public:
    vector<Infantryman> vi;
    vector<Archer>      va;
    vector<Horseman>    vh;
    vector<Catapult>    vc;
    vector<Elephant>    ve;
    void info() {
      int i;
      for(i=0; i<vi.size(); ++i)  vi[i].info();
      for(i=0; i<va.size(); ++i)  va[i].info();
      for(i=0; i<vh.size(); ++i)  vh[i].info();
      for(i=0; i<vc.size(); ++i)  vc[i].info();
      for(i=0; i<ve.size(); ++i)  ve[i].info();
    }
};


// Базовый класс ArmyBuilder объявляет интерфейс для поэтапного
// построения армии и предусматривает его реализацию по умолчанию
```

```
class ArmyBuilder
{
  protected:
    Army* p;
  public:
    ArmyBuilder(): p(0) {}
    virtual ~ArmyBuilder() {}
    virtual void createArmy() {}
    virtual void buildInfantryman() {}
    virtual void buildArcher() {}
    virtual void buildHorseman() {}
    virtual void buildCatapult() {}
    virtual void buildElephant() {}
    virtual Army* getArmy() { return p; }
};


// Римская армия имеет все типы боевых единиц кроме боевых слонов
class RomanArmyBuilder: public ArmyBuilder
{
  public:
    void createArmy() { p = new Army; }
    void buildInfantryman() { p->vi.push_back( Infantryman()); }
    void buildArcher() { p->va.push_back( Archer()); }
    void buildHorseman() { p->vh.push_back( Horseman()); }
    void buildCatapult() { p->vc.push_back( Catapult()); }
};


// Армия Карфагена имеет все типы боевых единиц кроме катапульт
class CarthaginianArmyBuilder: public ArmyBuilder
{
  public:
    void createArmy() { p = new Army; }
    void buildInfantryman() { p->vi.push_back( Infantryman()); }
    void buildArcher() { p->va.push_back( Archer()); }
    void buildHorseman() { p->vh.push_back( Horseman()); }
    void buildElephant() { p->ve.push_back( Elephant()); }
};


// Класс-распорядитель, поэтапно создающий армию той или иной
стороны.
// Именно здесь определен алгоритм построения армии.
class Director
{
```

```
  public:
    Army* createArmy( ArmyBuilder & builder )
    {
        builder.createArmy();
        builder.buildInfantryman();
        builder.buildArcher();
        builder.buildHorseman();
        builder.buildCatapult();
        builder.buildElephant();
        return( builder.getArmy());
    }
};


int main()
{
    Director dir;
    RomanArmyBuilder ra_builder;
    CarthaginianArmyBuilder ca_builder;

    Army * ra = dir.createArmy( ra_builder);
    Army * ca = dir.createArmy( ca_builder);
    cout << "Roman army:" << endl;
    ra->info();
    cout << "\nCarthaginian army:" << endl;
    ca->info();
    // ...

    return 0;
}
```

- Вывод программы будет следующим:

```
1 Roman army:
2 Infantryman
3 Archer
4 Horseman
5 Catapult
6
7 Carthaginian army:
8 Infantryman
9 Archer
10Horseman
11Elephant
```

Очень часто базовый класс строителя (в коде выше это `ArmyBuilder`) не только объявляет интерфейс для построения частей продукта, но и определяет ничего неделающую реализацию

по умолчанию. Тогда соответствующие подклассы (`RomanArmyBuilder`, `CarthaginianArmyBuilder`) переопределяют только те методы, которые участвуют в построении текущего объекта. Так класс `RomanArmyBuilder` не определяет метод `buildElephant`, поэтому Римская армия не может иметь слонов. А в классе `CarthaginianArmyBuilder` не определен `buildCatapult()`, поэтому армия Карфагена не может иметь катапульты.

Интересно сравнить приведенный код с кодом создания армии в реализации паттерна Abstract Factory, который также может использоваться для создания сложных продуктов. Если паттерн Abstract Factory акцентирует внимание на создании семейств некоторых объектов, то паттерн Builder подчеркивает поэтапное построение продукта. При этом класс `Builder` скрывает все подробности построения сложного продукта так, что `Director` ничего не знает о его составных частях.

Результаты применения паттерна Builder

Достоинства паттерна Builder

- Возможность контролировать процесс создания сложного продукта.
- Возможность получения разных представлений некоторых данных.

Недостатки паттерна Builder

- ConcreteBuilder и создаваемый им продукт жестко связаны между собой, поэтому при внесеннии изменений в класс продукта скорее всего придется соотвествующим образом изменять и класс ConcreteBuilder.

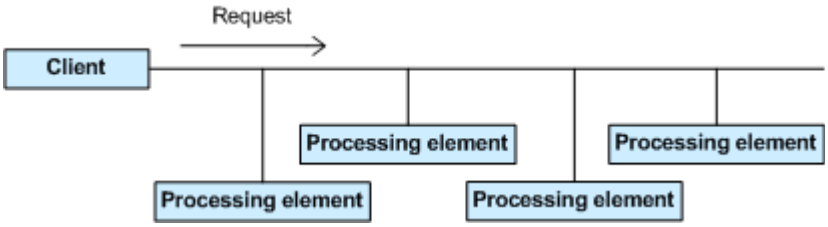
Паттерн Chain of Responsibility (цепочка обязанностей)

Назначение паттерна Chain of Responsibility

- Паттерн Chain of Responsibility позволяет избежать жесткой зависимости отправителя запроса от его получателя, при этом запрос может быть обработан несколькими объектами. Объекты-получатели связываются в цепочку. Запрос передается по этой цепочке, пока не будет обработан.
- Вводит конвейерную обработку для запроса с множеством возможных обработчиков.
- Объектно-ориентированный связанный список с рекурсивным обходом.

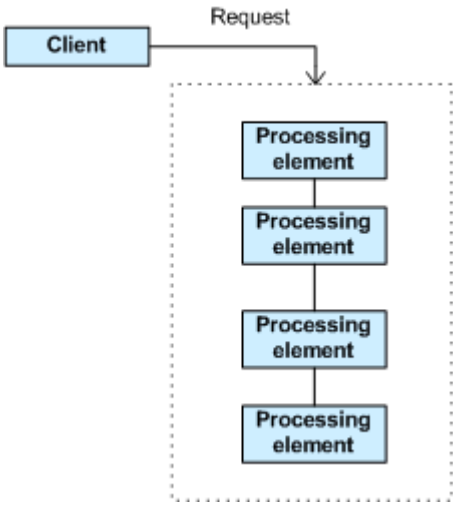
Решаемая проблема

Имеется поток запросов и переменное число "обработчиков" этих запросов. Необходимо эффективно обрабатывать запросы без жесткой привязки к их обработчикам, при этом запрос может быть обработан любым обработчиком.



Обсуждение паттерна Chain of Responsibility

Инкапсулирует элементы по обработке запросов внутри абстрактного "конвейера". Клиенты "кидают" свои запросы на вход этого конвейера.



Паттерн Chain of Responsibility связывает в цепочку объекты-получатели, а затем передает запрос-сообщение от одного объекта к другому до тех пор, пока не достигнет объекта, способного его обработать. Число и типы объектов-обработчиков заранее неизвестны, они могут настраиваться динамически. Механизм связывания в цепочку использует рекурсивную композицию, что позволяет использовать неограниченное число обработчиков.

Паттерн Chain of Responsibility упрощает взаимосвязи между объектами. Вместо хранения ссылок на всех кандидатов-получателей запроса, каждый отправитель хранит единственную ссылку на начало цепочки, а каждый получатель имеет единственную ссылку на своего преемника - последующий элемент в цепочке.

Убедитесь, что система корректно "отлавливает" случаи необработанных запросов.

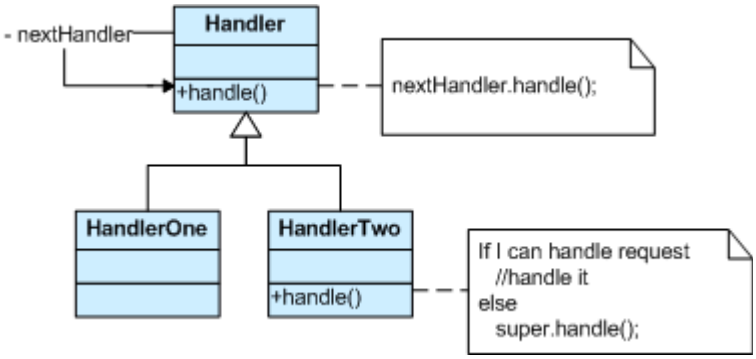
Не используйте паттерн Chain of Responsibility, когда каждый запрос обрабатывается только одним обработчиком, или когда клиент знает, какой именно объект должен обработать его

запрос.

Структура паттерна Chain of Responsibility

Производные классы знают, как обрабатывать запросы клиентов. Если "текущий" объект не может обработать запрос, то он делегирует его базовому классу, который делегирует "следующему" объекту и так далее.

UML-диаграмма классов паттерна Chain of Responsibility



Обработчики могут вносить свой вклад в обработку каждого запроса. Запрос может быть передан по всей длине цепочки до самого последнего звена.

Пример паттерна Chain of Responsibility

Паттерн Chain of Responsibility позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким получателям. Банкомат использует Chain of Responsibility в механизме выдачи денег.

Использование паттерна Chain of Responsibility

- Базовый класс имеет указатель на "следующий обработчик".
- Каждый производный класс реализует свой вклад в обработку запроса.
- Если запрос должен быть "передан дальше", то производный класс "вызывает" базовый класс, который с помощью указателя делегирует запрос далее.
- Клиент (или третья сторона) создает цепочку получателей (которая может иметь ссылку с последнего узла на корневой узел).
- Клиент передает каждый запрос в начало цепочки.
- Рекурсивное делегирование создает иллюзию волшебства.

Особенности паттерна Chain of Responsibility

- Паттерны Chain of Responsibility, Command, Mediator и Observer показывают, как можно разделить отправителей и получателей с учетом их особенностей. Chain of Responsibility передает запрос отправителя по цепочке потенциальных получателей.
- Chain of Responsibility может использовать Command для представления запросов в виде объектов.
- Chain of Responsibility часто применяется вместе с паттерном Composite. Родитель компонента может выступать в качестве его преемника.

Реализация паттерна Chain of Responsibility

Реализация паттерна Chain of Responsibility по шагам

1. Создайте указатель на следующий обработчик `next` в базовом классе.
2. Метод `handle()` базового класса всегда делегирует запрос следующему объекту.
3. Если производные классы не могут обработать запрос, они делегируют его базовому классу.

```
#include <iostream>
#include <vector>
#include <ctime>
using namespace std;

class Base
{
    // 1. Указатель "next" в базовом классе
    Base *next;
 public:
   Base()
   {
       next = 0;
   }
```

```
15      void setNext(Base *n)
16      {
17          next = n;
18      }
19      void add(Base *n)
20      {
21          if (next)
22            next->add(n);
23          else
24            next = n;
25      }
26      // 2. Метод базового класса, делегирующий запрос next-объекту
27      virtual void handle(int i)
28      {
29          next->handle(i);
30      }
31  };
32
33  class Handler1: public Base
34  {
35    public:
36       /*virtual*/void handle(int i)
37      {
38          if (rand() % 3)
39          {
40              // 3. 3 из 4 запросов не обрабатываем
41              cout << "H1 passsed " << i << "  ";
42              // 3. и делегируем базовому классу
43              Base::handle(i);        }
44          else
45            cout << "H1 handled " << i << "  ";
46      }
47  };
48
49  class Handler2: public Base
50  {
51    public:
52       /*virtual*/void handle(int i)
53      {
54          if (rand() % 3)
55          {
56              cout << "H2 passsed " << i << "  ";
57              Base::handle(i);
58          }
59          else
60            cout << "H2 handled " << i << "  ";
61      }
62  };
63
```

```
class Handler3: public Base
{
  public:
     /*virtual*/void handle(int i)
    {
        if (rand() % 3)
        {
            cout << "H3 passsed " << i << "  ";
            Base::handle(i);
        }
        else
          cout << "H3 handled " << i << "  ";
    }
};

int main()
{
  srand(time(0));
  Handler1 root;
  Handler2 two;
  Handler3 thr;
  root.add(&two);
  root.add(&thr);
  thr.setNext(&root);
  for (int i = 1; i < 10; i++)
  {
    root.handle(i);
    cout << '\n';
  }
}
```

Вывод программы:

```
H1 passsed 1  H2 passsed 1  H3 passsed 1  H1 passsed 1  H2 handled 1
H1 handled 2
H1 handled 3
H1 passsed 4  H2 passsed 4  H3 handled 4
H1 passsed 5  H2 handled 5
H1 passsed 6  H2 passsed 6  H3 passsed 6  H1 handled 6
H1 passsed 7  H2 passsed 7  H3 passsed 7  H1 passsed 7  H2 handled 7
H1 handled 8
H1 passsed 9  H2 passsed 9  H3 handled 9
```

Реализация паттерна Chain of Responsibility: Chain and Composite

1. Создайте указатель на следующий обработчик `next` в базовом классе.
2. Метод `handle()` базового класса всегда делегирует запрос следующему объекту.
3. Если производные классы не могут обработать запрос, они делегируют его базовому классу.

```
#include <iostream>
#include <vector>
#include <ctime>
using namespace std;

class Component
{
    int value;
    // 1. Указатель "next" в базовом классе
    Component *next;
  public:
    Component(int v, Component *n)
    {
        value = v;
        next = n;
    }
    void setNext(Component *n)
    {
        next = n;
    }
    virtual void traverse()
    {
        cout << value << ' ';
    }
    // 2. Метод базового класса, делегирующий запрос next-объекту
    virtual void volunteer()
    {
        next->volunteer();
    }
};

class Primitive: public Component
{
  public:
    Primitive(int val, Component *n = 0): Component(val, n){}
     /*virtual*/void volunteer()
    {
        Component::traverse();
        // 3. Примитивные объекты не обрабатывают 5 из 6 запросов
        if (rand() *100 % 6 != 0)
          // 3. Делегируем запрос в базовый класс
          Component::volunteer();
    }
};

class Composite: public Component
{
    vector < Component * > children;

```

```
  public:
    Composite(int val, Component *n = 0): Component(val, n){}
    void add(Component *c)
    {
        children.push_back(c);
    }
     /*virtual*/void traverse()
    {
        Component::traverse();
        for (int i = 0; i < children.size(); i++)
          children[i]->traverse();
    }
    // 3. Составные объекты никогда не обрабатывают запросы
     /*virtual*/void volunteer()
    {
        Component::volunteer();
    }
};

int main()
{
  srand(time(0));                 // 1
  Primitive seven(7);             // |
  Primitive six(6, &seven);       // +-- 2
  Composite three(3, &six);       // |   |
  three.add(&six);
  three.add(&seven);              // |   +-- 4 5
  Primitive five(5, &three);      // |
  Primitive four(4, &five);       // +-- 3
  Composite two(2, &four);        // |   |
  two.add(&four);
  two.add(&five);                 // |   +-- 6 7
  Composite one(1, &two);         // |
  Primitive nine(9, &one);        // +-- 8 9
  Primitive eight(8, &nine);
  one.add(&two);
  one.add(&three);
  one.add(&eight);
  one.add(&nine);
  seven.setNext(&eight);
  cout << "traverse: ";
  one.traverse();
  cout << '\n';
  for (int i = 0; i < 8; i++)
  {
    one.volunteer();
    cout << '\n';
  }
}
```

Вывод программы:

```
traverse: 1 2 4 5 3 6 7 8 9
4
4 5 6 7
4 5 6 7 8 9 4 5 6 7 8 9 4
4
4 5 6
4 5
4 5
4 5 6 7 8 9 4 5 6 7 8 9 4 5 6
```

Паттерн Command (команда)

Назначение паттерна Command

Используйте паттерн Command если

- Система управляется событиями. При появлении такого события (запроса) необходимо выполнить определенную последовательность действий.
- Необходимо параметризировать объекты выполняемым действием, ставить запросы в очередь или поддерживать операции отмены (undo) и повтора (redo) действий.
- Нужен объектно-ориентированный аналог функции обратного вызова в процедурном программировании.

Пример событийно-управляемой системы – приложение с пользовательским интерфейсом. При выборе некоторого пункта меню пользователем вырабатывается запрос на выполнение определенного действия (например, открытия файла).

Описание паттерна Command

Паттерн Command преобразовывает запрос на выполнение действия в отдельный объект-команду. Такая инкапсуляция позволяет передавать эти действия другим функциям и объектам в качестве параметра, приказывая им выполнить запрошенную операцию. Команда – это объект, поэтому над ней допустимы любые операции, что и над объектом.

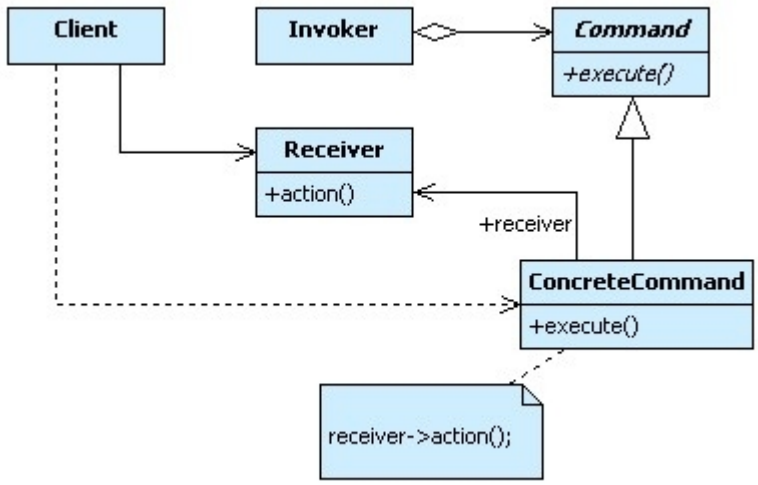
Интерфейс командного объекта определяется абстрактным базовым классом `Command` и в самом простом случае имеет единственный метод `execute()`. Производные классы определяют получателя запроса (указатель на объект-получатель) и необходимую для выполнения операцию (метод этого объекта). Метод `execute()` подклассов `Command` просто вызывает нужную операцию получателя.

В паттерне Command может быть до трех участников:

- Клиент, создающий экземпляр командного объекта.

- Инициатор запроса, использующий командный объект.
- Получатель запроса.

UML-диаграмма классов паттерна Command



Сначала клиент создает объект `ConcreteCommand`, конфигурируя его получателем запроса. Этот объект также доступен инициатору. Инициатор использует его при отправке запроса, вызывая метод `execute()`. Этот алгоритм напоминает работу функции обратного вызова в процедурном программировании – функция регистрируется, чтобы быть вызванной позднее.

Паттерн Command отделяет объект, инициирующий операцию, от объекта, который знает, как ее выполнить. Единственное, что должен знать инициатор, это как отправить команду. Это придает системе гибкость: позволяет осуществлять динамическую замену команд, использовать сложные составные команды, осуществлять отмену операций.

Реализация паттерна Command

Рассмотрим реализацию паттерна Command на примере игры «Шахматы». Имитируем возможность выполнения следующих операций:

- Создать новую игру.
- Открыть существующую игру.
- Сохранить игру.
- Сделать очередной ход.
- Отменить последний ход.

```
#include<iostream>
#include<vector>
```

```
#include<string>

class Game
{
  public:
    void create( ) {
      cout << "Create game " << endl;
    }
    void open( string file ) {
      cout << "Open game from " << file << endl;
    }
    void save( string file ) {
      cout << "Save game in " << file << endl;
    }
    void make_move( string move ) {
      cout << "Make move " << move << endl;
    }
};

string getPlayerInput( string prompt ) {
  string input;
  cout << prompt;
  cin >> input;
  return input;
}

// Базовый класс
class Command
{
  public:
    virtual ~Command() {}
    virtual void execute() = 0;
  protected:
    Command( Game* p ): pgame( p) {}
    Game * pgame;
};

class CreateGameCommand: public Command
{
  public:
    CreateGameCommand( Game * p ) : Command( p) {}
    void execute() {
      pgame->create( );
    }
};


class OpenGameCommand: public Command
```

```
{
  public:
    OpenGameCommand( Game * p ) : Command( p) {}
    void execute() {
      string file_name;
      file_name = getPlayerInput( "Enter file name:");
      pgame->open( file_name);
    }
};

class SaveGameCommand: public Command
{
  public:
    SaveGameCommand( Game * p ) : Command( p) {}
    void execute( ) {
      string file_name;
      file_name = getPlayerInput( "Enter file name:");
      pgame->save( file_name);
    }
};

class MakeMoveCommand: public Command
{
  public:
    MakeMoveCommand( Game * p) : Command( p) {}
    void execute() {
      // Сохраним игру для возможного последующего отката
      pgame->save( "TEMP_FILE");
      string move;
      move = getPlayerInput( "Enter your move:");
      pgame->make_move( move);
    }
};

class UndoCommand: public Command
{
  public:
    UndoCommand( Game * p ) : Command( p) {}
    void execute() {
      // Восстановим игру из временного файла
      pgame->open( "TEMP_FILE");
    }
};


int main()
{
  Game game;
  // Имитация действий игрока
```

```
  vector<Command*> v;
  // Создаем новую игру
  v.push_back( new CreateGameCommand( &game));
  // Делаем несколько ходов
  v.push_back( new MakeMoveCommand( &game));
  v.push_back( new MakeMoveCommand( &game));
  // Последний ход отменяем
  v.push_back( new UndoCommand( &game));
  // Сохраняем игру
  v.push_back( new SaveGameCommand( &game));

  for (size_t i=0; i<v.size(); ++i)
    v[i]->execute();

  for (size_t i=0; i<v.size(); ++i)
    delete v[i];

  return 0;
}
```

Вывод программы:

```
1 Create game
2 Save game in TEMP_FILE
3 Enter your move: E2-E4
4 Make move E2-E4
5 Save game in TEMP_FILE
6 Enter your move: D2-D3
7 Make move D2-D3
8 Open game from TEMP_FILE
9 Enter file name: game1.sav
10Save game in game1.sav
```

Паттерн Composite (компоновщик)

Назначение паттерна Composite

Используйте паттерн Composite если:

- Необходимо объединять группы схожих объектов и управлять ими.
- Объекты могут быть как примитивными (элементарными), так и составными (сложными). Составной объект может включать в себя коллекции других объектов, образуя сложные древовидные структуры. Пример: директория файловой системы состоит из элементов, каждый их которых также может быть директорией.
- Код клиента работает с примитивными и составными объектами единообразно.

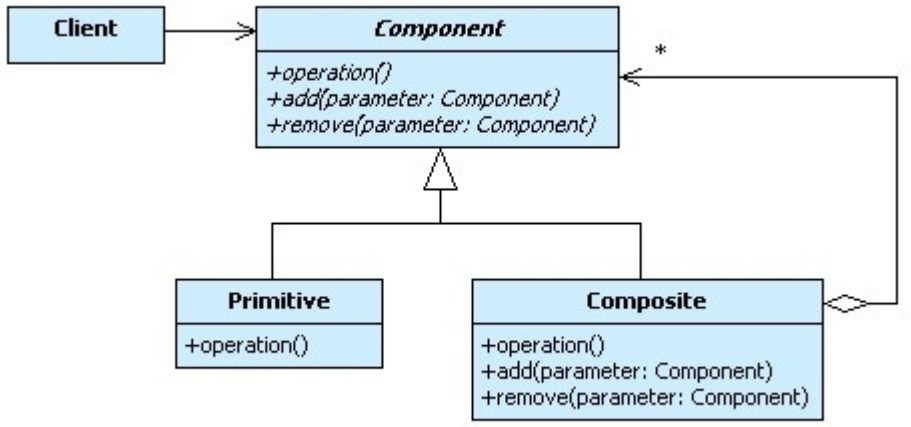
Описание паттерна Composite

Управление группами объектов может быть непростой задачей, особенно, если эти объекты содержат собственные объекты.

Для военной стратегической игры ”Пунические войны ”, описывающей военное противостояние между Римом и Карфагеном (см. раздел Порождающие паттерны), каждая боевая единица (всадник, лучник, пехотинец) имеет свою собственную разрушающую силу. Эти единицы могут объединяться в группы для образования более сложных военных подразделений, например, римские легионы, которые, в свою очередь, объединяясь, образуют целую армию. Как рассчитать боевую мощь таких иерархических соединений?

Паттерн Composite предлагает следующее решение. Он вводит абстрактный базовый класс `Component` с поведением, общим для всех примитивных и составных объектов. Для случая стратегической игры - это метод `getStrength()` для подсчета разрушающей силы. Подклассы `Primitive` and `Composite` являются производными от класса `Component`. Составной объект `Composite` хранит компоненты-потомки абстрактного типа `Component`, каждый из которых может быть также `Composite`.

UML-диаграмма классов паттерна Composite



Для добавления или удаления объектов-потомков в составной объект `Composite`, класс `Component` определяет интерфейсы `add()` и `remove()`.

Реализация паттерна Composite

Применим паттерн Composite для нашей стратегической игры. Сначала сформируем различные военные соединения римской армии, а затем рассчитаем разрушающую силу.

```
#include <iostream>
#include <vector>
#include <assert.h>

// Component
class Unit
{
  public:
    virtual int getStrength() = 0;
    virtual void addUnit(Unit* p) {
      assert( false);
    }
    virtual ~Unit() {}
};

// Primitives
class Archer: public Unit
{
  public:
    virtual int getStrength() {
      return 1;
    }
};

class Infantryman: public Unit
{
  public:
    virtual int getStrength() {
      return 2;
    }
};

class Horseman: public Unit
{
  public:
    virtual int getStrength() {
      return 3;
    }
};


// Composite
class CompositeUnit: public Unit
{
  public:
    int getStrength() {
      int total = 0;
      for(int i=0; i<c.size(); ++i)
```

```
        total += c[i]->getStrength();
      return total;
    }
    void addUnit(Unit* p) {
        c.push_back( p);
    }
    ~CompositeUnit() {
      for(int i=0; i<c.size(); ++i)
        delete c[i];
    }
  private:
   std::vector<Unit*> c;
};


// Вспомогательная функция для создания легиона
CompositeUnit* createLegion()
{
  // Римский легион содержит:
  CompositeUnit* legion = new CompositeUnit;
  // 3000 тяжелых пехотинцев
  for (int i=0; i<3000; ++i)
    legion->addUnit(new Infantryman);
  // 1200 легких пехотинцев
  for (int i=0; i<1200; ++i)
    legion->addUnit(new Archer);
  // 300 всадников
  for (int i=0; i<300; ++i)
    legion->addUnit(new Horseman);

  return legion;
}

int main()
{
  // Римская армия состоит из 4-х легионов
  CompositeUnit* army = new CompositeUnit;
  for (int i=0; i<4; ++i)
    army->addUnit( createLegion());

  cout << "Roman army damaging strength is "
       << army->getStrength() << endl;
  // …
  delete army;
  return 0;
}
```

Следует обратить внимание на один важный момент. Абстрактный базовый класс `Unit` объявляет интерфейс для добавления новых боевых единиц addUnit(), несмотря на то, что

объектам примитивных типов (`Archer`, `Infantryman`, `Horseman`) подобная операция не нужна. Сделано это в угоду прозрачности системы в ущерб ее безопасности. Клиент знает, что объект типа `Unit` всегда будет иметь метод `addUnit()`. Однако его вызов для примитивных объектов считается ошибочным и небезопасным.

Можно сделать систему более безопасной, переместив метод `addUnit()` в составной объект `CompositeUnit`. Однако при этом возникает следующая проблема: мы не знаем, содержит ли объект `Unit` метод `addUnit()`.

Рассмотрим следующий фрагмент кода.

```
class Unit
{
  public:
    virtual CompositeUnit* getComposite() {
      return 0;
    }
    // …
};

// Composite
class CompositeUnit: public Unit
{
  public:
   void addUnit(Unit* p);
    CompositeUnit* getComposite() {
      return this;
    }
    // …
};
```

В абстрактном базовом классе `Unit` появился новый виртуальный метод `getComposite()` с реализацией по умолчанию, которая возвращает 0. Класс `CompositeUnit` переопределяет этот метод, возвращая указатель на самого себя. Благодаря этому методу можно запросить у компонента его тип. Если он составной, то можно применить операцию `addUnit()`.

```
if (unit->getComposite())
{
   unit->getComposite()->addUnit( new Archer);
}
```

Результаты применения паттерна Composite

Достоинства паттерна Composite

- В систему легко добавлять новые примитивные или составные объекты, так как паттерн Composite использует общий базовый класс `Component`.
- Код клиента имеет простую структуру – примитивные и составные объекты

обрабатываются одинаковым образом.
- Паттерн Composite позволяет легко обойти все узлы древовидной структуры

Недостатки паттерна Composite

- Неудобно осуществить запрет на добавление в составной объект `Composite` объектов определенных типов. Так, например, в состав римской армии не могут входить боевые слоны.

Паттерн Decorator (декоратор, wrapper, обертка)

Назначение паттерна Decorator

- Паттерн Decorator динамически добавляет новые обязанности объекту. Декораторы являются гибкой альтернативой порождению подклассов для расширения функциональности.
- Рекурсивно декорирует основной объект.
- Паттерн Decorator использует схему "обертываем подарок, кладем его в коробку, обертываем коробку".

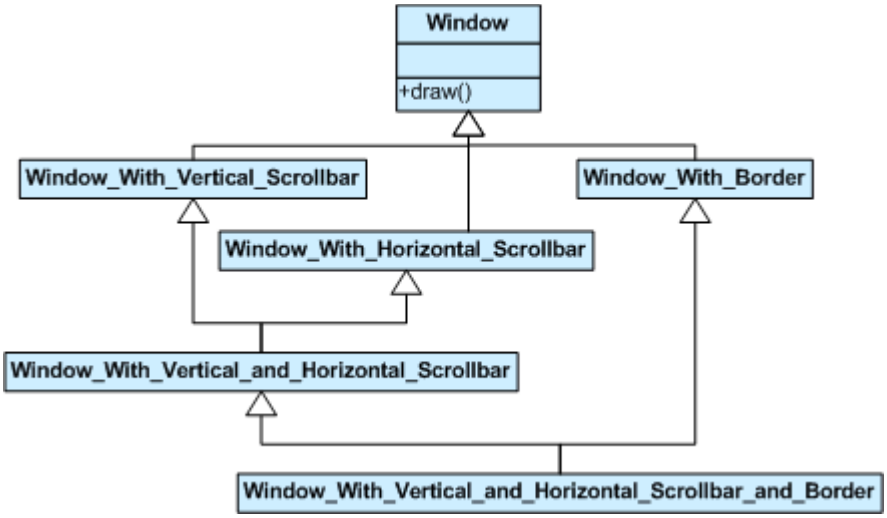
Решаемая проблема

Вы хотите добавить новые обязанности в поведении или состоянии отдельных объектов во время выполнения программы. Использование наследования не представляется возможным, поскольку это решение статическое и распространяется целиком на весь класс.

Обсуждение паттерна Decorator

Предположим, вы работаете над библиотекой для построения графических пользовательских интерфейсов и хотите иметь возможность добавлять в окно рамку и полосу прокрутки. Тогда вы могли бы определить иерархию наследования следующим образом...

UML-диаграмма классов паттерна Decorator

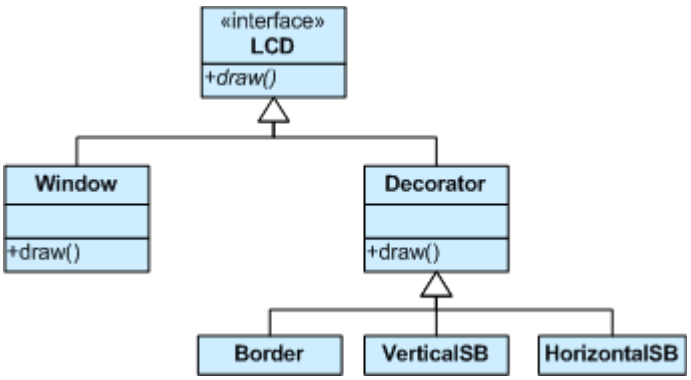


Эта схема имеет существенный недостаток - число классов сильно разрастается.

Паттерн Decorator дает клиенту возможность задавать любые комбинации желаемых "особенностей".

```
1Widget*  aWidget = new BorderDecorator(
2                         new HorizontalScrollBarDecorator(
3                           new VerticalScrollBarDecorator(
4                             new Window( 80, 24 ))));
5aWidget->draw();
```

Гибкость может быть достигнута следующим дизайном.



Другой пример каскадного соединения свойств (в цепочку) для придания объекту нужных характеристик…

```
1Stream*  aStream = new CompressingStream(
2                         new ASCII7Stream(
3                           new FileStream( "fileName.dat" )));
4aStream->putString( "Hello world" );
```

Решение этого класса задач предполагает инкапсуляцию исходного объекта в абстрактный интерфейс. Как объекты-декораторы, так и основной объект наследуют от этого абстрактного интерфейса. Интерфейс использует рекурсивную композицию для добавления к основному

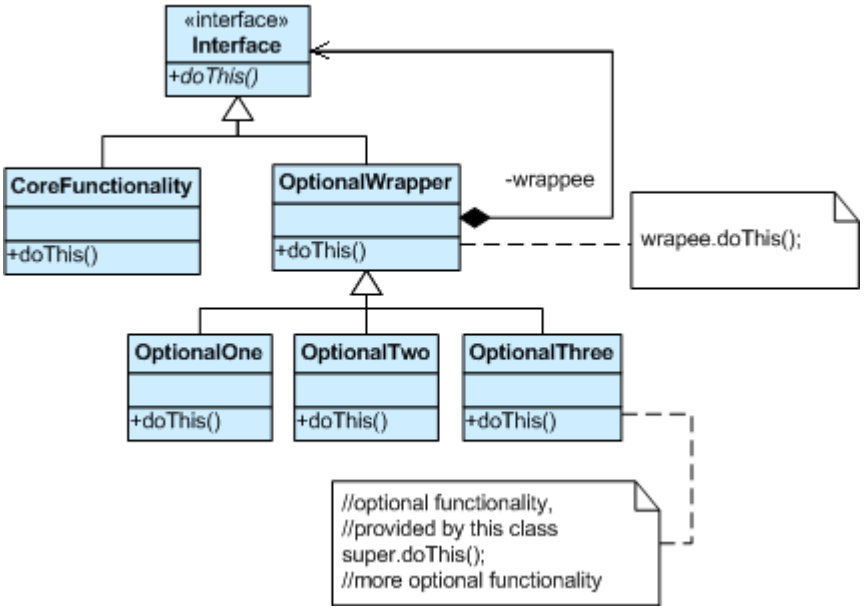
объекту неограниченного количества "слоев" - декораторов.

Обратите внимание, паттерн Decorator позволяет добавлять объекту новые обязанности, не изменяя его интерфейс (новые методы не добавляются). Известный клиенту интерфейс должен оставаться постоянным на всех, следующих друг за другом "слоях".

Отметим также, что основной объект теперь "скрыт" внутри объекта-декоратора. Доступ к основному объекту теперь проблематичен.

Структура паттерна Decorator

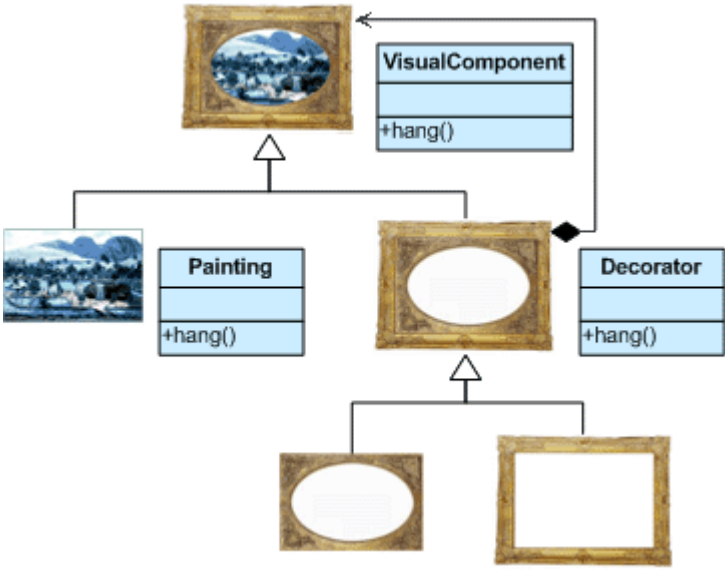
Клиент всегда заинтересован в функциональности `CoreFunctionality.doThis()`. Клиент может или не может быть заинтересован в методах `OptionalOne.doThis()` и `OptionalTwo.doThis()`. Каждый из этих классов переадресует запрос базовому классу `Decorator`, а тот направляет его в декорируемый объект.



Пример паттерна Decorator

Паттерн Decorator динамически добавляет новые обязанности объекту. Украшения для новогодней елки являются примерами декораторов. Огни, гирлянды, игрушки и т.д. вешают на елку для придания ей праздничного вида. Украшения не меняют саму елку, а только делают ее новогодней.

Хотя картины можно повесить на стену и без рамок, рамки часто добавляются для придания нового стиля.



Использование паттерна Decorator

- Подготовьте исходные данные: один основной компонент и несколько дополнительных (необязательных) "оберток".
- Создайте общий для всех классов интерфейс по принципу "наименьшего общего знаменателя НОЗ" (lowest common denominator LCD). Этот интерфейс должен делать все классы взаимозаменяемыми.
- Создайте базовый класс второго уровня (`Decorator`) для поддержки дополнительных декорирующих классов.
- Основной класс и класс `Decorator` наследуют общий НОЗ-интерфейс.
- Класс `Decorator` использует отношение композиции. Указатель на НОЗ-объект инициализируется в конструкторе.
- Класс `Decorator` делегирует выполнение операции НОЗ-объекту.
- Для реализации каждой дополнительной функциональности создайте класс, производный от `Decorator`.
- Подкласс `Decorator` реализует дополнительную функциональность и делегирует выполнение операции базовому классу `Decorator`.
- Клиент несет ответственность за конфигурирование системы: устанавливает типы и последовательность использования основного объекта и декораторов.

Особенности паттерна Decorator

- Adapter придает своему объекту новый интерфейс, Proxy предоставляет тот же интерфейс, а Decorator обеспечивает расширенный интерфейс.
- Adapter изменяет интерфейс объекта. Decorator расширяет ответственность объекта. Decorator, таким образом, более прозрачен для клиента. Как следствие, Decorator

поддерживает рекурсивную композицию, что невозможно с чистыми адаптерами.

- Decorator можно рассматривать как вырожденный случай Composite с единственным компонентом. Однако Decorator добавляет новые обязанности и не предназначен для агрегирования объектов.
- Decorator позволяет добавлять новые функции к объектам без наследования. Composite фокусирует внимание на представлении, а не декорировании. Эти характеристики являются различными, но взаимодополняющими, поэтому Composite и Decorator часто используются вместе.
- Decorator и Proxy имеют разное назначение, но схожие структуры. Их реализации хранят ссылку на объект, которому они отправляют запросы.
- Decorator позволяет изменить внешний облик объекта, Strategy – его внутреннее содержание.

Реализация паттерна Decorator

Паттерн Decorator: до и после

До

Используется следующая иерархия наследования:

```
class A {
  public:
    virtual void do_it() {
        cout << 'A';
    }
};

class AwithX: public A {
  public:
    /*virtual*/
    void do_it() {
        A::do_it();
        do_X();
    };
  private:
    void do_X() {
        cout << 'X';
    }
};

class AwithY: public A {
  public:
    /*virtual*/
    void do_it() {
        A::do_it();
```

```
        do_Y();
    }
  protected:
    void do_Y() {
        cout << 'Y';
    }
};

class AwithZ: public A {
  public:
    /*virtual*/
    void do_it() {
        A::do_it();
        do_Z();
    }
  protected:
    void do_Z() {
        cout << 'Z';
    }
};

class AwithXY: public AwithX, public AwithY
{
  public:
    /*virtual*/
    void do_it() {
        AwithX::do_it();
        AwithY::do_Y();
    }
};

class AwithXYZ: public AwithX, public AwithY, public AwithZ
{
  public:
    /*virtual*/
    void do_it() {
        AwithX::do_it();
        AwithY::do_Y();
        AwithZ::do_Z();
    }
};

int main() {
  AwithX anX;
  AwithXY anXY;
  AwithXYZ anXYZ;
  anX.do_it();
  cout << '\n';
  anXY.do_it();
```

```
75  cout << '\n';
76  anXYZ.do_it();
77  cout << '\n';
78}
```

Вывод программы:

```
1AX
2AXY
3AXYZ
```

После

Заменим наследование делегированием.

Обсуждение. Используйте агрегирование вместо наследования для декорирования "основного" объекта. Тогда клиент сможет динамически добавлять новые обязанности объектам, в то время как архитектура, основанная на множественном наследовании, является статичной.

```
class I {
  public:
    virtual ~I(){}
    virtual void do_it() = 0;
};

class A: public I {
  public:
    ~A() {
        cout << "A dtor" << '\n';
    }
    /*virtual*/
    void do_it() {
        cout << 'A';
    }
};

class D: public I {
  public:
    D(I *inner) {
        m_wrappee = inner;
    }
    ~D() {
        delete m_wrappee;
    }
    /*virtual*/
    void do_it() {
        m_wrappee->do_it();
```

```
    }
  private:
    I *m_wrappee;
};

class X: public D {
  public:
    X(I *core): D(core){}
    ~X() {
        cout << "X dtor" << "   ";
    }
    /*virtual*/
    void do_it() {
        D::do_it();
        cout << 'X';
    }
};

class Y: public D {
  public:
    Y(I *core): D(core){}
    ~Y() {
        cout << "Y dtor" << "   ";
    }
    /*virtual*/
    void do_it() {
        D::do_it();
        cout << 'Y';
    }
};

class Z: public D {
  public:
    Z(I *core): D(core){}
    ~Z() {
        cout << "Z dtor" << "   ";
    }
    /*virtual*/
    void do_it() {
        D::do_it();
        cout << 'Z';
    }
};

int main() {
  I *anX = new X(new A);
  I *anXY = new Y(new X(new A));
  I *anXYZ = new Z(new Y(new X(new A)));
```

```
  anX->do_it();
  cout << '\n';
  anXY->do_it();
  cout << '\n';
  anXYZ->do_it();
  cout << '\n';
  delete anX;
  delete anXY;
  delete anXYZ;
}
```

Вывод программы:

```
1AX
2AXY
3AXYZ
4X dtor   A dtor
5Y dtor   X dtor   A dtor
6Z dtor   Y dtor   X dtor   A dtor
```

Паттерн проектирования Decorator по шагам

- Создайте "наименьший общий знаменатель", делающий классы взаимозаменяемыми.
- Создайте базовый класс второго уровня для реализации дополнительной функциональности.
- Основной класс и класс-декоратор используют отношение "является".
- Класс-декоратор "имеет" экземпляр "наименьшего общего знаменателя".
- Класс `Decorator` делегирует выполнение операции объекту "имеет".
- Для реализации каждой дополнительной функциональности создайте подклассы `Decorator`.
- Подклассы `Decorator` делегируют выполнение операции базовому классу и реализуют дополнительную функциональность.
- Клиент несет ответственность за конфигурирование нужной функциональности.

```
#include <iostream>
using namespace std;

// 1. " Наименьший общий знаменатель"
class Widget
{
  public:
    virtual void draw() = 0;
};

// 3. Основной класс, использующий отношение "является"
```

```
class TextField: public Widget
{
    int width, height;
  public:
    TextField(int w, int h)
    {
        width = w;
        height = h;
    }

    /*virtual*/
    void draw()
    {
        cout << "TextField: " << width << ", " << height << '\n';
    }
};

// 2. Базовый класс второго уровня
class Decorator: public Widget  // 3. использует отношение "является"
{
    Widget *wid; // 4. отношение "имеет"
  public:
    Decorator(Widget *w)
    {
        wid = w;
    }

    /*virtual*/
    void draw()
    {
        wid->draw(); // 5. делегирование
    }
};

// 6. Дополнительное декорирование
class BorderDecorator: public Decorator
{
  public:
    BorderDecorator(Widget *w): Decorator(w){}

    /*virtual*/
    void draw()
    {
        // 7. Делегирование базовому классу и
        Decorator::draw();
        // 7. реализация дополнительной функциональности
        cout << "   BorderDecorator" << '\n';
    }
```

```
};

// 6. Дополнительное декорирование
class ScrollDecorator: public Decorator
{
  public:
    ScrollDecorator(Widget *w): Decorator(w){}

    /*virtual*/
    void draw()
    {
        // 7. Delegate to base class and add extra stuff
        Decorator::draw();
        cout << "   ScrollDecorator" << '\n';
    }
};

int main()
{
  // 8. Клиент ответственен за конфигурирование нужной
функциональности
  Widget *aWidget = new BorderDecorator(
                      new BorderDecorator(
                        new ScrollDecorator
                          (new TextField(80, 24))));
  aWidget->draw();
}
TextField: 80, 24
   ScrollDecorator
   BorderDecorator
   BorderDecorator
```

Паттерн Facade (фасад)

Назначение паттерна Facade

- Паттерн Facade предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Facade определяет интерфейс более высокого уровня, упрощающий использование подсистемы.
- Паттерн Facade "обертывает" сложную подсистему более простым интерфейсом.

Решаемая проблема

Клиенты хотят получить упрощенный интерфейс к общей функциональности сложной

подсистемы.

Обсуждение паттерна Facade

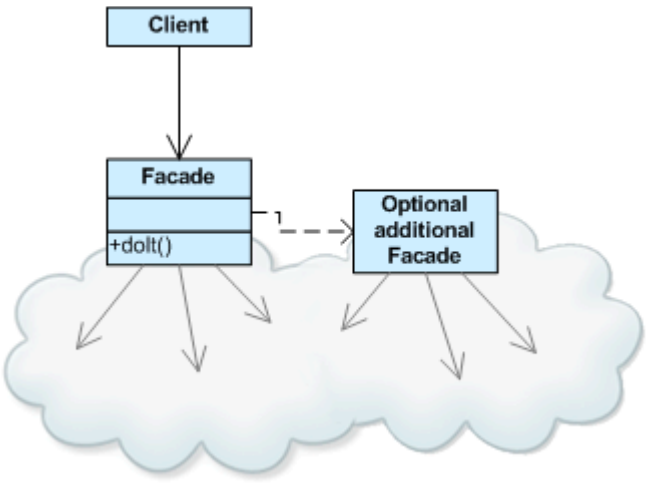
Паттерн Facade инкапсулирует сложную подсистему в единственный интерфейсный объект. Это позволяет сократить время изучения подсистемы, а также способствует уменьшению степени связанности между подсистемой и потенциально большим количеством клиентов. С другой стороны, если фасад является единственной точкой доступа к подсистеме, то он будет ограничивать возможности, которые могут понадобиться "продвинутым" пользователям.

Объект Facade, реализующий функции посредника, должен оставаться довольно простым и не быть всезнающим "оракулом".

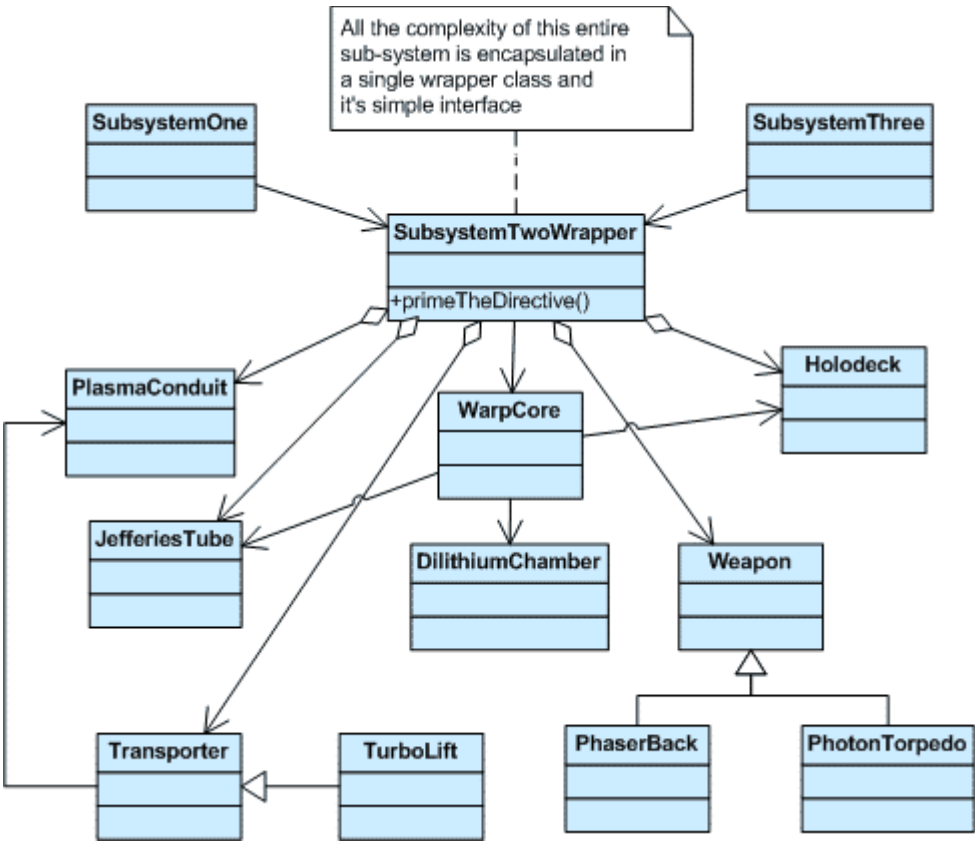
Структура паттерна Facade

Клиенты общаются с подсистемой через Facade. При получении запроса от клиента объект Facade переадресует его нужному компоненту подсистемы. Для клиентов компоненты подсистемы остаются "тайной, покрытой мраком".

UML-диаграмма классов паттерна Facade

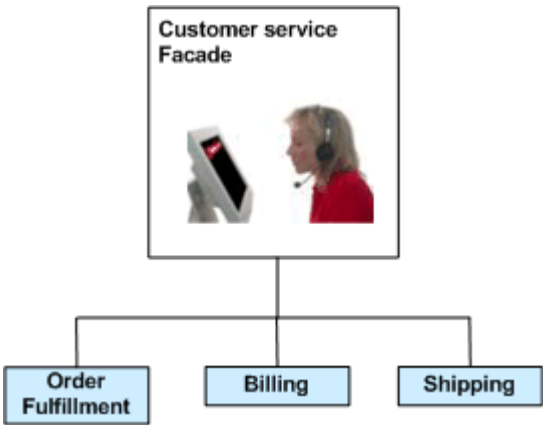


Подсистемы SubsystemOne и SubsystemThree не взаимодействуют напрямую с внутренними компонентами подсистемы SubsystemTwo. Они используют "фасад" SubsystemTwoWrapper (т.е. абстракцию более высокого уровня).



Пример паттерна Facade

Паттерн Facade определяет унифицированный высокоуровневый интерфейс к подсистеме, что упрощает ее использование. Покупатели сталкиваются с фасадом при заказе каталожных товаров по телефону. Покупатель звонит в службу поддержки клиентов и перечисляет товары, которые хочет приобрести. Представитель службы выступает в качестве "фасада", обеспечивая интерфейс к отделу исполнения заказов, отделу продаж и службе доставки.



Использование паттерна Facade

- Определите для подсистемы простой, унифицированный интерфейс.

- Спроектируйте класс "обертку", инкапсулирующий подсистему.
- Вся сложность подсистемы и взаимодействие ее компонентов скрыты от клиентов. "Фасад" / "обертка" переадресует пользовательские запросы подходящим методам подсистемы.
- Клиент использует только "фасад".
- Рассмотрите вопрос о целесообразности создания дополнительных "фасадов".

Особенности паттерна Facade

- Facade определяет новый интерфейс, в то время как Adapter использует уже имеющийся. Помните, Adapter делает работающими вместе два существующих интерфейса, не создавая новых.
- Если Flyweight показывает, как сделать множество небольших объектов, то Facade показывает, как сделать один объект, представляющий целую подсистему.
- Mediator похож на Facade тем, что абстрагирует функциональность существующих классов. Однако Mediator централизует функциональность между объектами-коллегами, не присущую ни одному из них. Коллеги обмениваются информацией друг с другом через Mediator. С другой стороны, Facade определяет простой интерфейс к подсистеме, не добавляет новой функциональности и не известен классам подсистемы.
- Abstract Factory может применяться как альтернатива Facade для сокрытия платформенно-зависимых классов.
- Объекты "фасадов" часто являются Singleton, потому что требуется только один объект Facade.
- Adapter и Facade в являются "обертками", однако эти "обертки" разных типов. Цель Facade – создание более простого интерфейса, цель Adapter – адаптация существующего интерфейса. Facade обычно "обертывает" несколько объектов, Adapter "обертывает" один объект.

Реализация паттерна Facade

Разбиение системы на компоненты позволяет снизить ее сложность. Ослабить связи между компонентами системы можно с помощью паттерна Facade. Объект "фасад" предоставляет единый упрощенный интерфейс к компонентам системы.

В примере ниже моделируется система сетевого обслуживания. Фасад FacilitiesFacade скрывает внутреннюю структуру системы. Пользователь, сделав однажды запрос на обслуживание, затем 1-2 раза в неделю в течение 5 месяцев справляется о ходе выполнения работ до тех пор, пока его запрос не будет полностью обслужен.

```
#include <iostream.h>

class MisDepartment
{
  public:
    void submitNetworkRequest()
    {
        _state = 0;
    }
    bool checkOnStatus()
    {
        _state++;
        if (_state == Complete)
          return 1;
        return 0;
    }
  private:
    enum States
    {
        Received, DenyAllKnowledge, ReferClientToFacilities,
        FacilitiesHasNotSentPaperwork, ElectricianIsNotDone,
        ElectricianDidItWrong, DispatchTechnician, SignedOff,
        DoesNotWork, FixElectriciansWiring, Complete
    };
    int _state;
};

class ElectricianUnion
{
  public:
    void submitNetworkRequest()
    {
        _state = 0;
    }
    bool checkOnStatus()
    {
        _state++;
        if (_state == Complete)
          return 1;
        return 0;
    }
  private:
    enum States
    {
        Received, RejectTheForm, SizeTheJob, SmokeAndJokeBreak,
        WaitForAuthorization, DoTheWrongJob, BlameTheEngineer,
        WaitToPunchOut, DoHalfAJob, ComplainToEngineer,
        GetClarification, CompleteTheJob, TurnInThePaperwork,
```

```
        Complete
    };
    int _state;
};

class FacilitiesDepartment
{
  public:
    void submitNetworkRequest()
    {
        _state = 0;
    }
    bool checkOnStatus()
    {
        _state++;
        if (_state == Complete)
          return 1;
        return 0;
    }
  private:
    enum States
    {
        Received, AssignToEngineer, EngineerResearches,
        RequestIsNotPossible, EngineerLeavesCompany,
        AssignToNewEngineer, NewEngineerResearches,
        ReassignEngineer,EngineerReturns,
        EngineerResearchesAgain, EngineerFillsOutPaperWork,
        Complete
    };
    int _state;
};

class FacilitiesFacade
{
  public:
    FacilitiesFacade()
    {
        _count = 0;
    }
    void submitNetworkRequest()
    {
        _state = 0;
    }
    bool checkOnStatus()
    {
        _count++;
        /* Запрос на обслуживание получен */
        if (_state == Received)
```

```
        {
            _state++;
            /* Перенаправим запрос инженеру */
            _engineer.submitNetworkRequest();
            cout << "submitted to Facilities - " << _count
                 << " phone calls so far" << endl;
        }
        else if (_state == SubmitToEngineer)
        {
            /* Если инженер свою работу выполнил,
               перенаправим запрос электрику */
            if (_engineer.checkOnStatus())
            {
                _state++;
                _electrician.submitNetworkRequest();
                cout << "submitted to Electrician - " << _count
                     << " phone calls so far" << endl;
            }
        }
        else if (_state == SubmitToElectrician)
        {
            /* Если электрик свою работу выполнил,
               перенаправим запрос технику */
            if (_electrician.checkOnStatus())
            {
                _state++;
                _technician.submitNetworkRequest();
                cout << "submitted to MIS - " << _count
                     << " phone calls so far" << endl;
            }
        }
        else if (_state == SubmitToTechnician)
        {
            /* Если техник свою работу выполнил,
               то запрос обслужен до конца */
            if (_technician.checkOnStatus())
              return 1;
        }
        /* Запрос еще не обслужен до конца */
        return 0;
    }
    int getNumberOfCalls()

    {
        return _count;
    }
private:
    enum States
```

```
    {
        Received, SubmitToEngineer, SubmitToElectrician,
        SubmitToTechnician
    };
    int _state;
    int _count;
    FacilitiesDepartment _engineer;
    ElectricianUnion _electrician;
    MisDepartment _technician;
};

int main()
{
  FacilitiesFacade facilities;

  facilities.submitNetworkRequest();
  /* Звоним, пока работа не выполнена полностью */
  while(!facilities.checkOnStatus())
    ;
  cout << "job completed after only "
       << facilities.getNumberOfCalls()
       << " phone calls" << endl;
}
```

Вывод программы:

```
submitted to Facilities - 1 phone calls so far
submitted to Electrician - 12 phone calls so far
submitted to MIS - 25 phone calls so far
job completed after only 35 phone calls
```

Паттерн Factory Method (фабричный метод)

Назначение паттерна Factory Method

В системе часто требуется создавать объекты самых разных типов. Паттерн Factory Method (фабричный метод) может быть полезным в решении следующих задач:

- Система должна оставаться расширяемой путем добавления объектов новых типов. Непосредственное использование выражения new является нежелательным, так как в этом случае код создания объектов с указанием конкретных типов может получиться разбросанным по всему приложению. Тогда такие операции как добавление в систему объектов новых типов или замена объектов одного типа на другой будут затруднительными (подробнее в разделе Порождающие паттерны). Паттерн Factory Method позволяет системе оставаться независимой как от самого процесса порождения

объектов, так и от их типов.
- Заранее известно, когда нужно создавать объект, но неизвестен его тип.

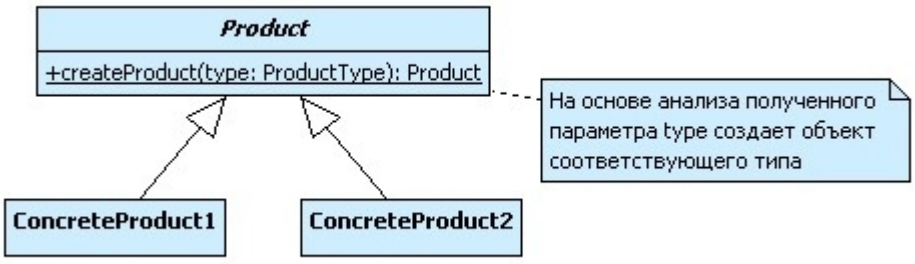
Описание паттерна Factory Method

Для того, чтобы система оставалась независимой от различных типов объектов, паттерн Factory Method использует механизм полиморфизма - классы всех конечных типов наследуют от одного абстрактного базового класса, предназначенного для полиморфного использования. В этом базовом классе определяется единый интерфейс, через который пользователь будет оперировать объектами конечных типов.

Для обеспечения относительно простого добавления в систему новых типов паттерн Factory Method локализует создание объектов конкретных типов в специальном классе-фабрике. Методы этого класса, посредством которых создаются объекты конкретных классов, называются фабричными. Существуют две разновидности паттерна Factory Method:

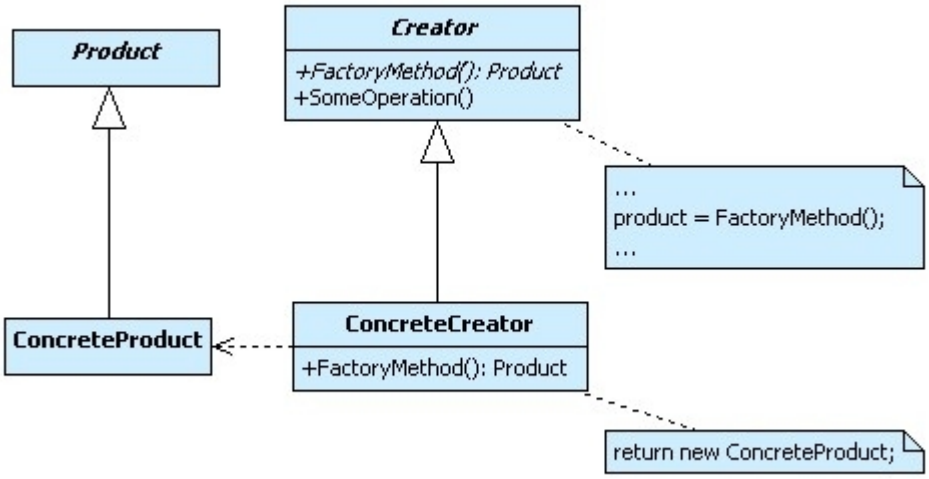
Обобщенный конструктор, когда в том же самом полиморфном базовом классе, от которого наследуют производные классы всех создаваемых в системе типов, определяется статический фабричный метод. В качестве параметра в этот метод должен передаваться идентификатор типа создаваемого объекта.

UML-диаграмма классов паттерна Factory Method. Обобщенный конструктор



Классический вариант фабричного метода, когда интерфейс фабричных методов объявляется в независимом классе-фабрике, а их реализация определяется конкретными подклассами этого класса.

UML-диаграмма классов паттерна Factory Method. Классическая реализация



Реализация паттерна Factory Method

Рассмотрим оба варианта реализации паттерна Factory Method на примере процесса порождения военных персонажей для нашей стратегической игры. Ее подробное описание можно найти в разделе Порождающие паттерны. Для упрощения демонстрационного кода будем создавать военные персонажи для некой абстрактной армии без учета особенностей воюющих сторон.

Реализация паттерна Factory Method на основе обобщенного конструктора

```
// #include <iostream>
#include <vector>

enum Warrior_ID { Infantryman_ID=0, Archer_ID, Horseman_ID };

// Иерархия классов игровых персонажей
class Warrior
{
  public:
    virtual void info() = 0;
    virtual ~Warrior() {}
    // Параметризированный статический фабричный метод
    static Warrior* createWarrior( Warrior_ID id );
};

class Infantryman: public Warrior
{
  public:
```

```
    void info() {
      cout << "Infantryman" << endl;
    }
};

class Archer: public Warrior
{
  public:
    void info() {
      cout << "Archer" << endl;
    }
};

class Horseman: public Warrior
{
  public:
    void info() {
      cout << "Horseman" << endl;
    }
};


// Реализация параметризированного фабричного метода
Warrior* Warrior::createWarrior( Warrior_ID id )
{
    Warrior * p;
    switch (id)
    {
        case Infantryman_ID:
            p = new Infantryman();
            break;
        case Archer_ID:
            p = new Archer();
            break;
        case Horseman_ID:
            p = new Horseman();
            break;
        default:
            assert( false);
    }
    return p;
};


// Создание объектов при помощи параметризированного фабричного
метода
int main()
```

```
{
    vector<Warrior*> v;
    v.push_back( Warrior::createWarrior( Infantryman_ID));
    v.push_back( Warrior::createWarrior( Archer_ID));
    v.push_back( Warrior::createWarrior( Horseman_ID));

    for(int i=0; i<v.size(); i++)
        v[i]->info();
    // ...
}
```

Представленный вариант паттерна Factory Method пользуется популярностью благодаря своей простоте. В нем статический фабричный метод `createWarrior()` определен непосредственно в полиморфном базовом классе `Warrior`. Этот фабричный метод является параметризированным, то есть для создания объекта некоторого типа в `createWarrior()` передается соответствующий идентификатор типа.

С точки зрения "чистоты" объектно-ориентированного кода у этого варианта есть следующие недостатки:

- Так как код по созданию объектов всех возможных типов сосредоточен в статическом фабричном методе класса `Warrior`, то базовый класс `Warrior` обладает знанием обо всех производных от него классах, что является нетипичным для объектно-ориентированного подхода.
- Подобное использование оператора `switch` (как в коде фабричного метода `createWarrior()`) в объектно-ориентированном программировании также не приветствуется.

Указанные недостатки отсутствуют в классической реализации паттерна Factory Method.

Классическая реализация паттерна Factory Method

```
//
#include <iostream>
#include <vector>

// Иерархия классов игровых персонажей
class Warrior
{
  public:
    virtual void info() = 0;
    virtual ~Warrior() {}
};

class Infantryman: public Warrior
{
  public:
```

```
    void info() {
      cout << "Infantryman" << endl;
    };
};

class Archer: public Warrior
{
  public:
    void info() {
      cout << "Archer" << endl;
    };
};

class Horseman: public Warrior
{
  public:
    void info() {
      cout << "Horseman" << endl;
    };
};


// Фабрики объектов
class Factory
{
  public:
    virtual Warrior* createWarrior() = 0;
    virtual ~Factory() {}
};

class InfantryFactory: public Factory
{
  public:
    Warrior* createWarrior() {
      return new Infantryman;
    }
};

class ArchersFactory: public Factory
{
  public:
    Warrior* createWarrior() {
      return new Archer;
    }
};

class CavalryFactory: public Factory
```

```
{
  public:
    Warrior* createWarrior() {
      return new Horseman;
    }
};


// Создание объектов при помощи фабрик объектов
int main()
{
    InfantryFactory* infantry_factory = new InfantryFactory;
    ArchersFactory*  archers_factory  = new ArchersFactory ;
    CavalryFactory*  cavalry_factory  = new CavalryFactory ;

    vector<Warrior*> v;
    v.push_back( infantry_factory->createWarrior());
    v.push_back( archers_factory->createWarrior());
    v.push_back( cavalry_factory->createWarrior());

    for(int i=0; i<v.size(); i++)
        v[i]->info();
    // ...
}
```

Классический вариант паттерна Factory Method использует идею полиморфной фабрики. Специально выделенный для создания объектов полиморфный базовый класс `Factory` объявляет интерфейс фабричного метода `createWarrior()`, а производные классы его реализуют.

Представленный вариант паттерна Factory Method является наиболее распространенным, но не единственным. Возможны следующие вариации:

- Класс `Factory` имеет реализацию фабричного метода `createWarrior()` по умолчанию.
- Фабричный метод `createWarrior()` класса `Factory` параметризирован типом создаваемого объекта (как и у представленного ранее, простого варианта Factory Method) и имеет реализацию по умолчанию. В этом случае, производные от `Factory` классы необходимы лишь для того, чтобы определить нестандартное поведение `createWarrior()`.

Результаты применения паттерна Factory Method

Достоинства паттерна Factory Method

- Создает объекты разных типов, позволяя системе оставаться независимой как от самого

процесса создания, так и от типов создаваемых объектов.

Недостатки паттерна Factory Method

- В случае классического варианта паттерна даже для порождения единственного объекта необходимо создавать соответствующую фабрику

Паттерн Flyweight (приспособленец)

Назначение паттерна Flyweight

- Паттерн Flyweight использует разделение для эффективной поддержки большого числа мелких объектов.
- Является стратегией Motif GUI для замены "тяжеловесных" виджетов "легковесными" гаджетами.

Motif–библиотека для разработки приложений с графическим интерфейсом под X Window System. Появилась в конце 1980-х и на данный момент считается устаревшей.

В Motif элементы графического интерфейса (кнопки, полосы прокрутки, меню и т.д.) строятся на основе виджетов. Каждый виджет имеет свое окно. Исторически окна считаются "тяжеловесными" объектами. Если приложение с графическим интерфейсом использует множество виджетов, то производительность системы может упасть. Для решения этой проблемы в Motif предусмотрены гаджеты, являющиеся безоконными аналогами виджетов. Гаджеты управляются менеджерами виджетов, использующих схему "parent-children".

Решаемая проблема

Проектирование системы из объектов самого низкого уровня обеспечивает оптимальную гибкость, но может быть неприемлемо "дорогим" решением с точки зрения производительности и расхода памяти.

Обсуждение паттерна Flyweight

Паттерн Flyweight описывает, как совместно разделять очень мелкие объекты без чрезмерно высоких издержек. Каждый объект-приспособленец имеет две части: внутреннее и внешнее состояния. Внутреннее состояние хранится (разделяется) в приспособленце и состоит из информации, не зависящей от его контекста. Внешнее состояние хранится или вычисляется объектами-клиентами и передается приспособленцу при вызове его методов.

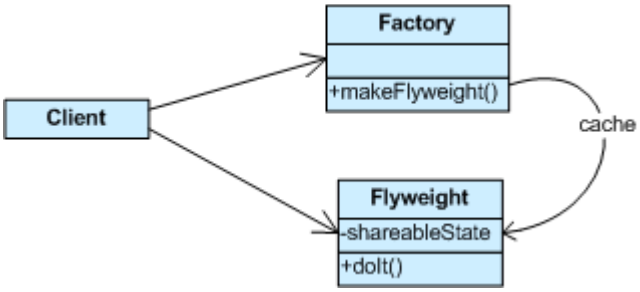
Замена Motif-виджетов "легковесными" гаджетами иллюстрирует этот подход. Если виджет являются достаточно самостоятельным элементом, то гаджет находится в зависимости от своего родительского менеджера компоновки виджетов. Каждый менеджер предоставляет своим

гаджетам контекстно-зависимую информацию по обработке событий, ресурсам. Гаджет хранит в себе только контекстно-независимые данные.

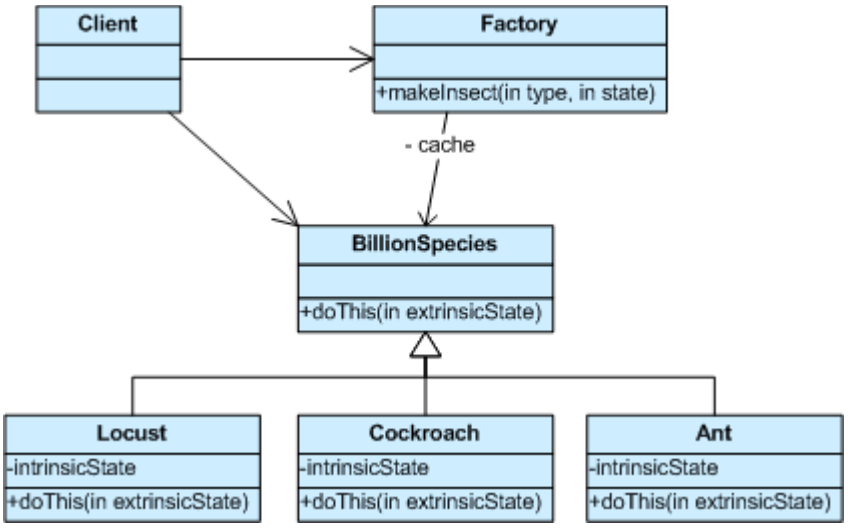
Структура паттерна Flyweight

Клиенты не создают приспособленцев напрямую, а запрашивают их у фабрики. Любые атрибуты (члены данных класса), которые не могут разделяться, являются внешним состоянием. Внешнее состояние передается приспособленцу при вызове его методов. При этом наибольшая экономия памяти достигается в том случае, если внешнее состояние не хранится, а вычисляется при вызове.

UML-диаграмма классов паттерна Flyweight



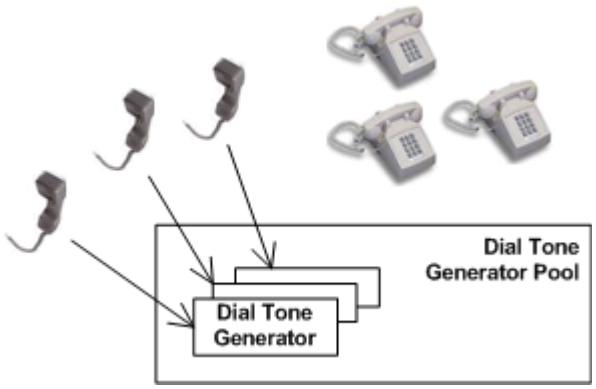
Классы, описывающие различных насекомых Ant, Locust и Cockroach могут быть "легковесными", потому что специфичная для экземпляров информация может быть вынесена наружу и затем, передаваться клиентом в запросе.



Пример паттерна Flyweight

Паттерн Flyweight использует разделение для эффективной поддержки большого числа мелких объектов. Телефонная сеть общего пользования ТФОП является примером Flyweight. Такие ресурсы как генераторы тональных сигналов (Занято, КПВ и т.д.), приемники цифр номера абонента, набираемого в тоновом наборе, являются общими для всех абонентов. Когда абонент

поднимает трубку, чтобы позвонить, ему предоставляется доступ ко всем нужным разделяемым ресурсам.



Использование паттерна Flyweight

- Убедитесь, что существует проблема повышенных накладных расходов.
- Разделите состояние целевого класса на разделяемое (внутреннее) и неразделяемое (внешнее).
- Удалите из атрибутов (членов данных) класса неразделяемое состояние и добавьте его в список аргументов, передаваемых методам.
- Создайте фабрику, которая может кэшировать и повторно использовать существующие экземпляры класса.
- Для создания новых объектов клиент использует эту фабрику вместо оператора new.
- Клиент (или третья сторона) должен находить или вычислять неразделяемое состояние и передавать его методам класса.

Особенности паттерна Flyweight

- Если Flyweight показывает, как сделать множество небольших объектов, то Facade показывает, как представить целую подсистему одним объектом.
- Flyweight часто используется совместно с Composite для реализации иерархической структуры в виде графа с разделяемыми листовыми вершинами.
- Терминальные символы абстрактного синтаксического дерева Interpreter могут разделяться при помощи Flyweight.
- Flyweight объясняет, когда и как могут разделяться объекты State.

Реализация паттерна Flyweight

Паттерн Flyweight: до и после

При использовании объектов на очень низких уровнях детализации накладные расходы могут быть непомерно большими. Использование паттерна Flyweight предполагает удаление

неразделяемого состояния из класса и его передачу клиентом в вызываемые методы. И хотя это накладывает большую ответственность на клиента, но теперь создается значительно меньшее число объектов-приспособленцев. Совместное использование этих экземпляров облегчается с помощью класса-фабрики, поддерживающей "кэш" существующих приспособленцев.

В этом примере, "X" состояние рассматривается как разделяемое, а "Y" состояние выносится наружу (передается клиентом при вызове метода `report()`).

До

```
class Gazillion
{
  public:
    Gazillion()
    {
        m_value_one = s_num / Y;
        m_value_two = s_num % Y;
        ++s_num;
    }
    void report()
    {
        cout << m_value_one << m_value_two << ' ';
    }
    static int X, Y;
  private:
    int m_value_one;
    int m_value_two;
    static int s_num;
};

int Gazillion::X = 6, Gazillion::Y = 10, Gazillion::s_num = 0;

int main()
{
  Gazillion matrix[Gazillion::X][Gazillion::Y];
  for (int i = 0; i < Gazillion::X; ++i)
  {
    for (int j = 0; j < Gazillion::Y; ++j)
      matrix[i][j].report();
    cout << '\n';
  }
}
```

Вывод программы:

```
00 01 02 03 04 05 06 07 08 09
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
```

После

```
class Gazillion
{
  public:
    Gazillion(int value_one)
    {
        m_value_one = value_one;
        cout << "ctor: " << m_value_one << '\n';
    }
    ~Gazillion()
    {
        cout << m_value_one << ' ';
    }
    void report(int value_two)
    {
        cout << m_value_one << value_two << ' ';
    }
  private:
    int m_value_one;
};

class Factory
{
  public:
    static Gazillion *get_fly(int in)
    {
        if (!s_pool[in])
          s_pool[in] = new Gazillion(in);
        return s_pool[in];
    }
    static void clean_up()
    {
        cout << "dtors: ";
        for (int i = 0; i < X; ++i)
          if (s_pool[i])
            delete s_pool[i];
        cout << '\n';
    }
    static int X, Y;
  private:
    static Gazillion *s_pool[];
```

```
};

int Factory::X = 6, Factory::Y = 10;
Gazillion *Factory::s_pool[] =
{
  0, 0, 0, 0, 0, 0
};

int main()
{
  for (int i = 0; i < Factory::X; ++i)
  {
    for (int j = 0; j < Factory::Y; ++j)
      Factory::get_fly(i)->report(j);
    cout << '\n';
  }
  Factory::clean_up();
}
```

Вывод программы:

```
ctor: 0
00 01 02 03 04 05 06 07 08 09
ctor: 1
10 11 12 13 14 15 16 17 18 19
ctor: 2
20 21 22 23 24 25 26 27 28 29
ctor: 3
30 31 32 33 34 35 36 37 38 39
ctor: 4
40 41 42 43 44 45 46 47 48 49
ctor: 5
50 51 52 53 54 55 56 57 58 59
dtors: 0 1 2 3 4 5
```

Паттерн Flyweight: разделение иконок.

Паттерн Flyweight показывает, как эффективно разделять множество мелких объектов. Ключевая концепция - различие между внутренним и внешним состояниями.

Внутреннее состояние состоит из информации, которая не зависит от контекста и может разделяться (например, имя иконки, ее ширина и высота). Оно хранится в приспособленце (то есть в классе `Icon`).

Внешнее состояние не может разделяться, оно зависит от контекста и изменяется вместе с ним (например, координаты верхнего левого угла для каждого экземпляра иконки). Внешнее состояние хранится или вычисляется клиентом и передается приспособленцу при вызове операций. Клиенты не должны создавать экземпляры приспособленцев напрямую, а получать их исключительно из объекта `FlyweightFactory` для правильного разделения.

```
#include <iostream.h>
#include <string.h>

class Icon
{
  public:
    Icon(char *fileName)
    {
        strcpy(_name, fileName);
        if (!strcmp(fileName, "go"))
        {
            _width = 20;
            _height = 20;
        }
        if (!strcmp(fileName, "stop"))
        {
            _width = 40;
            _height = 40;
        }
        if (!strcmp(fileName, "select"))
        {
            _width = 60;
            _height = 60;
        }
        if (!strcmp(fileName, "undo"))
        {
            _width = 30;
            _height = 30;
        }
    }
    const char *getName()
    {
        return _name;
    }
    draw(int x, int y)
    {
        cout << "   drawing " << _name
             << ": upper left (" << x << "," << y
             << ") - lower right (" << x + _width << ","
             << y + _height << ")" << endl;
    }
  private:
    char _name[20];
    int _width;
    int _height;
```

```
};

class FlyweightFactory
{
  public:
    static Icon *getIcon(char *name)
    {
        for (int i = 0; i < _numIcons; i++)
          if (!strcmp(name, _icons[i]->getName()))
            return _icons[i];
        _icons[_numIcons] = new Icon(name);
        return _icons[_numIcons++];
    }
    static void reportTheIcons()
    {
        cout << "Active Flyweights: ";
        for (int i = 0; i < _numIcons; i++)
          cout << _icons[i]->getName() << " ";
        cout << endl;
    }
  private:
    enum
    {
        MAX_ICONS = 5
    };
    static int _numIcons;
    static Icon *_icons[MAX_ICONS];
};

int FlyweightFactory::_numIcons = 0;
Icon *FlyweightFactory::_icons[];

class DialogBox
{
  public:
    DialogBox(int x, int y, int incr):
       _iconsOriginX(x), _iconsOriginY(y), _iconsXIncrement(incr){}
    virtual void draw() = 0;
  protected:
    Icon *_icons[3];
    int _iconsOriginX;
    int _iconsOriginY;
    int _iconsXIncrement;
};

class FileSelection: public DialogBox
{
  public:
```

```
    FileSelection(Icon *first, Icon *second, Icon *third):
        DialogBox(100, 100, 100)
    {
        _icons[0] = first;
        _icons[1] = second;
        _icons[2] = third;
    }
    void draw()
    {
        cout << "drawing FileSelection:" << endl;
        for (int i = 0; i < 3; i++)
          _icons[i]->draw(_iconsOriginX +
                          (i *_iconsXIncrement), _iconsOriginY);
    }
};

class CommitTransaction: public DialogBox
{
  public:
    CommitTransaction(Icon *first, Icon *second, Icon *third):
        DialogBox(150, 150, 150)
    {
        _icons[0] = first;
        _icons[1] = second;
        _icons[2] = third;
    }
    void draw()
    {
        cout << "drawing CommitTransaction:" << endl;
        for (int i = 0; i < 3; i++)
          _icons[i]->draw(_iconsOriginX +
                          (i *_iconsXIncrement), _iconsOriginY);
    }
};

int main()
{
  DialogBox *dialogs[2];
  dialogs[0] = new FileSelection(
                       FlyweightFactory::getIcon("go"),
                       FlyweightFactory::getIcon("stop"),
                       FlyweightFactory::getIcon("select"));
  dialogs[1] = new CommitTransaction(
                       FlyweightFactory::getIcon("select"),
                       FlyweightFactory::getIcon("stop"),
                       FlyweightFactory::getIcon("undo"));

  for (int i = 0; i < 2; i++)
```

```
    dialogs[i]->draw();

  FlyweightFactory::reportTheIcons();
}
```

Вывод программы:

```
drawing FileSelection:
   drawing go: upper left (100,100) - lower right (120,120)
   drawing stop: upper left (200,100) - lower right (240,140)
   drawing select: upper left (300,100) - lower right (360,160)
drawing CommitTransaction:
   drawing select: upper left (150,150) - lower right (210,210)
   drawing stop: upper left (300,150) - lower right (340,190)
   drawing undo: upper left (450,150) - lower right (480,180)
Active Flyweights: go stop select undo
```

Паттерн Interpreter (интерпетатор)

Назначение паттерна Interpreter

- Для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка.
- Отображает проблемную область в язык, язык – в грамматику, а грамматику – в иерархии объектно-ориентированного проектирования.

Решаемая проблема

Пусть в некоторой, хорошо определенной области периодически случается некоторая проблема. Если эта область может быть описана некоторым “языком“, то проблема может быть легко решена с помощью “интерпретирующей машины“.

Обсуждение паттерна Interpreter

Паттерн Interpreter определяет грамматику простого языка для проблемной области, представляет грамматические правила в виде языковых предложений и интерпретирует их для решения задачи. Для представления каждого грамматического правила паттерн Interpreter использует отдельный класс. А так как грамматика, как правило, имеет иерархическую структуру, то иерархия наследования классов хорошо подходит для ее описания.

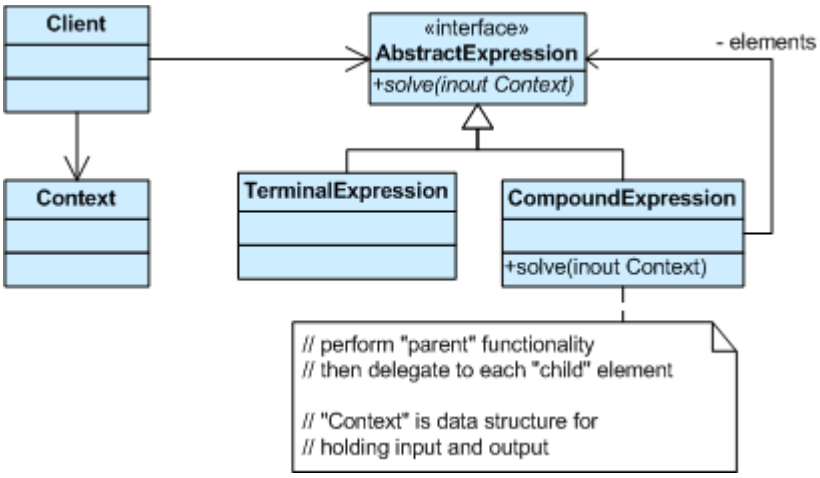
Абстрактный базовый класс определяет метод `interpret()`, принимающий (в качестве аргумента) текущее состояние языкового потока. Каждый конкретный подкласс реализует метод `interpret()`, добавляя свой вклад в процесс решения проблемы.

Структура паттерна Interpreter

Паттерн Interpreter моделирует проблемную область с помощью рекурсивной грамматики. Каждое грамматическое правило может быть либо составным (правило ссылается на другие правила) либо терминальным (листовой узел в структуре ”дерево”).

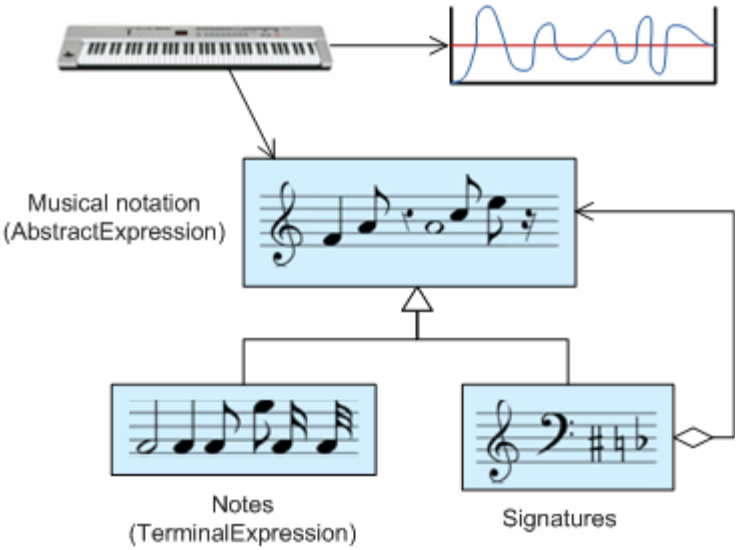
Для рекурсивного обхода ”предложений” при их интерпретации используется паттерн Composite.

UML-диаграмма классов паттерна Interpreter



Пример паттерна Interpreter

Паттерн Intepreter определяет грамматическое представление для языка и интерпретатор для интерпретации грамматики. Музыканты являются примерами интерпретаторов. Тональность и продолжительность звуков могут быть описаны нотами. Такое представление является музыкальным языком. Музыканты, используя ноты, способны воспроизвести оригинальные частоту и длительность каждого представленного звука.



Использование паттерна Interpreter

1. Определите “малый“ язык, “инвестиции” в который будут оправданными.
2. Разработайте грамматику для языка.
3. Для каждого грамматического правила (продукции) создайте свой класс.
4. Полученный набор классов организуйте в структуру с помощью паттерна Composite.
5. В полученной иерархии классов определите метод `interpret(Context)`.
6. Объект `Context` инкапсулирует информацию, глобальную по отношению к интерпретатору. Используется классами во время процесса ”интерпретации”.

Особенности паттерна Interpreter

- Абстрактное синтаксическое дерево интерпретатора – пример паттерна Composite.
- Для обхода узлов дерева может применяться паттерн Iterator.
- Терминальные символы могут разделяться с помощью Flyweight.
- Паттерн Interpreter не рассматривает вопросы синтаксического разбора. Когда грамматика очень сложная, должны использоваться другие методики.

Реализация паттерна Interpreter

Совместное использование паттернов Interpreter и Template Method

Рассмотрим задачу интерпретирования (вычисления) значений строковых представлений римских чисел. Используем следующую грамматику.

```
 romanNumeral ::= {thousands} {hundreds} {tens} {ones}
thousands,hundreds,tens,ones ::= nine | four | {five} {one} {one}
{one}
nine ::= "CM" | "XC" | "IX"
four ::= "CD" | "XL" | "IV"
five ::= 'D' | 'L' | 'V'
one  ::= 'M' | 'C' | 'X' | 'I'
```

Для проверки и интерпретации строки используется иерархия классов с общим базовым классом RNInterpreter, имеющим 4 под-интерпретатора. Каждый под-интерпретатор получает "контекст" (оставшуюся неразобранную часть строки и накопленное вычисленное значение разобранной части) и вносит свой вклад в процесс обработки. Под-переводчики просто определяют шаблонные методы, объявленные в базовом классе RNInterpreter.

```
#include <iostream.h>
#include <string.h>

class Thousand;
```

```
class Hundred;
class Ten;
class One;

class RNInterpreter
{
  public:
    RNInterpreter(); // ctor for client
    RNInterpreter(int){}
    // ctor for subclasses, avoids infinite loop
    int interpret(char*); // interpret() for client
    virtual void interpret(char *input, int &total)
    {
        // for internal use
        int index;
        index = 0;
        if (!strncmp(input, nine(), 2))
        {
            total += 9 * multiplier();
            index += 2;
        }
        else if (!strncmp(input, four(), 2))
        {
            total += 4 * multiplier();
            index += 2;
        }
        else
        {
            if (input[0] == five())
            {
                total += 5 * multiplier();
                index = 1;
            }
            else
              index = 0;
            for (int end = index + 3; index < end; index++)
              if (input[index] == one())
                total += 1 * multiplier();
              else
                break;
        }
        strcpy(input, &(input[index]));
    } // remove leading chars processed
  protected:
    // cannot be pure virtual because client asks for instance
    virtual char one(){}
    virtual char *four(){}
    virtual char five(){}
```

```
    virtual char *nine(){}
    virtual int multiplier(){}
  private:
    RNInterpreter *thousands;
    RNInterpreter *hundreds;
    RNInterpreter *tens;
    RNInterpreter *ones;
};

class Thousand: public RNInterpreter
{
  public:
    // provide 1-arg ctor to avoid infinite loop in base class ctor
    Thousand(int): RNInterpreter(1){}
  protected:
    char one()
    {
        return 'M';
    }
    char *four()
    {
        return "";
    }
    char five()
    {
        return '\0';
    }
    char *nine()
    {
        return "";
    }
    int multiplier()
    {
        return 1000;
    }
};

class Hundred: public RNInterpreter
{
  public:
    Hundred(int): RNInterpreter(1){}
  protected:
    char one()
    {
        return 'C';
    }
    char *four()
    {
```

```
        return "CD";
    }
    char five()
    {
        return 'D';
    }
    char *nine()
    {
        return "CM";
    }
    int multiplier()
    {
        return 100;
    }
};

class Ten: public RNInterpreter
{
  public:
    Ten(int): RNInterpreter(1){}
  protected:
    char one()
    {
        return 'X';
    }
    char *four()
    {
        return "XL";
    }
    char five()
    {
        return 'L';
    }
    char *nine()
    {
        return "XC";
    }
    int multiplier()
    {
        return 10;
    }
};

class One: public RNInterpreter
{
  public:
    One(int): RNInterpreter(1){}
  protected:
```

```
    char one()
    {
        return 'I';
    }
    char *four()
    {
        return "IV";
    }
    char five()
    {
        return 'V';
    }
    char *nine()
    {
        return "IX";
    }
    int multiplier()
    {
        return 1;
    }
};

RNInterpreter::RNInterpreter()
{
  // use 1-arg ctor to avoid infinite loop
  thousands = new Thousand(1);
  hundreds = new Hundred(1);
  tens = new Ten(1);
  ones = new One(1);
}

int RNInterpreter::interpret(char *input)
{
  int total;
  total = 0;
  thousands->interpret(input, total);
  hundreds->interpret(input, total);
  tens->interpret(input, total);
  ones->interpret(input, total);
  if (strcmp(input, ""))
  // if input was invalid, return 0
    return 0;
  return total;
}

int main()
{
  RNInterpreter interpreter;
```

```
  char input[20];
  cout << "Enter Roman Numeral: ";
  while(cin >> input)
  {
    cout << "   interpretation is "
         << interpreter.interpret(input) << endl;
    cout << "Enter Roman Numeral: ";
  }
}
```

Вывод программы:

```
Enter Roman Numeral: MCMXCVI
   interpretation is 1996
Enter Roman Numeral: MMMCMXCIX
   interpretation is 3999
Enter Roman Numeral: MMMM
   interpretation is 0
Enter Roman Numeral: MDCLXVIIII
   interpretation is 0
Enter Roman Numeral: CXCX
   interpretation is 0
Enter Roman Numeral: MDCLXVI
   interpretation is 1666
Enter Roman Numeral: DCCCLXXXVIII
   interpretation is 888
```

Паттерн Iterator (итератор, cursor, курсор)

Назначение паттерна Iterator

- Предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления.
- Абстракция в стандартных библиотеках C++ и Java, позволяющая разделить классы коллекций и алгоритмов.
- Придает обходу коллекции "объектно-ориентированный статус".
- Полиморфный обход.

Решаемая проблема

Вам необходим механизм "абстрактного" обхода различных структур данных так, что могут определяться алгоритмы, способные взаимодействовать со структурами прозрачно.

Обсуждение паттерна Iterator

Составной объект, такой как список, должен предоставлять способ доступа к его элементам без раскрытия своей внутренней структуры. Более того, иногда нужно перебирать элементы списка различными способами, в зависимости от конкретной задачи. Но вы, вероятно, не хотите раздувать интерфейс списка операциями для различных обходов, даже если они необходимы. Кроме того, иногда нужно иметь несколько активных обходов одного списка одновременно. Было бы хорошо иметь единый интерфейс для обхода разных типов составных объектов (т.е. полиморфная итерация).

Паттерн Iterator позволяет все это делать. Ключевая идея состоит в том, чтобы ответственность за доступ и обход переместить из составного объекта на объект Iterator, который будет определять стандартный протокол обхода.

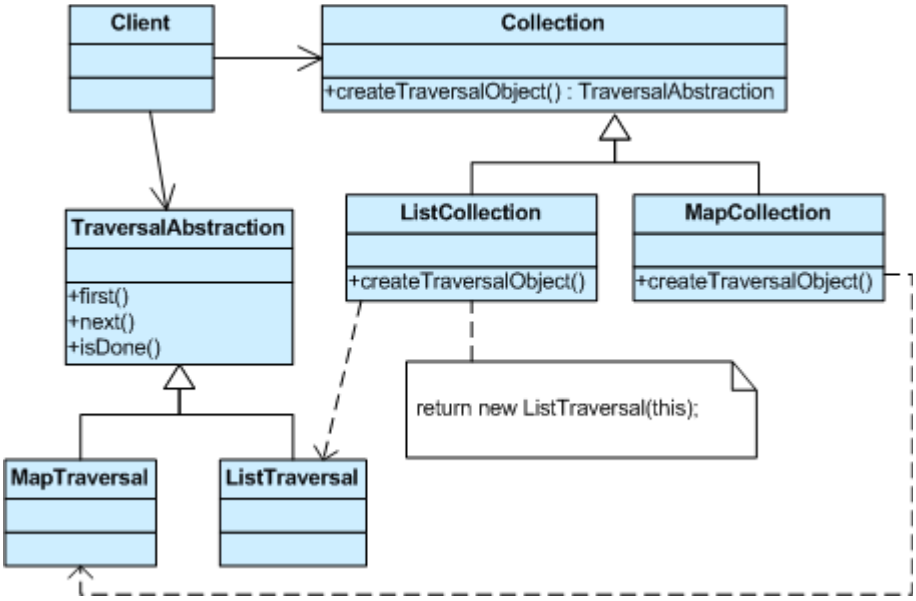
Абстракция Iterator имеет основополагающее значение для технологии, называемой "обобщенное программирование". Эта технология четко разделяет такие понятия как "алгоритм" и "структура данных". Мотивирующие факторы: способствование компонентной разработке, повышение производительности и снижение расходов на управление.

Рассмотрим пример. Если вы хотите одновременно поддерживать четыре вида структур данных (массив, бинарное дерево, связанный список и хэш-таблица) и три алгоритма (сортировка, поиск и слияние), то традиционный подход потребует 12 вариантов конфигураций (четыре раза по три), в то время как обобщенное программирование требует лишь 7 (четыре плюс три).

Структура паттерна Iterator

Для манипулирования коллекцией клиент использует открытый интерфейс класса `Collection`. Однако доступ к элементам коллекции инкапсулируется дополнительным уровнем абстракции, называемым `Iterator`. Каждый производный от `Collection` класс знает, какой производный от `Iterator` класс нужно создавать и возвращать. После этого клиент использует интерфейс, определенный в базовом классе `Iterator`.

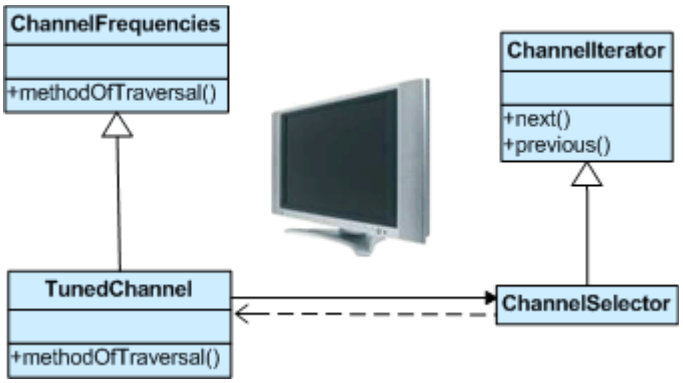
UML-диаграмма классов паттерна Iterator



Пример паттерна Iterator

Iterator предоставляет последовательный способ доступа к элементам коллекции, не раскрывая внутренней структуры. Папки с документами являются коллекциями. В офисных условиях, когда доступ к документам осуществляется через администратора или секретаря, именно секретарь выступает в качестве Iterator. Было уже снято несколько ТВ-комедий вокруг темы учета документов секретарем. Для руководителя эта система регистрации кажется запутанной и нелогичной, однако она работает – его секретарь может быстро и эффективно найти нужный документ.

У телевизоров прошлого поколения для смены ТВ-каналов использовался вращающийся переключатель. Каждой позиции на переключателе назначался свой канал. У современных телевизоров для смены каналов можно использовать кнопки "Далее" и "Назад". Когда телезритель нажимает кнопку "Далее", будет отображаться следующий настроенный канал. Телезритель всегда может запросить следующий канал, не зная его номера.



Использование паттерна Iterator

- Добавьте классу "коллекции" `create_iterator()` метод и назначьте привилегированный доступ классу "итератор".
- Спроектируйте класс "итератор", инкапсулирующий обход "коллекции" класса.
- Клиенты запрашивают у объекта `Collection` создание объекта-итератора.
- Клиенты используют `the first()`, `is_done()`, `next()` и `current_item()` для доступа к элементам класса `Collection`.

Особенности паттерна Iterator

- Iterator может применяться для обхода сложных структур, создаваемых Composite.
- Для создания экземпляра подкласса Iterator полиморфные итераторы используют Factory Method.
- Часто Memento и Iterator используются совместно. Iterator может использовать Memento для сохранения состояния итерации и содержит его внутри себя.

Реализация паттерна Iterator

Реализация паттерна Iterator с использованием методов

Вынесите из коллекции функциональность "обход элементов коллекции" и придайте ей в "объектный статус". Это упростит саму коллекцию, позволит одновременно создавать множество активных обходов и отделит алгоритмы от структур данных коллекции.

Каждый контейнерный класс должен иметь итератор. Может показаться, что это является нарушением принципа инкапсуляции, так как пользователи класса Stack получают доступ к его содержимому напрямую. Однако Джон Лакош (John Lakos) приводит следующие аргументы: дизайнер класса неизбежно что-то упустит. Позже, когда пользователям потребуется дополнительная функциональность, если итератор первоначально был предусмотрен, то они смогут добавить эту функциональность в соответствии с принципом "открыт для расширения, закрыт для модификации". Без наличия итератора их единственным выходом было бы докучливое изменение рабочего кода. Ниже исходный класс `Stack` не содержит оператор равенства, но имеет итератор. В результате, оператор равенства может быть легко добавлен.

- Спроектируйте класс "итератор" для класса "контейнер".
- Добавьте контейнерному классу `createIterator()` метод.
- Клиенты запрашивают у объекта "контейнер" создание объекта-итератора.
- Клиенты используют `first()`, `is_done()`, `next()` и `current_item()` методы.

```
#include <iostream>
using namespace std;

class Stack
{
    int items[10];
    int sp;
  public:
    friend class StackIter;
    Stack()
    {
        sp =  - 1;
    }
    void push(int in)
    {
        items[++sp] = in;
    }
    int pop()
    {
        return items[sp--];
    }
    bool isEmpty()
    {
        return (sp ==  - 1);
    }
    // 2. Добавьте член createIterator()
    StackIter *createIterator() const;
};

// 1. Спроектируйте класс "iterator"
class StackIter
{
    const Stack *stk;
    int index;
  public:
    StackIter(const Stack *s)
    {
        stk = s;
    }
    void first()
    {
        index = 0;
    }
    void next()
    {
        index++;
    }
    bool isDone()
```

```
    {
        return index == stk->sp + 1;
    }
    int currentItem()
    {
        return stk->items[index];
    }
};

StackIter *Stack::createIterator()const
{
  return new StackIter(this);
}

bool operator == (const Stack &l, const Stack &r)
{
  // 3. Клиенты запрашивают создание объекта StackIter у объекта
Stack
  StackIter *itl = l.createIterator();
  StackIter *itr = r.createIterator();
  // 4. Клиенты используют first(), isDone(), next(), and
currentItem()
  for ( itl->first(), itr->first();
       !itl->isDone();
        itl->next(), itr->next() )
    if (itl->currentItem() != itr->currentItem())
      break;
  bool ans = itl->isDone() && itr->isDone();
  delete itl;
  delete itr;
  return ans;
}

int main()
{
  Stack s1;
  for (int i = 1; i < 5; i++)
    s1.push(i);
  Stack s2(s1), s3(s1), s4(s1), s5(s1);
  s3.pop();
  s5.pop();
  s4.push(2);
  s5.push(9);
  cout << "1 == 2 is " << (s1 == s2) << endl;
  cout << "1 == 3 is " << (s1 == s3) << endl;
  cout << "1 == 4 is " << (s1 == s4) << endl;
  cout << "1 == 5 is " << (s1 == s5) << endl;
}
```

Вывод программы:

```
1 == 2 is 1
1 == 3 is 0
1 == 4 is 0
1 == 5 is 0
```

Реализация паттерна Iterator: использование операторов вместо методов

Джон Лакош отмечает, что интерфейсы GOF-итераторов дают возможность неправильного написания имен методов, являются неуклюжими и требуют слишком много печати. Представленная ниже реализация паттерна Iterator основана на использовании "интуитивных" операторов. Отметим также, что метод `createIterator()` больше не нужен. Пользователь создает итераторы как локальные переменные, поэтому очистка не требуется.

```
#include <iostream>
using namespace std;

class Stack
{
    int items[10];
    int sp;
  public:
    friend class StackIter;
    Stack()
    {
        sp =  - 1;
    }
    void push(int in)
    {
        items[++sp] = in;
    }
    int pop()
    {
        return items[sp--];
    }
    bool isEmpty()
    {
        return (sp ==  - 1);
    }
};

class StackIter
{
    const Stack &stk;
    int index;
```

```
  public:
    StackIter(const Stack &s): stk(s)
    {
        index = 0;
    }
    void operator++()
    {
        index++;
    }
    bool operator()()
    {
        return index != stk.sp + 1;
    }
    int operator *()
    {
        return stk.items[index];
    }
};

bool operator == (const Stack &l, const Stack &r)
{
  StackIter itl(l), itr(r);
  for (; itl(); ++itl, ++itr)
    if (*itl !=  *itr)
      break;
  return !itl() && !itr();
}

int main()
{
  Stack s1;
  int i;
  for (i = 1; i < 5; i++)
    s1.push(i);
  Stack s2(s1), s3(s1), s4(s1), s5(s1);
  s3.pop();
  s5.pop();
  s4.push(2);
  s5.push(9);
  cout << "1 == 2 is " << (s1 == s2) << endl;
  cout << "1 == 3 is " << (s1 == s3) << endl;
  cout << "1 == 4 is " << (s1 == s4) << endl;
  cout << "1 == 5 is " << (s1 == s5) << endl;
}
```

Вывод программы:

```
1 == 2 is 1
1 == 3 is 0
```

```
31 == 4 is 0
41 == 5 is 0
```

Паттерн Mediator (посредник)

Назначение паттерна Mediator

- Паттерн Mediator определяет объект, инкапсулирующий взаимодействие множества объектов. Mediator делает систему слабо связанной, избавляя объекты от необходимости ссылаться друг на друга, что позволяет изменять взаимодействие между ними независимо.
- Паттерн Mediator вводит посредника для развязывания множества взаимодействующих объектов.
- Заменяет взаимодействие "все со всеми" взаимодействием "один со всеми".

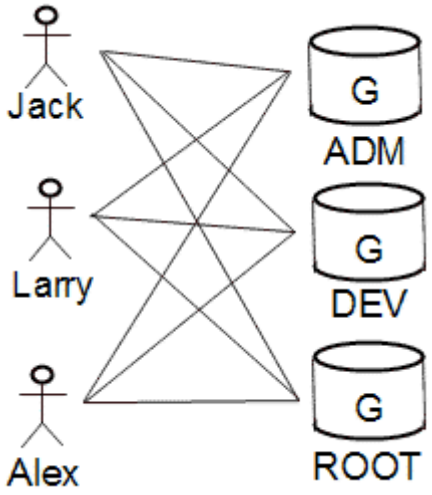
Решаемая проблема

Мы хотим спроектировать систему с повторно используемыми компонентами, однако существующие связи между этими компонентами можно охарактеризовать феноменом "спагетти-кода".

Спагетти-код - плохо спроектированная, слабо структурированная, запутанная и трудная для понимания программа. Спагетти-код назван так, потому что ход выполнения программы похож на миску спагетти, то есть извилистый и запутанный.

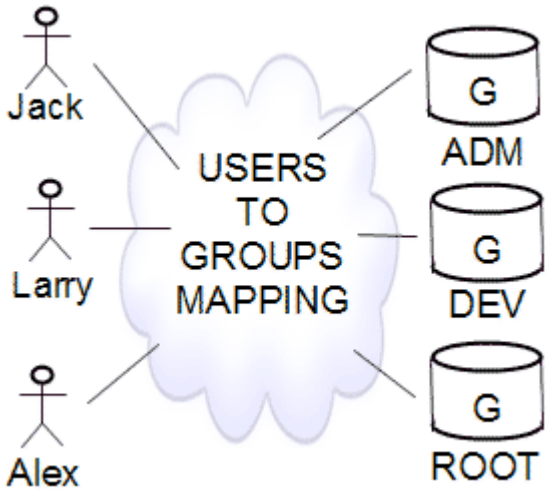
Обсуждение паттерна Mediator

В Unix права доступа к системным ресурсам определяются тремя уровнями: владелец, группа и прочие. Группа представляет собой совокупность пользователей, обладающих некоторой функциональной принадлежностью. Каждый пользователь в системе может быть членом одной или нескольких групп, и каждая группа может иметь 0 или более пользователей, назначенных этой группе. Следующий рисунок показывает трех пользователей, являющихся членами всех трех групп.



Если нам нужно было бы построить программную модель такой системы, то мы могли бы связать каждый объект User с каждым объектом Group, а каждый объект Group - с каждым объектом User. Однако из-за наличия множества взаимосвязей модифицировать поведение такой системы очень непросто, пришлось бы изменять все существующие классы.

Альтернативный подход - введение "дополнительного уровня косвенности" или построение абстракции из отображения (соответствия) пользователей в группы и групп в пользователей. Такой подход обладает следующими преимуществами: пользователи и группы отделены друг от друга, отображениями легко управлять одновременно и абстракция отображения может быть расширена в будущем путем определения производных классов.



Разбиение системы на множество объектов в общем случае повышает степень повторного использования, однако множество взаимосвязей между этими объектами, как правило, приводит к обратному эффекту. Чтобы этого не допустить, инкапсулируйте взаимодействия между объектами в объект-посредник. Действуя как центр связи, этот объект-посредник контролирует и координирует взаимодействие группы объектов. При этом объект-посредник делает взаимодействующие объекты слабо связанными, так как им больше не нужно хранить ссылки

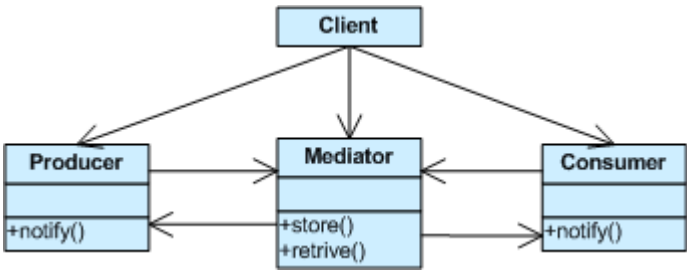
друг на друга – все взаимодействие идет через этого посредника. Расширить или изменить это взаимодействие можно через его подклассы.

Паттерн Mediator заменяет взаимодействие "все со всеми" взаимодействием "один со всеми".

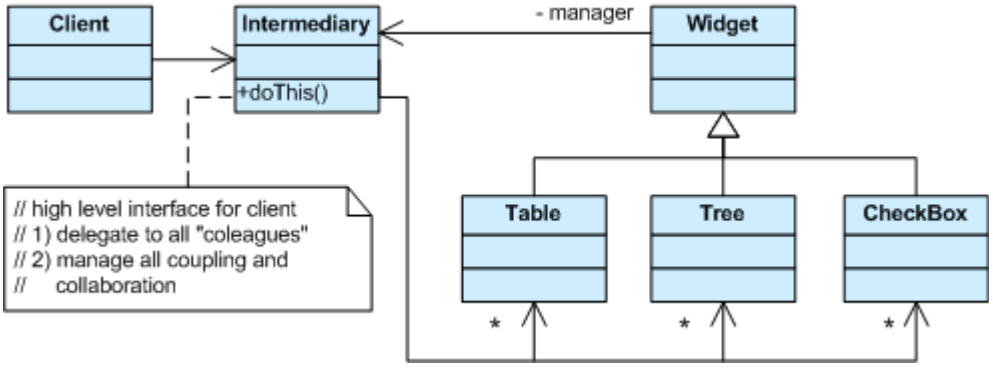
Пример рационального использования паттерна Mediator – моделирование отношений между пользователями и группами операционной системы. Группа может иметь 0 или более пользователей, а пользователь может быть членом 0 или более групп. Паттерн Mediator предусматривает гибкий способ управления пользователями и группами.

Структура паттерна Mediator

UML-диаграмма классов паттерна Mediator



Коллеги (или взаимодействующие объекты) не связаны друг с другом. Каждый из них общается с посредником, который, в свою очередь, знает об остальных и управляет ими. Паттерн Mediator делает статус взаимодействия "все со всеми" "полностью объектным".



Пример паттерна Mediator

Паттерн Mediator определяет объект, управляющий набором взаимодействующих объектов. Слабая связанность достигается благодаря тому, что вместо непосредственного взаимодействия друг с другом коллеги общаются через объект-посредник. Башня управления полетами в аэропорту хорошо демонстрирует этот паттерн. Пилоты взлетающих или идущих на посадку самолетов в районе аэропорта общаются с башней вместо непосредственного общения друг с другом. Башня определяет, кто и в каком порядке будет садиться или взлетать. Важно отметить, что башня контролирует самолеты только в районе аэродрома, а не на протяжении всего полета.

Использование паттерна Mediator

- Определите совокупность взаимодействующих объектов, связанность между которыми нужно уменьшить.
- Инкапсулируйте все взаимодействия в абстракцию нового класса.
- Создайте экземпляр этого нового класса. Объекты-коллеги для взаимодействия друг с другом используют только этот объект.
- Найдите правильный баланс между принципом слабой связанности и принципом распределения ответственности.
- Будьте внимательны и не создавайте объект-"контроллер" вместо объекта-посредника.

Особенности паттерна Mediator

- Паттерны Chain of Responsibility, Command, Mediator и Observer показывают, как можно разделить отправителей и получателей запросов с учетом их особенностей. Chain of Responsibility передает запрос отправителя по цепочке потенциальных получателей. Command номинально определяет связь - "оправитель-получатель" с помощью подкласса. В Mediator отправитель и получатель ссылаются друг на друга косвенно, через объект-посредник. В паттерне Observer связь между отправителем и получателем слабее, при этом число получателей может конфигурироваться во время выполнения.
- Mediator и Observer являются конкурирующими паттернами. Если Observer распределяет взаимодействие с помощью объектов "наблюдатель" и "субъект", то Mediator использует объект-посредник для инкапсуляции взаимодействия между другими объектами. Мы обнаружили, что легче сделать повторно используемыми Наблюдателей и Субъектов, чем Посредников.
- С другой стороны, Mediator может использовать Observer для динамической регистрации коллег и их взаимодействия с посредником.
- Mediator похож Facade в том, что он абстрагирует функциональность существующих классов. Mediator абстрагирует/централизует взаимодействие между объектами-

коллегами, добавляет новую функциональность и известен всем объектам-коллегам (то есть определяет двунаправленный протокол взаимодействия). Facade, наоборот, определяет более простой интерфейс к подсистеме, не добавляя новой функциональности, и неизвестен классам подсистемы (то есть имеет однонаправленный протокол взаимодействия, то есть запросы отправляются в подсистему, но не наоборот).

Реализация паттерна Mediator

Демонстрация паттерна Mediator

Степень повторного использования можно повысить через разбиение системы на множество объектов, однако при этом возникает множество взаимосвязей между этими объектами, которое приводит к обратному эффекту. Для исключения этой проблемы инкапсулируйте взаимодействие между объектами (коллективное поведение) в объект-посредник. Этот посредник будет управлять взаимодействием группы объектов.

В этом примере объект диалогового окна функционирует в качестве посредника. Виджеты диалогового окна ничего не знают о своих соседях. Всякий раз, когда происходит взаимодействие с пользователем виджет в `Widget::changed()` "делегирует" это событие посреднику `mediator->widgetChanged(this)`.

`FileSelectionDialog:: widgetChanged()` инкапсулирует все коллективное поведение для диалогового окна (служит центром взаимодействия). Пользователь может выбрать "взаимодействие" с полем редактирования Filter, списком Directories, списком Files или полем редактирования Selection.

```
#include <iostream.h>

class FileSelectionDialog;

class Widget
{
  public:
    Widget(FileSelectionDialog *mediator, char *name)
    {
        _mediator = mediator;
        strcpy(_name, name);
    }
    virtual void changed();
    virtual void updateWidget() = 0;
    virtual void queryWidget() = 0;
  protected:
    char _name[20];
```

```
  private:
    FileSelectionDialog *_mediator;
};

class List: public Widget
{
  public:
    List(FileSelectionDialog *dir, char *name): Widget(dir, name){}
    void queryWidget()
    {
        cout << "   " << _name << " list queried" << endl;
    }
    void updateWidget()
    {
        cout << "   " << _name << " list updated" << endl;
    }
};

class Edit: public Widget
{
  public:
    Edit(FileSelectionDialog *dir, char *name): Widget(dir, name){}
    void queryWidget()
    {
        cout << "   " << _name << " edit queried" << endl;
    }
    void updateWidget()
    {
        cout << "   " << _name << " edit updated" << endl;
    }
};

class FileSelectionDialog
{
  public:
    enum Widgets
    {
        FilterEdit, DirList, FileList, SelectionEdit
    };
    FileSelectionDialog()
    {
        _components[FilterEdit] = new Edit(this, "filter");
        _components[DirList] = new List(this, "dir");
        _components[FileList] = new List(this, "file");
        _components[SelectionEdit] = new Edit(this, "selection");
    }
    virtual ~FileSelectionDialog();
    void handleEvent(int which)
```

```
    {
        _components[which]->changed();
    }
    virtual void widgetChanged(Widget *theChangedWidget)
    {
        if (theChangedWidget == _components[FilterEdit])
        {
            _components[FilterEdit]->queryWidget();
            _components[DirList]->updateWidget();
            _components[FileList]->updateWidget();
            _components[SelectionEdit]->updateWidget();
        }
        else if (theChangedWidget == _components[DirList])
        {
            _components[DirList]->queryWidget();
            _components[FileList]->updateWidget();
            _components[FilterEdit]->updateWidget();
            _components[SelectionEdit]->updateWidget();
        }
        else if (theChangedWidget == _components[FileList])
        {
            _components[FileList]->queryWidget();
            _components[SelectionEdit]->updateWidget();
        }
        else if (theChangedWidget == _components[SelectionEdit])
        {
            _components[SelectionEdit]->queryWidget();
            cout << "   file opened" << endl;
        }
    }
  private:
    Widget *_components[4];
};

FileSelectionDialog::~FileSelectionDialog()
{
  for (int i = 0; i < 3; i++)
    delete _components[i];
}

void Widget::changed()
{
  _mediator->widgetChanged(this);
}

int main()
{
  FileSelectionDialog fileDialog;
```

```
  int i;

  cout << "Exit[0], Filter[1], Dir[2], File[3], Selection[4]: ";
  cin >> i;

  while (i)
  {
    fileDialog.handleEvent(i - 1);
    cout << "Exit[0], Filter[1], Dir[2], File[3], Selection[4]: ";
    cin >> i;
  }
}
```

Результаты вывода программы:

```
Exit[0], Filter[1], Dir[2], File[3], Selection[4]: 1
   filter edit queried
   dir list updated
   file list updated
   selection edit updated
Exit[0], Filter[1], Dir[2], File[3], Selection[4]: 2
   dir list queried
   file list updated
   filter edit updated
   selection edit updated
Exit[0], Filter[1], Dir[2], File[3], Selection[4]: 3
   file list queried
   selection edit updated
Exit[0], Filter[1], Dir[2], File[3], Selection[4]: 4
   selection edit queried
   file opened
Exit[0], Filter[1], Dir[2], File[3], Selection[4]: 3
   file list queried
   selection edit updated
```

Реализация паттерна Mediator: до и после

До

Объекты `Node` взаимодействуют друг с другом напрямую, требуется рекурсия, неудобное удаление узла, при этом первый узел удалить не возможно.

```
class Node
{
  public:
    Node(int v)
    {
        m_val = v;
        m_next = 0;
    }
    void add_node(Node *n)
```

```
    {
        if (m_next)
          m_next->add_node(n);
        else
          m_next = n;
    }
    void traverse()
    {
        cout << m_val << "  ";
        if (m_next)
          m_next->traverse();
        else
          cout << '\n';
    }
    void remove_node(int v)
    {
        if (m_next)
          if (m_next->m_val == v)
            m_next = m_next->m_next;
          else
            m_next->remove_node(v);
    }
  private:
    int m_val;
    Node *m_next;
};

int main()
{
  Node lst(11);
  Node two(22), thr(33), fou(44);
  lst.add_node(&two);
  lst.add_node(&thr);
  lst.add_node(&fou);
  lst.traverse();
  lst.remove_node(44);
  lst.traverse();
  lst.remove_node(22);
  lst.traverse();
  lst.remove_node(11);
  lst.traverse();
}
```

Результаты вывода программы:

```
11  22  33  44
11  22  33
11  33
11  33
```

После

"Посреднический" класс `List` упрощает все административные функции, рекурсия исключена.

```
class Node
{
  public:
    Node(int v)
    {
        m_val = v;
    }
    int get_val()
    {
        return m_val;
    }
  private:
    int m_val;
};

class List
{
  public:
    void add_node(Node *n)
    {
        m_arr.push_back(n);
    }
    void traverse()
    {
        for (int i = 0; i < m_arr.size(); ++i)
          cout << m_arr[i]->get_val() << "  ";
        cout << '\n';
    }
    void remove_node(int v)
    {
        for (vector::iterator it = m_arr.begin();
             it != m_arr.end(); ++it)
        if ((*it)->get_val() == v)
        {
            m_arr.erase(it);
            break;
        }
    }
  private:
    vector m_arr;
};

int main()
{
  List lst;
```

```
  Node one(11), two(22);
  Node thr(33), fou(44);
  lst.add_node(&one);
  lst.add_node(&two);
  lst.add_node(&thr);
  lst.add_node(&fou);
  lst.traverse();
  lst.remove_node(44);
  lst.traverse();
  lst.remove_node(22);
  lst.traverse();
  lst.remove_node(11);
  lst.traverse();
}
```

Результаты вывода программы:

```
11  22  33  44
11  22  33
11  33
33
```

Паттерн Memento (хранитель)

Назначение паттерна Memento

- Не нарушая инкапсуляции, паттерн Memento получает и сохраняет за пределами объекта его внутреннее состояние так, чтобы позже можно было восстановить объект в таком же состоянии.
- Является средством для инкапсуляции "контрольных точек" программы.
- Паттерн Memento придает операциям "Отмена" (undo) или "Откат" (rollback) статус "полноценного объекта".

Решаемая проблема

Вам нужно восстановить объект обратно в прежнее состояние (те есть выполнить операции "Отмена" или "Откат").

Обсуждение паттерна Memento

Клиент запрашивает Memento (хранителя) у исходного объекта, когда ему необходимо сохранить состояние исходного объекта (установить контрольную точку). Исходный объект инициализирует Memento своим текущим состоянием. Клиент является "посыльным" за Memento, но только исходный объект может сохранять и извлекать информацию из Memento (Memento является "непрозрачным" для клиентов и других объектов). Если клиенту в

дальнейшем нужно "откатить" состояние исходного объекта, он передает Memento обратно в исходный объект для его восстановления.

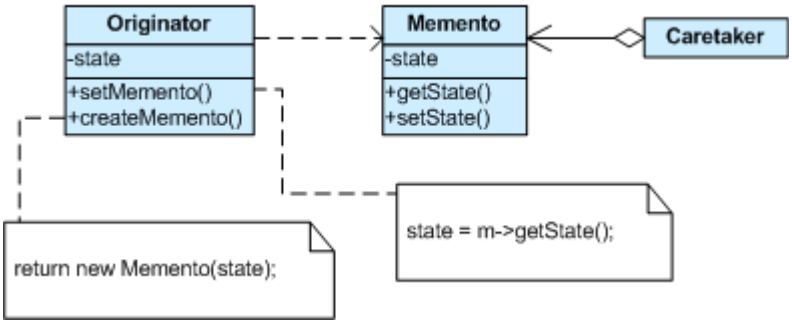
Реализовать возможность выполнения неограниченного числа операций "Отмена" (undo) и "Повтор" (redo) можно с помощью стека объектов Command и стека объектов Memento.

Паттерн проектирования Memento определяет трех различных участников:

- **Originator (хозяин)** - объект, умеющий создавать хранителя, а также знающий, как восстановить свое внутреннее состояние из хранителя.
- **Caretaker (смотритель)** - объект, который знает, почему и когда хозяин должен сохранять и восстанавливать себя.
- **Memento (хранитель)** - "ящик на замке", который пишется и читается хозяином и за которым присматривает смотритель.

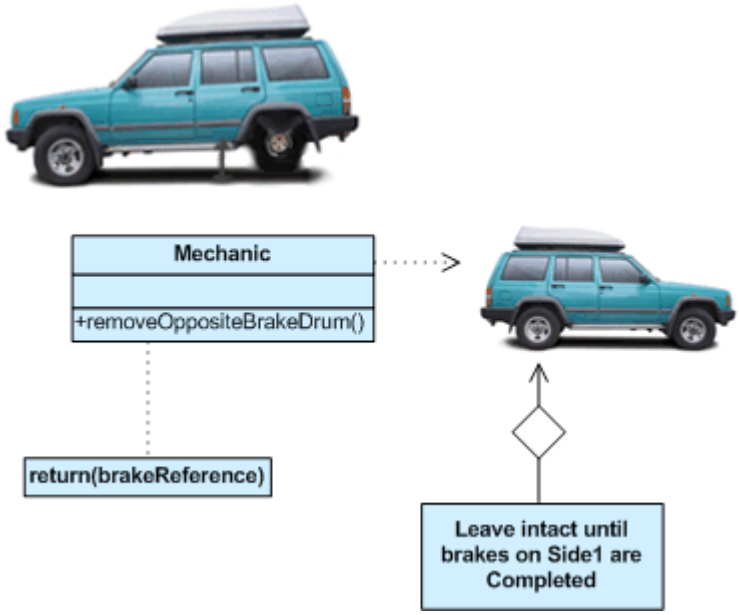
Структура паттерна Memento

UML-диаграмма классов паттерна Memento



Пример паттерна Memento

Паттерн Memento фиксирует и сохраняет за пределами объекта его внутреннее состояние так, чтобы позже этот объект можно было бы восстановить в таком же состоянии. Этот паттерн часто используется механиками-любителями для ремонта барабанных тормозов на своих автомобилях. Барабаны удаляются с обеих сторон, чтобы сделать видимыми правые и левые тормоза. При этом разбирается только одна сторона, другая же служит напоминанием (Memento) о том, как части тормозной системы тормозной системы собраны вместе. Только после того, как завершена работа с одной стороны, разбирается другая сторона. При этом в качестве Memento выступает уже первая сторона.



Использование паттерна Memento

- Определите роли "смотрителя" и "хозяина".
- Создайте класс Memento и объявите хозяина другом.
- Смотритель знает, когда создавать "контрольную точку" хозяина.
- Хозяин создает хранителя Memento и копирует свое состояние в этот Memento.
- Смотритель сохраняет хранителя Memento (но смотритель не может заглянуть в Memento).
- Смотритель знает, когда нужно "откатить" хозяина.
- Хозяин восстанавливает себя, используя сохраненное в Memento состояние.

Особенности паттерна Memento

- Паттерны Command и Memento определяют объекты "волшебная палочка", которые передаются от одного владельца к другому и используются позднее. В Command такой "волшебной палочкой" является запрос; в Memento - внутреннее состояние объекта в некоторый момент времени. Полиморфизм важен для Command, но не важен для Memento потому, что интерфейс Memento настолько "узкий", что его можно передавать как значение.
- Command может использовать Memento для сохранения состояния, необходимого для выполнения отмены действий.
- Memento часто используется совместно с Iterator. Iterator может использовать Memento для сохранения состояния итерации.

Реализация паттерна Memento

Memento - это объект, хранящий "снимок" внутреннего состояния другого объекта. Memento может использоваться для поддержки "многоуровневой" отмены действий паттерна Command. В этом примере перед выполнением команды по изменению объекта `Number`, текущее состояние этого объекта сохраняется в статическом списке истории хранителей `Memento`, а сама команда сохраняется в статическом списке истории команд. `Undo()` просто восстанавливает состояние объекта `Number`, получаемое из списка истории хранителей. `Redo()` использует список истории команд. Обратите внимание, `Memento` "открыт" для `Number`.

```
#include <iostream.h>
class Number;

class Memento
{
  public:
    Memento(int val)
    {
        _state = val;
    }
  private:
    friend class Number;
    int _state;
};

class Number
{
  public:
    Number(int value)
    {
        _value = value;
    }
    void dubble()
    {
        _value = 2 * _value;
    }
    void half()
    {
        _value = _value / 2;
    }
    int getValue()
    {
        return _value;
    }
```

```
    Memento *createMemento()
    {
        return new Memento(_value);
    }
    void reinstateMemento(Memento *mem)
    {
        _value = mem->_state;
    }
  private:
    int _value;
};

class Command
{
  public:
    typedef void(Number:: *Action)();
    Command(Number *receiver, Action action)
    {
        _receiver = receiver;
        _action = action;
    }
    virtual void execute()
    {
        _mementoList[_numCommands] = _receiver->createMemento();
        _commandList[_numCommands] = this;
        if (_numCommands > _highWater)
          _highWater = _numCommands;
        _numCommands++;
        (_receiver-> *_action)();
    }
    static void undo()
    {
        if (_numCommands == 0)
        {
            cout << "*** Attempt to run off the end!! ***" << endl;
            return ;
        }
           _commandList[_numCommands - 1]->_receiver-
>reinstateMemento
          (_mementoList[_numCommands - 1]);
        _numCommands--;
    }
    void static redo()
    {
        if (_numCommands > _highWater)
        {
            cout << "*** Attempt to run off the end!! ***" << endl;
            return ;
```

```
        }
  (_commandList[_numCommands]->_receiver-
>*(_commandList[_numCommands]
          ->_action))();
        _numCommands++;
    }
  protected:
    Number *_receiver;
    Action _action;
    static Command *_commandList[20];
    static Memento *_mementoList[20];
    static int _numCommands;
    static int _highWater;
};

Command *Command::_commandList[];
Memento *Command::_mementoList[];
int Command::_numCommands = 0;
int Command::_highWater = 0;

int main()
{
  int i;
  cout << "Integer: ";
  cin >> i;
  Number *object = new Number(i);

  Command *commands[3];
  commands[1] = new Command(object, &Number::dubble);
  commands[2] = new Command(object, &Number::half);

  cout << "Exit[0], Double[1], Half[2], Undo[3], Redo[4]: ";
  cin >> i;

  while (i)
  {
    if (i == 3)
      Command::undo();
    else if (i == 4)
      Command::redo();
    else
      commands[i]->execute();
    cout << "   " << object->getValue() << endl;
    cout << "Exit[0], Double[1], Half[2], Undo[3], Redo[4]: ";
    cin >> i;
  }
}
```

Вывод программы:

```
Integer: 11
Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 2
   5
Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 1
   10
Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 2
   5
Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 3
   10
Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 3
   5
Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 3
   11
Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 3
*** Attempt to run off the end!! ***
   11
Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 4
   5
Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 4
   10
Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 4
   5
Exit[0], Double[1], Half[2], Undo[3], Redo[4]: 4
*** Attempt to run off the end!! ***
   5
```

Паттерн Object Pool (пул объектов)

Назначение паттерна Object Pool

Применение паттерна Object Pool может значительно повысить производительность системы; его использование наиболее эффективно в ситуациях, когда создание экземпляров некоторого класса требует больших затрат, объекты в системе создаются часто, но число создаваемых объектов в единицу времени ограничено.

Решаемая проблема

Пулы объектов (известны также как пулы ресурсов) используются для управления кэшированием объектов. Клиент, имеющий доступ к пулу объектов может избежать создания новых объектов, просто запрашивая в пуле уже созданный экземпляр. Пул объектов может быть растущим, когда при отсутствии свободных создаются новые объекты или с ограничением количества создаваемых объектов.

Желательно, чтобы все многократно используемые объекты, свободные в некоторый момент

времени, хранились в одном и том же пуле объектов. Тогда ими можно управлять на основе единой политики. Для этого класс Object Pool проектируется с помощью паттерна Singleton.

Обсуждение паттерна Object Pool

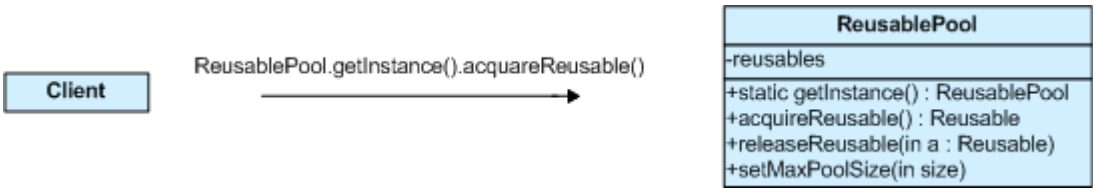
Процессы запрашивают объекты из пула объектов. Когда эти объекты больше не нужны, они возвращаются в пул для дальнейшего повторного использования.

Если при очередном запросе все объекты пула заняты, то процесс будет ожидать освобождения объекта. Для исключения подобной ситуации пул объектов должен уметь создавать новые объекты по мере необходимости. При этом он также должен реализовывать механизм периодической очистки неиспользуемых объектов.

Структура паттерна Object Pool

Основная идея паттерна Object Pool состоит в том, чтобы избежать создания новых экземпляров класса в случае возможности их повторного использования.

UML-диаграмма классов паттерна Object Pool



- **`Reusable`** - экземпляры классов в этой роли взаимодействуют с другими объектами в течение ограниченного времени, а затем они больше не нужны для этого взаимодействия.
- **`Client`** - экземпляры классов в этой роли используют объекты Reusable.
- **`ReusablePool`** - экземпляры классов в этой роли управляют объектами Reusable для использования объектами Client.

Как правило, желательно хранить все объекты Reusable в одном и том же пуле объектов. Это дает возможность управлять ими на основе единой политики. Для этого класс `ReusablePool` проектируется как Singleton. Его конструкторы объявляются как private, поэтому единственный экземпляр класса `ReusablePool` доступен другим классам только через метод `getInstance()`.

Клиент запрашивает объект `Reusable` через метод `acquireReusable()`объекта класса `ReusablePool`. Объект `ReusablePool` содержит коллекцию повторно используемых объектов `Reusable` для построения пула.

Если при вызове метода `acquireReusable()` в пуле имеются свободные объекты, то `acquireReusable()` удаляет объект `Reusable` из пула и возвращает его. Если же пул пустой, то метод `acquireReusable()` создает новый объект `Reusable`, если это

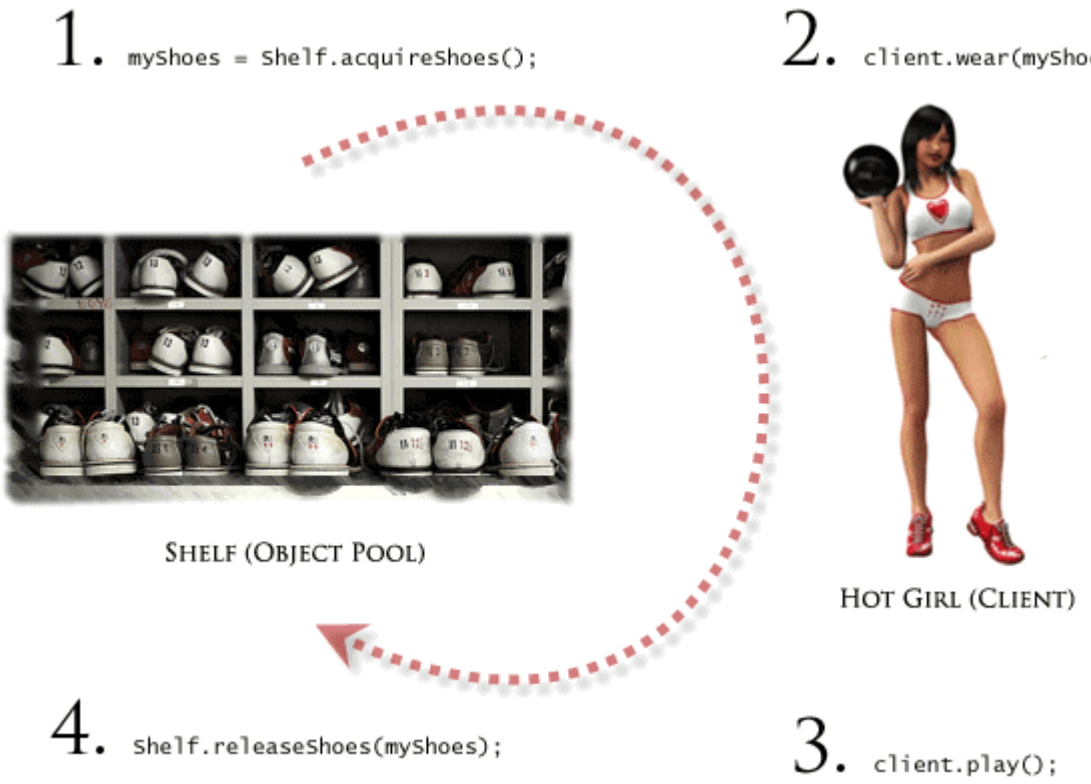
предусмотрено реализацией. Если же метод `acquireReusable()` не может создавать новые объекты, то он ждет, пока повторно используемый объект не возвратится в коллекцию.

После использования клиент передает объект `Reusable` в метод `releaseReusable()` объекта `ReusablePool`. Метод `releaseReusable()` возвращает этот объект в пул свободных для повторного использования объектов.

Во многих приложениях с применением паттерна Object Pool существуют причины для ограничения общего количества существующих объектов `Reusable`. В таких случаях объект `ReusablePool`, создающий объекты `Reusable`, ответственен за создание числа объектов `Reusable`, не превышающего указанного максимального. Если объект `ReusablePool` несет ответственность за ограничение количества создаваемого им объектов, то класс `ReusablePool` должен иметь метод для определения максимального количества создаваемых объектов. На рисунке выше этот метод обозначен как `setMaxPoolSize()`.

Пример паттерна Object Pool

Вам нравится боулинг? Если да, то вы, наверное, знаете, что должны сменить вашу обувь при посещении боулинг-клуба. Полка с обувью - прекрасный пример пула объектов. Когда вы хотите играть, вы берете с нее пару (`acquireReusable`). А после игры, вы возвращаете обувь обратно на полку (`releaseReusable`).



Использование паттерна Object Pool

- Создайте класс ObjectPool с private массивом объектов внутри.
- Создайте `acquire` и `release` методы в классе `ObjectPool.`
- Убедитесь, что ваш `ObjectPool` является одиночкой.

Особенности паттерна Object Pool

- Паттерн Factory Method может использоваться для инкапсуляции логики создания объектов. Однако он не управляет ими после их создания. Пул объектов отслеживает объекты, которые он создает.
- Пулы объектов, как правило, реализуются в виде одиночек.

Паттерн Observer (наблюдатель, издатель-подписчик)

Назначение паттерна Observer

- Паттерн Observer определяет зависимость "один-ко-многим" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически.
- Паттерн Observer инкапсулирует главный (независимый) компонент в абстракцию Subject и изменяемые (зависимые) компоненты в иерархию Observer.
- Паттерн Observer определяет часть "View" в модели Model-View-Controller (MVC) .

Паттерн Observer находит широкое применение в системах пользовательского интерфейса, в которых данные и их представления ("виды") отделены друг от друга. При изменении данных должны быть изменены все представления этих данных (например, в виде таблицы, графика и диаграммы).

Решаемая проблема

Имеется система, состоящая из множества взаимодействующих классов. При этом взаимодействующие объекты должны находиться в согласованных состояниях. Вы хотите избежать монолитности такой системы, сделав классы слабо связанными (или повторно используемыми).

Обсуждение паттерна Observer

Паттерн Observer определяет объект Subject, хранящий данные (модель), а всю

функциональность "представлений" делегирует слабосвязанным отдельным объектам Observer. При создании наблюдатели Observer регистрируются у объекта Subject. Когда объект Subject изменяется, он извещает об этом всех зарегистрированных наблюдателей. После этого каждый обозреватель запрашивает у объекта Subject ту часть состояния, которая необходима для отображения данных.

Такая схема позволяет динамически настраивать количество и "типы" представлений объектов.

Описанный выше протокол взаимодействия соответствует модели вытягивания (pull), когда субъект информирует наблюдателей о своем изменении, и каждый наблюдатель ответственен за "вытягивание" у Subject нужных ему данных. Существует также модель проталкивания, когда субъект Subject посылает ("проталкивает") наблюдателям детальную информацию о своем изменении.

Существует также ряд вопросов, о которых следует упомянуть, но обсуждение которых останется за рамками данной статьи:

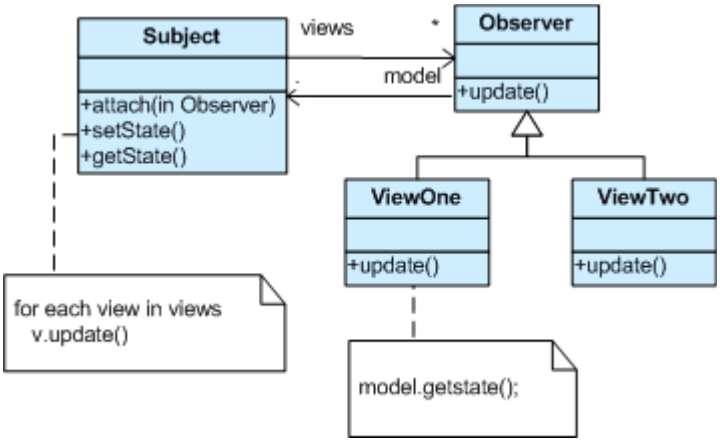
- Реализация "компрессии" извещений (посылка единственного извещения на серию последовательных изменений субъекта Subject).
- Мониторинг нескольких субъектов с помощью одного наблюдателя Observer.
- Исключение висячих ссылок у наблюдателей на удаленные субъекты. Для этого субъект должен уведомить наблюдателей о своем удалении.

Паттерн Observer впервые был применен в архитектуре Model-View-Controller языка Smalltalk, представляющей каркас для построения пользовательских интерфейсов.

Структура паттерна Observer

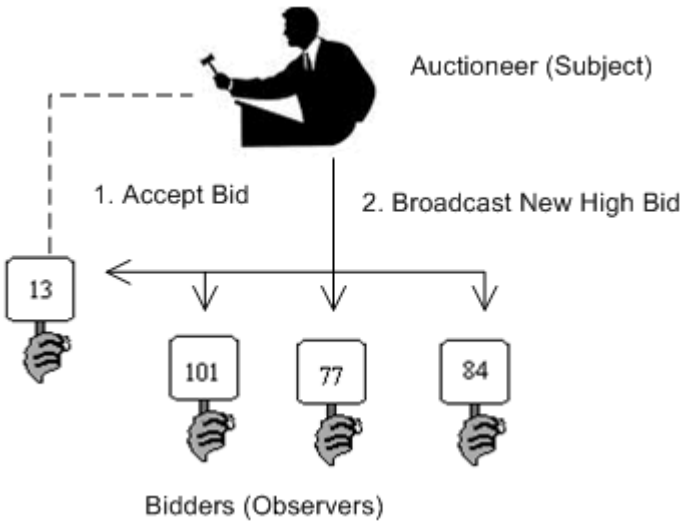
Subject представляет главную (независимую) абстракцию. Observer представляет изменяемую (зависимую) абстракцию. Субъект извещает наблюдателей о своем изменении, на что каждый наблюдатель может запросить состояние субъекта.

UML-диаграмма классов паттерна Observer



Пример паттерна Observer

Паттерн Observer определяет зависимость "один-ко-многим" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически. Некоторые аукционы демонстрируют этот паттерн. Каждый участник имеет карточку с цифрами, которую он использует для обозначения предлагаемой цены (ставки). Ведущий аукциона (Subject) начинает торги и наблюдает, когда кто-нибудь поднимает карточку, предлагая новую более высокую цену. Ведущий принимает заявку, о чем тут же извещает всех участников аукциона (Observers).



Использование паттерна Observer

1. Проведите различия между основной (или независимой) и дополнительной (или зависимой) функциональностями.
2. Смоделируйте "независимую" функциональность с помощью абстракции "субъект".
3. Смоделируйте "зависимую" функциональность с помощью иерархии "наблюдатель".
4. Класс `Subject` связан только с базовым классом `Observer`.
5. Клиент настраивает количество и типы наблюдателей.
6. Наблюдатели регистрируются у субъекта.
7. Субъект извещает всех зарегистрированных наблюдателей.
8. Субъект может "протолкнуть" информацию в наблюдателей, или наблюдатели могут "вытянуть" необходимую им информацию от объекта Subject.

Особенности паттерна Observer

- Паттерны Chain of Responsibility, Command, Mediator и Observer показывают, как можно разделить отправителей и получателей запросов с учетом своих особенностей. Chain of Responsibility передает запрос отправителя по цепочке потенциальных получателей. Command определяет связь - "оправитель-получатель" с помощью подкласса. В Mediator

отправитель и получатель ссылаются друг на друга косвенно, через объект-посредник. В паттерне Observer связь между отправителем и получателем получается слабой, при этом число получателей может конфигурироваться во время выполнения.

- Mediator и Observer являются конкурирующими паттернами. Если Observer распределяет взаимодействие с помощью объектов "наблюдатель" и "субъект", то Mediator использует объект-посредник для инкапсуляции взаимодействия между другими объектами. Мы обнаружили, что легче сделать повторно используемыми Наблюдателей и Субъектов, чем Посредников.
- Mediator может использовать Observer для динамической регистрации коллег и их взаимодействия с посредником.

Реализация паттерна Observer

Реализация паттерна Observer по шагам

1. Смоделируйте "независимую" функциональность с помощью абстракции "субъект".
2. Смоделируйте "зависимую" функциональность с помощью иерархии "наблюдатель".
3. Класс `Subject` связан только с базовым классом `Observer`.
4. Наблюдатели регистрируются у субъекта.
5. Субъект извещает всех зарегистрированных наблюдателей.
6. Наблюдатели "вытягивают" необходимую им информацию от объекта Subject.
7. Клиент настраивает количество и типы наблюдателей.

```
#include <iostream>
#include <vector>
using namespace std;

// 1. "Независимая" функциональность
class Subject {
    // 3. Связь только базовым классом Observer
    vector < class Observer * > views;
    int value;
  public:
    void attach(Observer *obs) {
        views.push_back(obs);
    }
    void setVal(int val) {
        value = val;
        notify();
    }
    int getVal() {
        return value;
    }
    void notify();
};

// 2. "Зависимая" функциональность
class Observer {
    Subject *model;
    int denom;
  public:
    Observer(Subject *mod, int div) {
        model = mod;
        denom = div;
        // 4. Наблюдатели регистрируются у субъекта
        model->attach(this);
    }
    virtual void update() = 0;
  protected:
    Subject *getSubject() {
        return model;
    }
    int getDivisor() {
        return denom;
    }
};

void Subject::notify() {
  // 5. Извещение наблюдателей
  for (int i = 0; i < views.size(); i++)
    views[i]->update();
}
```

Вывод программы:

```
14 div 4 is 3 14 div 3 is 4 14 mod 3 is 2
```

Реализация паттерна Observer: до и после

До

Количество и типы "зависимых" объектов определяются классом `Subject`. Пользователь не имеет возможности влиять на эту конфигурацию.

```
class DivObserver
{
    int m_div;
  public:
    DivObserver(int div)
    {
        m_div = div;
    }
    void update(int val)
    {
        cout << val << " div " << m_div
             << " is " << val / m_div << '\n';
    }
};

class ModObserver
{
    int m_mod;
  public:
    ModObserver(int mod)
    {
        m_mod = mod;
    }
    void update(int val)
    {
        cout << val << " mod " << m_mod
             << " is " << val % m_mod << '\n';
    }
};

class Subject
{
    int m_value;
    DivObserver m_div_obj;
    ModObserver m_mod_obj;
  public:
    Subject(): m_div_obj(4), m_mod_obj(3){}
    void set_value(int value)
```

```
    {
        m_value = value;
        notify();
    }
    void notify()
    {
        m_div_obj.update(m_value);
        m_mod_obj.update(m_value);
    }
};

int main()
{
  Subject subj;
  subj.set_value(14);
}
```

Вывод программы:

```
114 div 4 is 3 14 mod 3 is 2
```

После

Теперь класс Subject не связан с непосредственной настройкой числа и типов объектов `Observer`. Клиент установил два наблюдателя `DivObserver` и одного `ModObserver`.

```
class Observer
{
  public:
    virtual void update(int value) = 0;
};

class Subject
{
    int m_value;
    vector m_views;
  public:
    void attach(Observer *obs)
    {
        m_views.push_back(obs);
    }
    void set_val(int value)
    {
        m_value = value;
        notify();
    }
    void notify()
    {
        for (int i = 0; i < m_views.size(); ++i)
          m_views[i]->update(m_value);
    }
};

class DivObserver: public Observer
{
    int m_div;
  public:
    DivObserver(Subject *model, int div)
    {
        model->attach(this);
        m_div = div;
    }
     /* virtual */void update(int v)
    {
```

```
        cout << v << " div " << m_div << " is " << v / m_div << '\n';
    }
};

class ModObserver: public Observer
{
    int m_mod;
  public:
    ModObserver(Subject *model, int mod)
    {
        model->attach(this);
        m_mod = mod;
    }
     /* virtual */void update(int v)
    {
        cout << v << " mod " << m_mod << " is " << v % m_mod << '\n';
    }
};

int main()
{
  Subject subj;
  DivObserver divObs1(&subj, 4);
  DivObserver divObs2(&subj, 3);
  ModObserver modObs3(&subj, 3);
  subj.set_val(14);
}
```

Вывод программы:

```
14 div 4 is 3 14 div 3 is 4 14 mod 3 is 2
```

Паттерн Prototype (прототип)

Назначение паттерна Prototype

Паттерн Prototype (прототип) можно использовать в следующих случаях:

- Система должна оставаться независимой как от процесса создания новых объектов, так и от типов порождаемых объектов. Непосредственное использование выражения new в коде приложения считается нежелательным (подробнее об этом в разделе Порождающие паттерны).
- Необходимо создавать объекты, точные классы которых становятся известными уже на стадии выполнения программы.

Паттерн Factory Method также делает систему независимой от типов порождаемых объектов, но

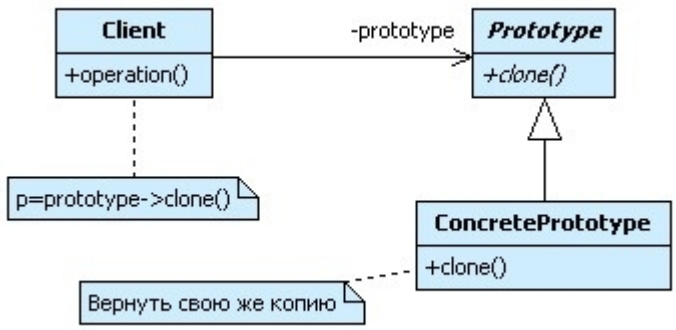
для этого он вводит параллельную иерархию классов: для каждого типа создаваемого объекта должен присутствовать соответствующий класс-фабрика, что может быть нежелательно. Паттерн Prototype лишен этого недостатка.

Описание паттерна Prototype

Для создания новых объектов паттерн Prototype использует прототипы. Прототип - это уже существующий в системе объект, который поддерживает операцию клонирования, то есть умеет создавать копию самого себя. Таким образом, для создания объекта некоторого класса достаточно выполнить операцию `clone()` соответствующего прототипа.

Паттерн Prototype реализует подобное поведение следующим образом: все классы, объекты которых нужно создавать, должны быть подклассами одного общего абстрактного базового класса. Этот базовый класс должен объявлять интерфейс метода `clone()`. Также здесь могут объявляться виртуальными и другие общие методы, например, `initialize()` в случае, если после клонирования нужна инициализация вновь созданного объекта. Все производные классы должны реализовывать метод `clone()`. В языке C++ для создания копий объектов используется конструктор копирования, однако, в общем случае, создание объектов при помощи операции копирования не является обязательным.

UML-диаграмма классов паттерна Prototype



Для порождения объекта некоторого типа в системе должен существовать его прототип. Прототип представляет собой объект того же типа, единственным назначением которого является создание подобных ему объектов. Обычно для удобства все существующие в системе прототипы организуются в специальные коллекции-хранилища или реестры прототипов. Такое хранилище может иметь реализацию в виде ассоциативного массива, каждый элемент которого представляет пару "Идентификатор типа" - "Прототип". Реестр прототипов позволяет добавлять или удалять прототип, а также создавать объект по идентификатору типа. Именно операции динамического добавления и удаления прототипов в хранилище обеспечивают дополнительную гибкость системе, позволяя управлять процессом создания новых объектов.

Реализация паттерна Prototype

Приведем реализацию паттерна Prototype на примере построения армий для военной стратегии "Пунические войны". Подробное описание этой игры можно найти в разделе Порождающие паттерны. Для упрощения демонстрационного кода будем создавать военные персонажи для некой абстрактной армии без учета особенностей воюющих сторон.

Также как и для паттерна Factory Method приведем две возможные реализации паттерна Prototype, а именно:

1. В виде обобщенного конструктора на основе прототипов, когда в полиморфном базовом классе Prototype определяется статический метод, предназначенный для создания объектов. При этом в качестве параметра в этот метод должен передаваться идентификатор типа создаваемого объекта.
2. На базе специально выделенного класса-фабрики.

Реализация паттерна Ptototype на основе обобщенного конструктора

```
#include <iostream>
#include <vector>
#include <map>

// Идентификаторы всех родов войск
enum Warrior_ID { Infantryman_ID, Archer_ID, Horseman_ID };

class Warrior;  // Опережающее объявление
typedef map<Warrior_ID, Warrior*> Registry;

// Реестр прототипов определен в виде Singleton Мэйерса
Registry& getRegistry()
{
  static Registry _instance;
  return _instance;
}

// Единственное назначение этого класса - помощь в выборе нужного
// конструктора при создании прототипов
class Dummy { };

// Полиморфный базовый класс. Здесь также определен статический
// обобщенный конструктор для создания боевых единиц всех родов войск
class Warrior
{
  public:
```

```
    virtual Warrior* clone() = 0;
    virtual void info() = 0;
    virtual ~Warrior() {}
    // Параметризированный статический метод для создания воинов
    // всех родов войск
    static Warrior* createWarrior( Warrior_ID id ) {
      Registry& r = getRegistry();
      if (r.find(id) != r.end())
        return r[id]->clone();
      return 0;
    }
  protected:
    // Добавление прототипа в множество прототипов
    static void addPrototype( Warrior_ID id, Warrior * prototype ) {
      Registry& r = getRegistry();
      r[id] = prototype;
    }
    // Удаление прототипа из множества прототипов
    static void removePrototype( Warrior_ID id ) {
      Registry& r = getRegistry();
      r.erase( r.find( id));
    }
};


// В производных классах различных родов войск в виде статических
// членов-данных определяются соответствующие прототипы
class Infantryman: public Warrior
{
  public:
    Warrior* clone() {
      return new Infantryman( *this);
    }
    void info() {
      cout << "Infantryman" << endl;
    }
  private:
    Infantryman( Dummy ) {
      Warrior::addPrototype( Infantryman_ID, this);
    }
    Infantryman() {}
    static Infantryman prototype;
};

class Archer: public Warrior
{
  public:
    Warrior* clone() {
```

```
      return new Archer( *this);
    }
    void info() {
      cout << "Archer" << endl;
    }
  private:
    Archer(Dummy) {
      addPrototype( Archer_ID, this);
    }
    Archer() {}
    static Archer prototype;
};

class Horseman: public Warrior
{
  public:
    Warrior* clone() {
      return new Horseman( *this);
    }
    void info() {
      cout << "Horseman" << endl;
    }
  private:
    Horseman(Dummy) {
      addPrototype( Horseman_ID, this);
    }
    Horseman() {}
    static Horseman prototype;
};


Infantryman Infantryman::prototype = Infantryman( Dummy());
Archer Archer::prototype = Archer( Dummy());
Horseman Horseman::prototype = Horseman( Dummy());


int main()
{
  vector<Warrior*> v;
  v.push_back( Warrior::createWarrior( Infantryman_ID));
  v.push_back( Warrior::createWarrior( Archer_ID));
  v.push_back( Warrior::createWarrior( Horseman_ID));

  for(int i=0; i<v.size(); i++)
    v[i]->info();
  // ...
}
```

В приведенной реализации классы всех создаваемых военных единиц, таких как лучники,

пехотинцы и конница, являются подклассами абстрактного базового класса Warrior. В этом классе определен обобщенный конструктор в виде статического метода createWarrior(Warrior_ID id). Передавая в этот метод в качестве параметра тип боевой единицы, можно создавать воинов нужных родов войск. Для этого обобщенный конструктор использует реестр прототипов, реализованный в виде ассоциативного массива std::map, каждый элемент которого представляет собой пару "идентификатор типа воина" - "его прототип".

Добавление прототипов в реестр происходит автоматически. Сделано это следующим образом. В подклассах Infantryman, Archer, Horseman, прототипы определяются в виде статических членов данных тех же типов. При создании такого прототипа будет вызываться конструктор с параметром типа Dummy, который и добавит этот прототип в реестр прототипов с помощью метода addPrototype() базового класса Warrior. Важно, чтобы к этому моменту сам объект реестра был полностью сконструирован, именно поэтому он выполнен в виде singleton Мэйерса.

Для приведенной реализации паттерна Prototype можно отметить следующие особенности:

- Создавать новых воинов можно только при помощи обобщенного конструктора. Их непосредственное создание невозможно, так как соответствующие конструкторы объявлены со спецификатором доступа private.
- Отсутствует недостаток реализации на базе обобщенного конструктора для паттерна Factory Method, а именно базовый класс Warrior ничего не знает о своих подклассах.

Реализация паттерна Prototype с помощью выделенного класса-фабрики

```
#include <iostream>
#include <vector>

// Иерархия классов игровых персонажей
// Полиморфный базовый класс
class Warrior
{
  public:
    virtual Warrior* clone() = 0;
    virtual void info() = 0;
    virtual ~Warrior() {}
};


// Производные классы различных родов войск
class Infantryman: public Warrior
{
    friend class PrototypeFactory;
  public:
    Warrior* clone() {
```

```
      return new Infantryman( *this);
    }
    void info() {
      cout << "Infantryman" << endl;
    }
  private:
    Infantryman() {}
};

class Archer: public Warrior
{
    friend class PrototypeFactory;
  public:
    Warrior* clone() {
      return new Archer( *this);
    }
    void info() {
      cout << "Archer" << endl;
    }
  private:
    Archer() {}
};

class Horseman: public Warrior
{
    friend class PrototypeFactory;
  public:
    Warrior* clone() {
      return new Horseman( *this);
    }
    void info() {
      cout << "Horseman" << endl;
    }
  private:
    Horseman() {}
};


// Фабрика для создания боевых единиц всех родов войск
class PrototypeFactory
{
  public:
    Warrior* createInfantrman() {
      static Infantryman prototype;
      return prototype.clone();
    }
    Warrior* createArcher() {
      static Archer prototype;
```

```
      return prototype.clone();
    }
    Warrior* createHorseman() {
      static Horseman prototype;
      return prototype.clone();
    }
};


int main()
{
  PrototypeFactory factory;
  vector<Warrior*> v;
  v.push_back( factory.createInfantrman());
  v.push_back( factory.createArcher());
  v.push_back( factory.createHorseman());

  for(int i=0; i<v.size(); i++)
    v[i]->info();
    // ...
}
```

В приведенной реализации для упрощения кода реестр прототипов не ведется. Воины всех родов войск создаются при помощи соответствующих методов фабричного класса PrototypeFactory, где и определены прототипы в виде статических переменных.

Результаты применения паттерна Prototype

Достоинства паттерна Prototype

- Для создания новых объектов клиенту необязательно знать их конкретные классы.
- Возможность гибкого управления процессом создания новых объектов за счет возможности динамических добавления и удаления прототипов в реестр.

Недостатки паттерна Prototype

- Каждый тип создаваемого продукта должен реализовывать операцию клонирования clone(). В случае, если требуется **глубокое копирование** объекта (объект содержит ссылки или указатели на другие объекты), это может быть непростой задачей.

Паттерн Proxy (заместитель, surrogate, суррогат)

Назначение паттерна Proxy

- Паттерн Proxy является суррогатом или заместителем другого объекта и контролирует доступ к нему.
- Предоставляя дополнительный уровень косвенности при доступе к объекту, может применяться для поддержки распределенного, управляемого или интеллектуального доступа.
- Являясь "оберткой" реального компонента, защищает его от излишней сложности.

Решаемая проблема

Вам нужно управлять ресурсоемкими объектами. Вы не хотите создавать экземпляры таких объектов до момента их реального использования.

Обсуждение паттерна Proxy

Суррогат или заместитель это объект, интерфейс которого идентичен интерфейсу реального объекта. При первом запросе клиента заместитель создает реальный объект, сохраняет его адрес и затем отправляет запрос этому реальному объекту. Все последующие запросы просто переадресуются инкапсулированному реальному объекту.

Существует четыре ситуации, когда можно использовать паттерн Proxy:

- Виртуальный proxy является заместителем объектов, создание которых обходится дорого. Реальный объект создается только при первом запросе/доступе клиента к объекту.
- Удаленный proxy предоставляет локального представителя для объекта, который находится в другом адресном пространстве ("заглушки" в RPC и CORBA).
- Защитный proxy контролирует доступ к основному объекту. "Суррогатный" объект предоставляет доступ к реальному объекту, только вызывающий объект имеет соответствующие права.
- Интеллектуальный proxy выполняет дополнительные действия при доступе к объекту.

Вот типичные области применения интеллектуальных proxy:

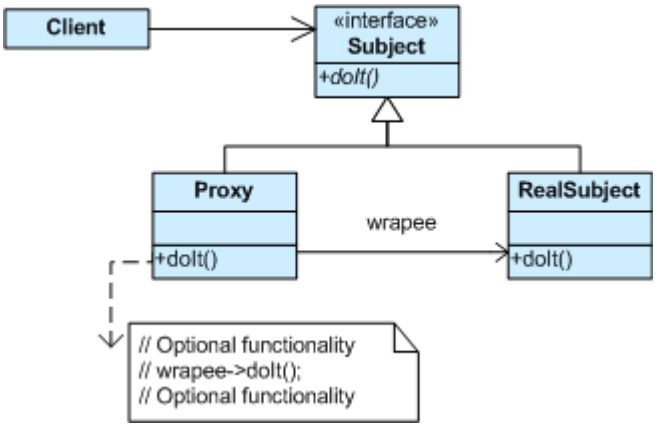
- Подсчет числа ссылок на реальный объект. При отсутствии ссылок память под объект автоматически освобождается (известен также как интеллектуальный указатель или smart pointer).
- Загрузка объекта в память при первом обращении к нему.
- Установка запрета на изменение реального объекта при обращении к нему других

объектов.

Структура паттерна Proxy

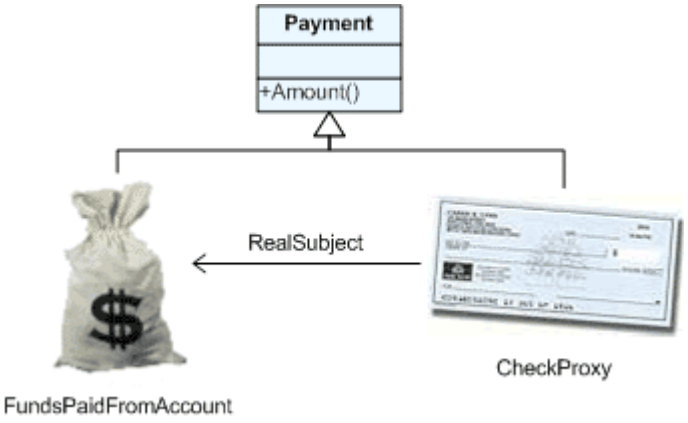
Заместитель Proxy и реальный объект RealSubject имеют одинаковые интерфейсы класса Subject, поэтому заместитель может использоваться "прозрачно" для клиента вместо реального объекта.

UML-диаграмма классов паттерна Proxy



Пример паттерна Proxy

Паттерн Proxy для доступа к реальному объекту использует его суррогат или заместитель. Банковский чек является заместителем денежных средств на счете. Чек может быть использован вместо наличных денег для совершения покупок и, в конечном счете, контролирует доступ к наличным деньгам на счете чекодателя.



Использование паттерна Proxy

- Определите ту часть системы, которая лучше всего реализуется через суррогата.
- Определите интерфейс, который сделает суррогата и оригинальный компонент взаимозаменяемыми.
- Рассмотрите вопрос об использовании фабрики, инкапсулирующей решение о том, что

желательно использовать на практике: оригинальный объект или его суррогат.

- Класс суррогата содержит указатель на реальный объект и реализует общий интерфейс.
- Указатель на реальный объект может инициализироваться в конструкторе или при первом использовании.
- Методы суррогата выполняют дополнительные действия и вызывают методы реального объекта.

Особенности паттерна Proxy

- Adapter предоставляет своему объекту другой интерфейс . Proxy предоставляет тот же интерфейс. Decorator предоставляет расширенный интерфейс.
- Decorator и Proxy имеют разные цели, но схожие структуры. Оба вводят дополнительный уровень косвенности: их реализации хранят ссылку на объект, на который они отправляют запросы.

Реализация паттерна Proxy

Паттерн Proxy: до и после

До

Прямые связи, накладные расходы на запуск и останов.

```
class Image
{
    int m_id;
    static int s_next;
  public:
    Image()
    {
        m_id = s_next++;
        cout << "   $$ ctor: " << m_id << '\n';
    }
    ~Image()
    {
        cout << "   dtor: " << m_id << '\n';
    }
    void draw()
    {
        cout << "   drawing image " << m_id << '\n';
    }
};
int Image::s_next = 1;

int main()
{
  Image images[5];

  for (int i; true;)
  {
    cout << "Exit[0], Image[1-5]: ";
    cin >> i;
    if (i == 0)
      break;
    images[i - 1].draw();
  }
}
```

Вывод программы:

```
   $$ ctor: 1
   $$ ctor: 2
   $$ ctor: 3
   $$ ctor: 4
   $$ ctor: 5
Exit[0], Image[1-5]: 2
   drawing image 2
Exit[0], Image[1-5]: 4
   drawing image 4
Exit[0], Image[1-5]: 2
   drawing image 2
Exit[0], Image[1-5]: 0
   dtor: 5
   dtor: 4
   dtor: 3
   dtor: 2
   dtor: 1
```

После

Инициализация при первом использовании.

- Спроектируйте класс-обертку с "дополнительным уровнем косвенности".
- Этот класс содержит указатель на реальный класс.
- Этот указатель инициализируется нулевым значением.
- Реальный объект создается при поступлении запроса "на первом использовании" (отложенная инициализация или lazy intialization).
- Запрос всегда делегируется реальному объекту.

```
class RealImage
{
    int m_id;
  public:
    RealImage(int i)
    {
        m_id = i;
        cout << "   $$ ctor: " << m_id << '\n';
    }
    ~RealImage()
    {
        cout << "   dtor: " << m_id << '\n';
    }
    void draw()
    {
        cout << "   drawing image " << m_id << '\n';
    }
};

// 1. Класс-обертка с "дополнительным уровнем косвенности"
```

```
class Image
{
    // 2. Класс-обертка содержит указатель на реальный класс
    RealImage *m_the_real_thing;
    int m_id;
    static int s_next;
  public:
    Image()
    {
        m_id = s_next++;
        // 3. Инициализируется нулевым значением
        m_the_real_thing = 0;
    }
    ~Image()
    {
        delete m_the_real_thing;
    }
    void draw()
    {
        // 4. Реальный объект создается при поступлении
        //    запроса "на первом использовании"
        if (!m_the_real_thing)
          m_the_real_thing = new RealImage(m_id);
        // 5. Запрос всегда делегируется реальному объекту
        m_the_real_thing->draw();
    }
};
int Image::s_next = 1;

int main()
{
  Image images[5];

  for (int i; true;)
  {
    cout << "Exit[0], Image[1-5]: ";
    cin >> i;
    if (i == 0)
      break;
    images[i - 1].draw();
  }
}
```

Вывод программы:

```
  Exit[0], Image[1-5]: 2
   $$ ctor: 2
   drawing image 2
Exit[0], Image[1-5]: 4
   $$ ctor: 4
   drawing image 4
Exit[0], Image[1-5]: 2
   drawing image 2
Exit[0], Image[1-5]: 0
   dtor: 4
   dtor: 2
```

Паттерн Proxy: защитный proxy контролирует доступ к основному объекту

```
class Person
{
    string nameString;
    static string list[];
    static int next;
  public:
    Person()
    {
        nameString = list[next++];
    }
    string name()
    {
        return nameString;
    }
};
string Person::list[] =
{
  "Tom", "Dick", "Harry", "Bubba"
};
int Person::next = 0;

class PettyCashProtected
{
    int balance;
  public:
    PettyCashProtected()
    {
        balance = 500;
    }
    bool withdraw(int amount)
    {
        if (amount > balance)
          return false;

```

```
        balance -= amount;
        return true;
    }
    int getBalance()
    {
        return balance;
    }
};

class PettyCash
{
    PettyCashProtected realThing;
  public:
    bool withdraw(Person &p, int amount)
    {
        if (p.name() == "Tom" || p.name() == "Harry"
                              || p.name() == "Bubba")
          return realThing.withdraw(amount);
        else
          return false;
    }
    int getBalance()
    {
        return realThing.getBalance();
    }
};

int main()
{
  PettyCash pc;
  Person workers[4];
  for (int i = 0, amount = 100; i < 4; i++, amount += 100)
    if (!pc.withdraw(workers[i], amount))
      cout << "No money for " << workers[i].name() << '\n';
    else
      cout << amount << " dollars for " << workers[i].name() <<
'\n';
  cout << "Remaining balance is " << pc.getBalance() << '\n';
}
```

Вывод программы:

```
100 dollars for Tom
No money for Dick
300 dollars for Harry
No money for Bubba
Remaining balance is 100
```

Паттерн Proxy: операторы "->" and "." дают различные результаты

```
class Subject
{
  public:
    virtual void execute() = 0;
};

class RealSubject: public Subject
{
    string str;
  public:
    RealSubject(string s)
    {
        str = s;
    }
     /*virtual*/void execute()
    {
        cout << str << '\n';
    }
};

class ProxySubject: public Subject
{
    string first, second, third;
    RealSubject *ptr;
  public:
    ProxySubject(string s)
    {
        int num = s.find_first_of(' ');
        first = s.substr(0, num);
        s = s.substr(num + 1);
        num = s.find_first_of(' ');
        second = s.substr(0, num);
        s = s.substr(num + 1);
        num = s.find_first_of(' ');
        third = s.substr(0, num);
        s = s.substr(num + 1);
        ptr = new RealSubject(s);
    }
    ~ProxySubject()
    {
        delete ptr;
    }
    RealSubject *operator->()
    {
```

```
        cout << first << ' ' << second << ' ';
        return ptr;
    }
     /*virtual*/ void execute()
    {
        cout << first << ' ' << third << ' ';
        ptr->execute();
    }
};

int main()
{
  ProxySubject obj(string("the quick brown fox jumped over the
dog"));
  obj->execute();
  obj.execute();
}
```

Вывод программы:

```
1 the quick fox jumped over the dog
2 the brown fox jumped over the dog
```

Паттерн Singleton (одиночка,синглет)

Назначение паттерна Singleton

Часто в системе могут существовать сущности только в единственном экземпляре, например, система ведения системного журнала сообщений или драйвер дисплея. В таких случаях необходимо уметь создавать единственный экземпляр некоторого типа, предоставлять к нему доступ извне и запрещать создание нескольких экземпляров того же типа.

Паттерн Singleton предоставляет такие возможности.

Описание паттерна Singleton

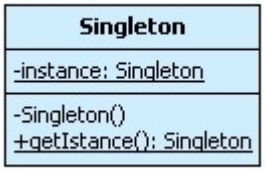
Архитектура паттерна Singleton основана на идее использования глобальной переменной, имеющей следующие важные свойства:

1. Такая переменная доступна всегда. Время жизни глобальной переменной - от запуска программы до ее завершения.
2. Предоставляет глобальный доступ, то есть, такая переменная может быть доступна из любой части программы.

Однако, использовать глобальную переменную некоторого типа непосредственно невозможно, так как существует проблема обеспечения единственности экземпляра, а именно, возможно создание нескольких переменных того же самого типа (например, стековых).

Для решения этой проблемы паттерн Singleton возлагает контроль над созданием единственного объекта на сам класс. Доступ к этому объекту осуществляется через статическую функцию-член класса, которая возвращает указатель или ссылку на него. Этот объект будет создан только при первом обращении к методу, а все последующие вызовы просто возвращают его адрес. Для обеспечения уникальности объекта, конструкторы и оператор присваивания объявляются закрытыми.

UML-диаграмма классов паттерна Singleton



Паттерн Singleton часто называют усовершенствованной глобальной переменной.

Реализация паттерна Singleton

Классическая реализация Singleton

Рассмотрим наиболее часто встречающуюся реализацию паттерна Singleton.

```
// Singleton.h
class Singleton
{
  private:
    static Singleton * p_instance;
    // Конструкторы и оператор присваивания недоступны клиентам
    Singleton() {}
    Singleton( const Singleton& );
    Singleton& operator=( Singleton& );
  public:
    static Singleton * getInstance() {
        if(!p_instance)
            p_instance = new Singleton();
        return p_instance;
    }
};

// Singleton.cpp
#include "Singleton.h"

Singleton* Singleton::p_instance = 0;
```

Клиенты запрашивают единственный объект класса через статическую функцию-член `getInstance()`, которая при первом запросе динамически выделяет память под этот объект и затем возвращает указатель на этот участок памяти. Впоследствии клиенты должны сами позаботиться об освобождении памяти при помощи оператора `delete`.

Последняя особенность является серьезным недостатком классической реализации шаблона Singleton. Так как класс сам контролирует создание единственного объекта, было бы логичным возложить на него ответственность и за разрушение объекта. Этот недостаток отсутствует в реализации Singleton, впервые предложенной Скоттом Мэйерсом.

Singleton Мэйерса

```
// Singleton.h
class Singleton
{
private:
    Singleton() {}
    Singleton( const Singleton&);
    Singleton& operator=( Singleton& );
public:
    static Singleton& getInstance() {
        static Singleton  instance;
        return instance;
    }
```

Внутри `getInstance()` используется статический экземпляр нужного класса. Стандарт языка программирования C++ гарантирует автоматическое уничтожение статических объектов при завершении программы. Досрочного уничтожения и не требуется, так как объекты `Singleton`

обычно являются долгоживущими объектами. Статическая функция-член `getInstance()` возвращает не указатель, а ссылку на этот объект, тем самым, затрудняя возможность ошибочного освобождения памяти клиентами.

Приведенная реализация паттерна Singleton использует так называемую отложенную инициализацию (lazy initialization) объекта, когда объект класса инициализируется не при старте программы, а при первом вызове `getInstance()`. В данном случае это обеспечивается тем, что статическая переменная `instance` объявлена внутри функции - члена класса `getInstance()`, а не как статический член данных этого класса. Отложенную инициализацию, в первую очередь, имеет смысл использовать в тех случаях, когда инициализация объекта представляет собой дорогостоящую операцию и не всегда используется.

К сожалению, у реализации Мэйерса есть недостатки: сложности создания объектов производных классов и невозможность безопасного доступа нескольких клиентов к единственному объекту в многопоточной среде.

Улучшенная версия классической реализации Singleton

С учетом всего вышесказанного классическая реализация паттерна Singleton может быть улучшена.

```
1  // Singleton.h
2  class Singleton;  // опережающее объявление
3
4  class SingletonDestroyer
5  {
6    private:
7      Singleton* p_instance;
8    public:
9      ~SingletonDestroyer();
10     void initialize( Singleton* p );
11 };
12
13 class Singleton
14 {
15   private:
16     static Singleton* p_instance;
17     static SingletonDestroyer destroyer;
18   protected:
19     Singleton() { }
20     Singleton( const Singleton& );
21     Singleton& operator=( Singleton& );
22    ~Singleton() { }
23     friend class SingletonDestroyer;
24   public:
25     static Singleton& getInstance();
26 };
```

```
27
28// Singleton.cpp
29#include "Singleton.h"
30
31Singleton * Singleton::p_instance = 0;
32SingletonDestroyer Singleton::destroyer;
33
34SingletonDestroyer::~SingletonDestroyer() {
35    delete p_instance;
36}
37void SingletonDestroyer::initialize( Singleton* p ) {
38    p_instance = p;
39}
40Singleton& Singleton::getInstance() {
41    if(!p_instance)     {
42        p_instance = new Singleton();
43        destroyer.initialize( p_instance);
44    }
45    return *p_instance;
46}
```

Ключевой особенностью этой реализации является наличие класса `SingletonDestroyer`, предназначенного для автоматического разрушения объекта Singleton. Класс `Singleton` имеет статический член `SingletonDestroyer`, который инициализируется при первом вызове `Singleton::getInstance()` создаваемым объектом `Singleton`. При завершении программы этот объект будет автоматически разрушен деструктором `SingletonDestroyer` (для этого `SingletonDestroyer` объявлен другом класса `Singleton`).

Для предотвращения случайного удаления пользователями объекта класса `Singleton`, деструктор теперь уже не является общедоступным как ранее. Он объявлен защищенным.

Использование нескольких взаимозависимых одиночек

До сих пор предполагалось, что в программе используется один одиночка либо несколько несвязанных между собой. При использовании взаимосвязанных одиночек появляются новые вопросы:

- Как гарантировать, что к моменту использования одного одиночки, экземпляр другого зависимого уже создан?
- Как обеспечить возможность безопасного использования одного одиночки другим при завершении программы? Другими словами, как гарантировать, что в момент разрушения первого одиночки в его деструкторе еще возможно использование второго зависимого одиночки (то есть второй одиночка к этому моменту еще не разрушен)?

Управлять порядком создания одиночек относительно просто. Следующий код демонстрирует один из возможных методов.

```
// Singleton.h
class Singleton1
{
  private:
    Singleton1() { }
    Singleton1( const Singleton1& );
    Singleton1& operator=( Singleton1& );
  public:
    static Singleton1& getInstance() {
        static Singleton1 instance;
        return instance;
    }
};

class Singleton2
{
  private:
    Singleton2( Singleton1& instance): s1( instance) { }
    Singleton2( const Singleton2& );
    Singleton2& operator=( Singleton2& );
    Singleton1& s1;
  public:
    static Singleton2& getInstance() {
        static Singleton2 instance( Singleton1::getInstance());
        return instance;
    }
};

// main.cpp
#include "Singleton.h"

int main()
{
    Singleton2& s = Singleton2::getInstance();
    return 0;
}
```

Объект `Singleton1` гарантированно инициализируется раньше объекта `Singleton2`, так как в момент создания объекта `Singleton2` происходит вызов `Singleton1::getInstance()`.

Гораздо сложнее управлять временем жизни одиночек. Существует несколько способов это сделать, каждый из них обладает своими достоинствами и недостатками и заслуживают отдельного рассмотрения. Обсуждение этой непростой темы остается за рамками проекта. Подробную информацию можно найти в [3].

Несмотря на кажущуюся простоту паттерна Singleton (используется всего один класс), его реализация не является тривиальной.

Результаты применения паттерна Singleton

Достоинства паттерна Singleton

- Класс сам контролирует процесс создания единственного экземпляра.
- Паттерн легко адаптировать для создания нужного числа экземпляров.
- Возможность создания объектов классов, производных от Singleton.

Недостатки паттерна Singleton

- В случае использования нескольких взаимозависимых одиночек их реализация может резко усложниться.

Паттерн State (состояние)

Назначение паттерна State

- Паттерн State позволяет объекту изменять свое поведение в зависимости от внутреннего состояния. Создается впечатление, что объект изменил свой класс.
- Паттерн State является объектно-ориентированной реализацией конечного автомата.

Решаемая проблема

Поведение объекта зависит от его состояния и должно изменяться во время выполнения программы. Такую схему можно реализовать, применив множество условных операторов: на основе анализа текущего состояния объекта предпринимаются определенные действия. Однако при большом числе состояний условные операторы будут разбросаны по всему коду, и такую программу будет трудно поддерживать.

Обсуждение паттерна State

Паттерн State решает указанную проблему следующим образом:

- Вводит класс `Context`, в котором определяется интерфейс для внешнего мира.
- Вводит абстрактный класс `State`.
- Представляет различные "состояния" конечного автомата в виде подклассов `State`.
- В классе `Context` имеется указатель на текущее состояние, который изменяется при изменении состояния конечного автомата.

Паттерн State не определяет, где именно определяется условие перехода в новое состояние. Существует два варианта: класс `Context` или подклассы `State`. Преимущество последнего варианта заключается в простоте добавления новых производных классов. Недостаток

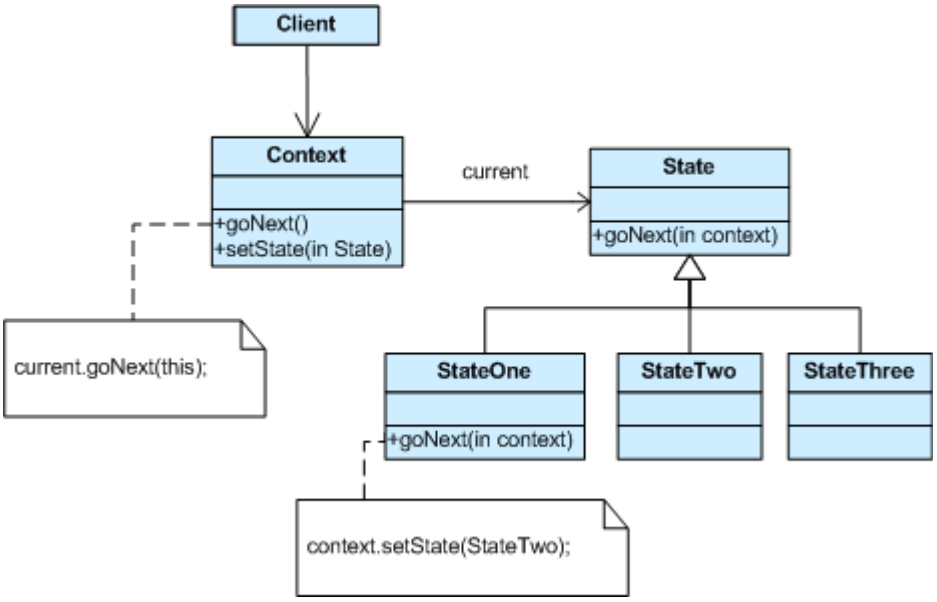
заключается в том, что каждый подкласс `State` для осуществления перехода в новое состояние должен знать о своих соседях, что вводит зависимости между подклассами.

Существует также альтернативный таблично-ориентированный подход к проектированию конечных автоматов, основанный на использовании таблицы однозначного отображения входных данных на переходы между состояниями. Однако этот подход обладает недостатками: трудно добавить выполнение действий при выполнении переходов. Подход, основанный на использовании паттерна State, для осуществления переходов между состояниями использует код (вместо структур данных), поэтому эти действия легко добавляемы.

Структура паттерна State

Класс `Context` определяет внешний интерфейс для клиентов и хранит внутри себя ссылку на текущее состояние объекта `State`. Интерфейс абстрактного базового класса `State` повторяет интерфейс `Context` за исключением одного дополнительного параметра - указателя на экземпляр `Context`. Производные от `State` классы определяют поведение, специфичное для конкретного состояния. Класс "обертка" `Context` делегирует все полученные запросы объекту "текущее состояние", который может использовать полученный дополнительный параметр для доступа к экземпляру `Context`.

UML-диаграмма классов паттерна State

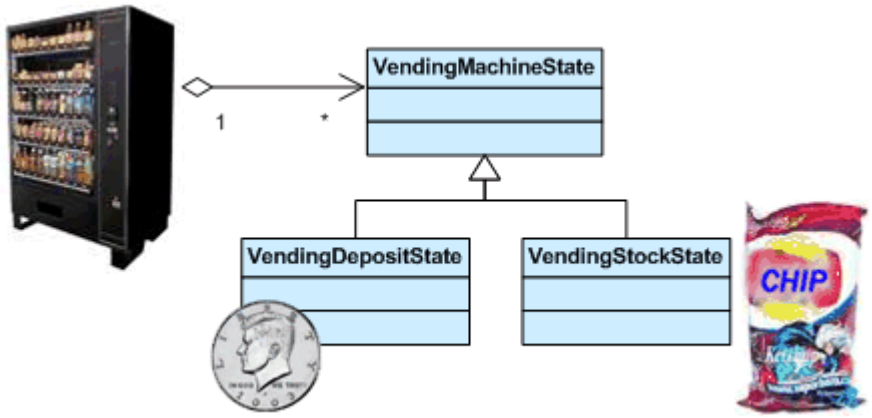


Пример паттерна State

Паттерн State позволяет объекту изменять свое поведение в зависимости от внутреннего состояния. Похожая картина может наблюдаться в работе торгового автомата. Автоматы могут иметь различные состояния в зависимости от наличия товаров, суммы полученных монет, возможности размена денег и т.д. После того как покупатель выбрал и оплатил товар, возможны

следующие ситуации (состояния):

- Выдать покупателю товар, выдавать сдачу не требуется.
- Выдать покупателю товар и сдачу.
- Покупатель товар не получит из-за отсутствия достаточной суммы денег.
- Покупатель товар не получит из-за его отсутствия.



Использование паттерна State

- Определите существующий или создайте новый класс-"обертку" `Context`, который будет использоваться клиентом в качестве "конечного автомата".
- Создайте базовый класс `State`, который повторяет интерфейс класса `Context`. Каждый метод принимает один дополнительный параметр: экземпляр класса `Context`. Класс `State` может определять любое полезное поведение "по умолчанию".
- Создайте производные от `State` классы для всех возможных состояний.
- Класс-"обертка" `Context` имеет ссылку на объект "текущее состояние".
- Все полученные от клиента запросы класс `Context` просто делегирует объекту "текущее состояние", при этом в качестве дополнительного параметра передается адрес объекта `Context`.
- Используя этот адрес, в случае необходимости методы класса `State` могут изменить "текущее состояние" класса `Context`.

Особенности паттерна State

- Объекты класса `State` часто бывают одиночками.
- Flyweight показывает, как и когда можно разделять объекты `State`.
- Паттерн Interpreter может использовать State для определения контекстов при синтаксическом разборе.
- Паттерны State и Bridge имеют схожие структуры за исключением того, что Bridge допускает иерархию классов-конвертов (аналогов классов-"оберток"), а State-нет. Эти

паттерны имеют схожие структуры, но решают разные задачи: State позволяет объекту изменять свое поведение в зависимости от внутреннего состояния, в то время как Bridge разделяет абстракцию от ее реализации так, что их можно изменять независимо друг от друга.

- Реализация паттерна State основана на паттерне Strategy. Различия заключаются в их назначении.

Реализация паттерна State

Рассмотрим пример конечного автомата с двумя возможными состояниями и двумя событиями.

```
#include <iostream>
using namespace std;
class Machine
{
  class State *current;
  public:
    Machine();
    void setCurrent(State *s)
    {
        current = s;
    }
    void on();
    void off();
};

class State
{
  public:
    virtual void on(Machine *m)
    {
        cout << "   already ON\n";
    }
    virtual void off(Machine *m)
    {
        cout << "   already OFF\n";
    }
};

void Machine::on()
{
  current->on(this);
}

void Machine::off()
{
  current->off(this);

```

```
}

class ON: public State
{
  public:
    ON()
    {
        cout << "   ON-ctor ";
    };
    ~ON()
    {
        cout << "   dtor-ON\n";
    };
    void off(Machine *m);
};

class OFF: public State
{
  public:
    OFF()
    {
        cout << "   OFF-ctor ";
    };
    ~OFF()
    {
        cout << "   dtor-OFF\n";
    };
    void on(Machine *m)
    {
        cout << "   going from OFF to ON";
        m->setCurrent(new ON());
        delete this;
    }
};

void ON::off(Machine *m)
{
  cout << "   going from ON to OFF";
  m->setCurrent(new OFF());
  delete this;
}

Machine::Machine()
{
  current = new OFF();
  cout << '\n';
}


```

```
  int main()
87{
88  void(Machine:: *ptrs[])() =
89  {
90    Machine::off, Machine::on
91  };
92  Machine fsm;
93  int num;
94  while (1)
95  {
96    cout << "Enter 0/1: ";
97    cin >> num;
98    (fsm. *ptrs[num])();
99  }
  }
```

Вывод программы:

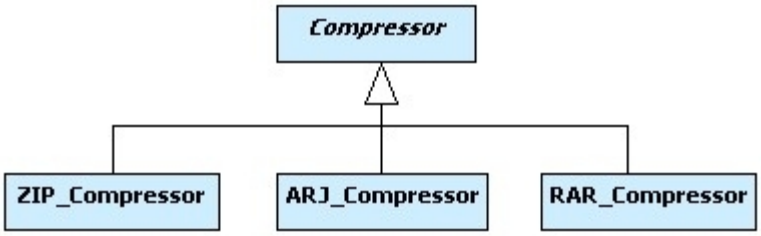
```
1    OFF-ctor
2 Enter 0/1: 0
3    already OFF
4 Enter 0/1: 1
5    going from OFF to ON   ON-ctor    dtor-OFF
6 Enter 0/1: 1
7    already ON
8 Enter 0/1: 0
9    going from ON to OFF   OFF-ctor    dtor-ON
10Enter 0/1: 1
11   going from OFF to ON   ON-ctor    dtor-OFF
12Enter 0/1: 0
13   going from ON to OFF   OFF-ctor    dtor-ON
14Enter 0/1: 0
15   already OFF
16Enter 0/1:
```

Паттерн Strategy (стратегия)

Назначение паттерна Strategy

Существуют системы, поведение которых может определяться согласно одному алгоритму из некоторого семейства. Все алгоритмы этого семейства являются родственными: предназначены для решения общих задач, имеют одинаковый интерфейс для использования и отличаются только реализацией (поведением). Пользователь, предварительно настроив программу на нужный алгоритм (выбрав стратегию), получает ожидаемый результат. Как пример, - приложение, предназначенное для компрессии файлов использует один из доступных алгоритмов: zip, arj или rar.

Объектно-ориентированный дизайн такой программы может быть построен на идее использования полиморфизма. В результате получаем набор родственных классов с общим интерфейсом и различными реализациями алгоритмов.



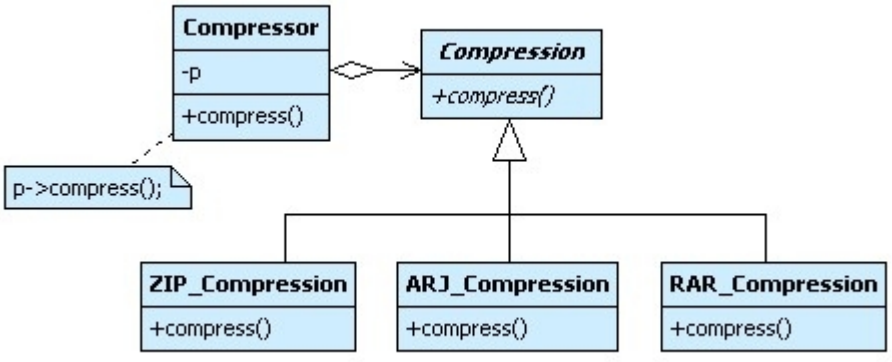
Представленному подходу свойственны следующие недостатки:

1. Реализация алгоритма жестко привязана к его подклассу, что затрудняет поддержку и расширение такой системы.
2. Система, построенная на основе наследования, является статичной. Заменить один алгоритм на другой в ходе выполнения программы уже невозможно.

Применение паттерна Strategy позволяет устранить указанные недостатки.

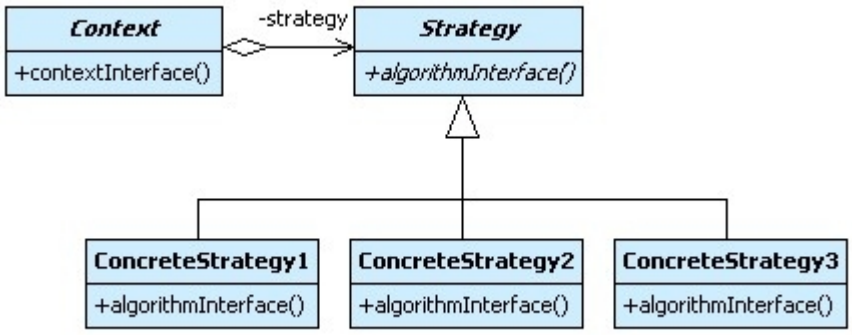
Описание паттерна Strategy

Паттерн Strategy переносит в отдельную иерархию классов все детали, связанные с реализацией алгоритмов. Для случая программы сжатия файлов абстрактный базовый класс Compression этой иерархии объявляет интерфейс, общий для всех алгоритмов и используемый классом Compressor. Подклассы ZIP_Compression, ARJ_Compression и RAR_Compression его реализуют в соответствии с тем или иным алгоритмом. Класс Compressor содержит указатель на объект абстрактного типа Compression и предназначен для переадресации пользовательских запросов конкретному алгоритму. Для замены одного алгоритма другим достаточно перенастроить этот указатель на объект нужного типа.



Структура паттерна Strategy

UML-диаграмма классов паттерна Strategy



Реализация паттерна Strategy

Приведем реализацию приложения для сжатия файлов, спроектированного с применением паттерна Strategy.

```
#include <iostream>
#include <string>

// Иерархия классов, определяющая алгоритмы сжатия файлов
class Compression
{
  public:
    virtual ~Compression() {}
    virtual void compress( const string & file ) = 0;
};

class ZIP_Compression : public Compression
{
  public:
    void compress( const string & file ) {
        cout << "ZIP compression" << endl;
    }
};

class ARJ_Compression : public Compression
{
  public:
    void compress( const string & file ) {
        cout << "ARJ compression" << endl;
    }
};

class RAR_Compression : public Compression
{
  public:
    void compress( const string & file ) {
        cout << "RAR compression" << endl;
    }
};


// Класс для использования
class Compressor
{
  public:
    Compressor( Compression* comp): p(comp) {}
   ~Compressor() { delete p; }
    void compress( const string & file ) {
      p->compress( file);
    }
  private:
    Compression* p;
};
```

Результаты применения паттерна Strategy

Достоинства паттерна Strategy

- Систему проще поддерживать и модифицировать, так как семейство алгоритмов перенесено в отдельную иерархию классов.
- Паттерн Strategy предоставляет возможность замены одного алгоритма другим в процессе выполнения программы.
- Паттерн Strategy позволяет скрыть детали реализации алгоритмов от клиента.

Недостатки паттерна Strategy

- Для правильной настройки системы пользователь должен знать об особенностях всех алгоритмов.
- Число классов в системе, построенной с применением паттерна Strategy, возрастает.

Паттерн Template Method (шаблонный метод)

Назначение паттерна Template Method

- Паттерн Template Method определяет основу алгоритма и позволяет подклассам изменить некоторые шаги этого алгоритма без изменения его общей структуры.
- Базовый класс определяет шаги алгоритма с помощью абстрактных операций, а производные классы их реализуют.

Решаемая проблема

Имеются два разных, но в тоже время очень похожих компонента. Вы хотите внести изменения в оба компонента, избежав дублирования кода.

Обсуждение паттерна Template Method

Проектировщик компонента решает, какие шаги алгоритма являются неизменными (или стандартными), а какие изменяемыми (или настраиваемыми). Абстрактный базовый класс реализует стандартные шаги алгоритма и может предоставлять (или нет) реализацию по умолчанию для настраиваемых шагов. Изменяемые шаги могут (или должны) предоставляться клиентом компонента в конкретных производных классах.

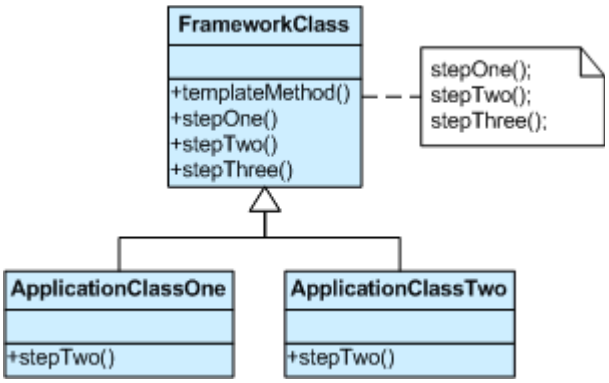
Проектировщик компонента определяет необходимые шаги алгоритма, порядок их выполнения, но позволяет клиентам компонента расширять или замещать некоторые из этих шагов.

Паттерн Template Method широко применяется в каркасах приложений (frameworks). Каждый

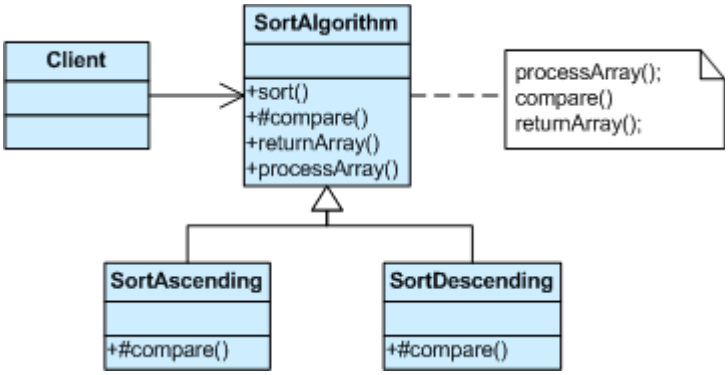
каркас реализует неизменные части архитектуры в предметной области, а также определяет те части, которые могут или должны настраиваться клиентом. Таким образом, каркас приложения становится "центром вселенной", а настройки клиента являются просто "третьей планетой от Солнца". Эту инвертированную структуру кода ласково называют принципом Голливуда - "Не звоните нам, мы сами вам позвоним".

Структура паттерна Template Method

UML-диаграмма классов паттерна Template Method

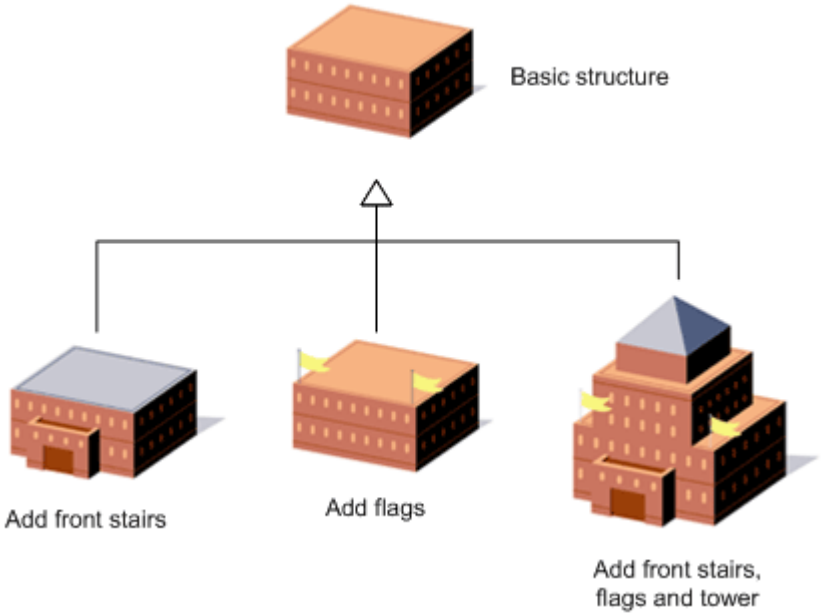


Реализация метода `templateMethod()` вызывает методы `stepOne()`, `stepTwo()` и `stepThree()`. Метод `stepTwo()` является "замещающим" методом. Он объявлен в базовом классе, а определяется в производных классах. Каркасы приложений широко используют паттерн Template Method. Весь повторно используемый код определяется в базовых классах каркаса, нужное поведение системы клиенты определяют в создаваемых производных классах.



Пример паттерна Template Method

Паттерн Template Method определяет основу алгоритма и позволяет подклассам изменить некоторые шаги этого алгоритма без изменения его общей структуры. Строители зданий используют шаблонный метод при проектировании новых домов. Здесь могут использоваться уже существующие типовые планы, в которых модифицируются только отдельные части.



Использование паттерна Template Method

1. Исследуйте алгоритм и решите, какие шаги являются стандартными, а какие должны определяться подклассами.
2. Создайте новый абстрактный базовый класс, в котором будет реализован принцип "не звоните нам, мы сами вам позвоним".
3. Поместите в новый класс основу алгоритма (шаблонный метод) и определения стандартных шагов.
4. Для каждого шага, требующего различные реализации, определите "замещающий" виртуальный метод. Этот метод может иметь реализацию по умолчанию или быть чисто виртуальным.
5. Вызовите "замещающий" метод из шаблонного метода.
6. Создайте подклассы от нового абстрактного базового класса и реализуйте в них "замещающие" методы.

Особенности паттерна Template Method

- Template Method использует наследование для модификации части алгоритма. Стратегия использует делегирование для модификации всего алгоритма.
- Стратегия изменяет логику отдельных объектов. Template Method изменяет логику всего класса.
- Фабричные методы часто вызываются из шаблонных методов.

Реализация паттерна Template Method

1. Стандартизуйте основу алгоритма в шаблонном методе базового класса.

2. Для шагов, требующих особенной реализации, определите "замещающие" методы.
3. Производные классы реализуют "замещающие" методы.

```
#include <iostream>
using namespace std;

class Base
{
    void a()
    {
        cout << "a  ";
    }
    void c()
    {
        cout << "c  ";
    }
    void e()
    {
        cout << "e  ";
    }
    // 2. Для шагов, требующих особенной реализации, определите
    //    "замещающие" методы.
    virtual void ph1() = 0;
    virtual void ph2() = 0;
  public:
    // 1. Стандартизуйте основу алгоритма в шаблонном методе
    //    базового класса
    void execute()
    {
        a();
        ph1();
        c();
        ph2();
        e();
    }
};

class One: public Base
{
   // 3. Производные классы реализуют "замещающие" методы.
     /*virtual*/void ph1()
    {
        cout << "b  ";
    }
     /*virtual*/void ph2()
    {
        cout << "d  ";
    }

```

```
};

class Two: public Base
{
     /*virtual*/void ph1()
    {
        cout << "2  ";
    }
     /*virtual*/void ph2()
    {
        cout << "4  ";
    }
};

int main()
{
  Base *array[] =
  {
     &One(), &Two()
  };
  for (int i = 0; i < 2; i++)
  {
    array[i]->execute();
    cout << '\n';
  }
}
```

Вывод программы:

```
1a b c d e a 2 c 4 e
```

Паттерн Visitor (посетитель)

Назначение паттерна Visitor

- Паттерн Visitor определяет операцию, выполняемую на каждом элементе из некоторой структуры. Позволяет, не изменяя классы этих объектов, добавлять в них новые операции.
- Является классической техникой для восстановления потерянной информации о типе.
- Паттерн Visitor позволяет выполнить нужные действия в зависимости от типов двух объектов.
- Предоставляет механизм двойной диспетчеризации.

Решаемая проблема

Различные и несвязанные операции должны выполняться над узловыми объектами некоторой гетерогенной совокупной структуры. Вы хотите избежать "загрязнения" классов этих узлов такими операциями (то есть избежать добавления соответствующих методов в эти классы). И вы не хотите запрашивать тип каждого узла и осуществлять приведение указателя к правильному типу, прежде чем выполнить нужную операцию.

Обсуждение паттерна Visitor

Основным назначением паттерна Visitor является введение абстрактной функциональности для совокупной иерархической структуры объектов "элемент", а именно, паттерн Visitor позволяет, не изменяя классы `Element`, добавлять в них новые операции. Для этого вся обрабатывающая функциональность переносится из самих классов `Element` (эти классы становятся "легковесными") в иерархию наследования `Visitor`.

При этом паттерн Visitor использует технику "двойной диспетчеризации". Обычно при передаче запросов используется "одинарная диспетчеризация" – то, какая операция будет выполнена для обработки запроса, зависит от имени запроса и типа получателя. В "двойной диспетчеризации" вызываемая операция зависит от имени запроса и типов двух получателей (типа `Visitor` и типа посещаемого элемента `Element`).

Реализуйте паттерн Visitor следующим образом. Создайте иерархию классов `Visitor`, в абстрактном базовом классе которой для каждого подкласса `Element` совокупной структуры определяется чисто виртуальный метод `visit()`. Каждый метод `visit()` принимает один аргумент - указатель или ссылку на подкласс `Element`.

Каждая новая добавляемая операция моделируется при помощи конкретного подкласса `Visitor`. Подклассы `Visitor` реализуют `visit()` методы, объявленные в базовом классе `Visitor`.

Добавьте один чисто виртуальный метод `accept()` в базовый класс иерархии `Element`. В качестве параметра `accept()` принимает единственный аргумент - указатель или ссылку на абстрактный базовый класс иерархии `Visitor`.

Каждый конкретный подкласс `Element` реализует метод `accept()` следующим образом: используя полученный в качестве параметра адрес экземпляра подкласса `Visitor`, просто вызывает его метод `visit()`, передавая в качестве единственного параметра указатель `this`.

Теперь "элементы" и "посетители" готовы. Если клиенту нужно выполнить какую-либо операцию, то он создает экземпляр объекта соответствующего подкласса `Visitor` и вызывает `accept()` метод для каждого объекта `Element`, передавая экземпляр `Visitor` в качестве параметра.

При вызове метода `accept()` ищется правильный подкласс `Element`. Затем, при вызове метода `visit()` программное управление передается правильному подклассу `Visitor`. Таким образом, двойная диспетчеризация получается как сумма одинарных диспетчеризаций сначала в методе `accept()`, а затем в методе `visit()`.

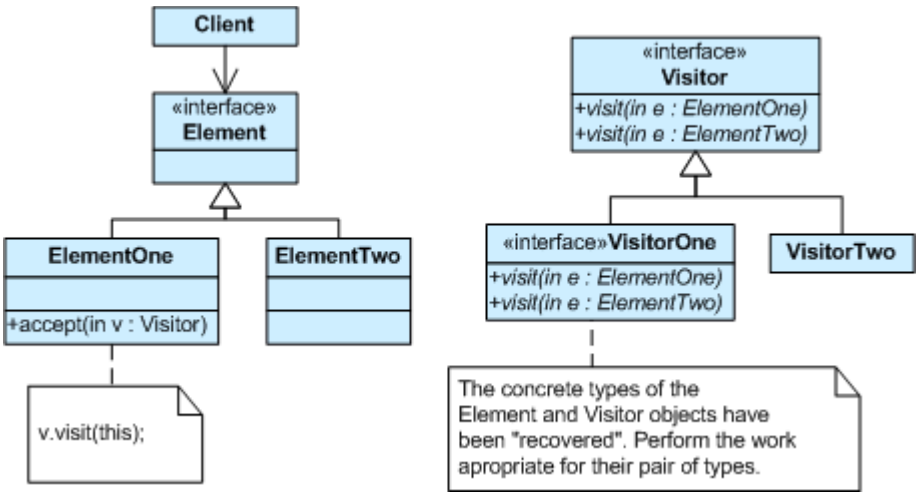
Паттерн Visitor позволяет легко добавлять новые операции – нужно просто добавить новый производный от `Visitor` класс. Однако паттерн Visitor следует использовать только в том случае, если подклассы `Element` совокупной иерархической структуры остаются стабильными (неизменяемыми). В противном случае, нужно приложить значительные усилия на обновление всей иерархии `Visitor`.

Иногда приводятся возражения по поводу использования паттерна Visitor, поскольку он разделяет данные и алгоритмы, что противоречит концепции объектно-ориентированного программирования. Однако успешный опыт применения STL, где разделение данных и алгоритмов положено в основу, доказывает возможность использования паттерна Visitor.

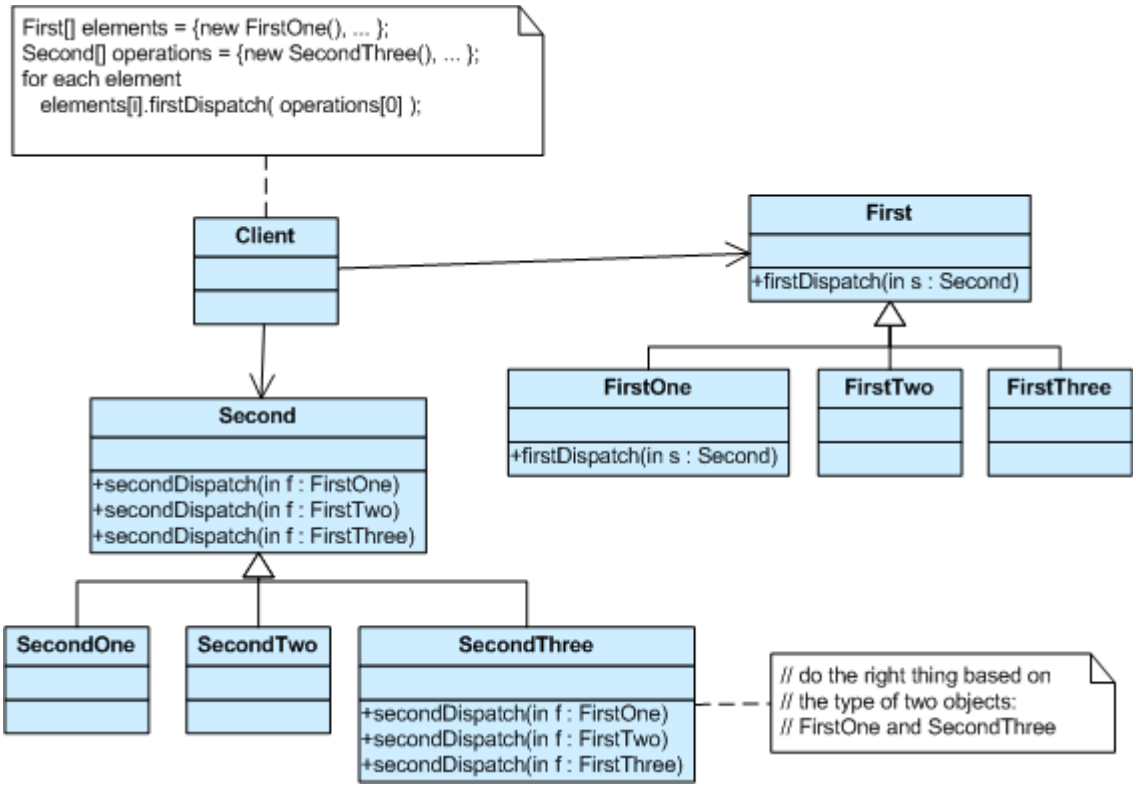
Структура паттерна Visitor

Несмотря на то, что реализации метода `accept()` в подклассах иерархии `Element` всегда одинаковая, этот метод не может быть перенесен в базовый класс `Element` и наследоваться производными классами. В этом случае адрес, получаемый с помощью указателя `this`, будет всегда соответствовать базовому типу `Element`.

UML-диаграмма классов паттерна Visitor



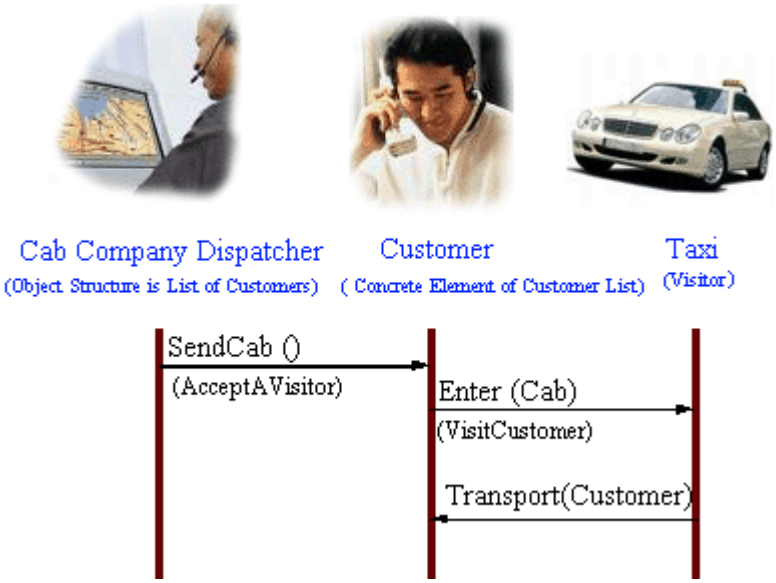
При вызове полиморфного метода `firstDispatch()` у объекта абстрактного типа `First` "восстанавливается" конкретный тип этого объекта. Когда вызывается полиморфный метод `secondDispatch()` объекта абстрактного типа `Second`, "восстанавливается" его конкретный тип. Теперь может выполняться функциональность, соответствующая этой паре типов.



Пример паттерна Visitor

Паттерн Visitor определяет операцию, выполняемую на каждом элементе из некоторой структуры без изменения классов этих объектов. Таксомоторная компания использует этот паттерн в своей работе. Когда клиент звонит в такую компанию, диспетчер отправляет к нему

свободное такси. После того как клиент садится в такси, его доставляют до места.



Использование паттерна Visitor

1. Убедитесь, что текущая иерархия Element будет оставаться стабильной и, что открытый интерфейс этих классов достаточно эффективен для доступа классов `Visitor`. Если это не так, то паттерн Visitor – не очень хорошее решение.
2. Создайте базовый класс `Visitor` с методами `visit(ElementXxx)` для каждого подкласса Element.
3. Добавьте метод `accept(Visitor)` в иерархию `Element`. Реализация этого метода во всех подклассах `Element` всегда одна и та же – `accept( Visitor v ) { v.visit( this ); }`. Из-за циклических зависимостей объявления классов `Element` и `Visitor` должны чередоваться.
4. Иерархия Element связана только с базовым классом `Visitor`, в то время как иерархия `Visitor` связана с каждым производным от `Element` классом. Если стабильность иерархии `Element` низкая, а стабильность иерархии `Visitor` высокая, рассмотрите возможность обмена "ролей" этих двух иерархий.
5. Для каждой "операции", которая должна выполняться для объектов `Element`, создайте производный от `Visitor` класс. Реализации метода `visit()` должны использовать открытый интерфейс класса `Element`.
6. Клиенты создают объекты `Visitor` и передают их каждому объекту `Element`, вызывая `accept()`.

Особенности паттерна Visitor

- Совокупная структура объектов `Elements` может определяться с помощью паттерна Composite.

- Для обхода Composite может использоваться Iterator.
- Паттерн Visitor демонстрирует классический прием восстановления информации о потерянных типах, не прибегая к понижающему приведению типов (dynamic cast).

Реализация паттерна Visitor

Реализация паттерна Visitor по шагам

1. Добавьте метод `accept(Visitor)` иерархию "элемент".
2. Создайте базовый класс `Visitor` и определите методы `visit()` для каждого типа "элемента".
3. Создайте производные классы `Visitor` для каждой "операции", исполняемой над "элементами".
4. Клиент создает объект `Visitor` и передает его в вызываемый метод `accept()`.

```
#include <iostream>
#include <string>
using namespace std;

// 1. Добавьте метод accept(Visitor) иерархию "элемент"
class Element
{
  public:
    virtual void accept(class Visitor &v) = 0;
};

class This: public Element
{
  public:
     /*virtual*/void accept(Visitor &v);
    string thiss()
    {
        return "This";
    }
};

class That: public Element
{
  public:
     /*virtual*/void accept(Visitor &v);
    string that()
    {
        return "That";
```

```
    }
};

class TheOther: public Element
{
  public:
     /*virtual*/void accept(Visitor &v);
    string theOther()
    {
        return "TheOther";
    }
};

// 2. Создайте базовый класс Visitor и определите
// методы visit()для каждого типа "элемента"
class Visitor
{
  public:
    virtual void visit(This *e) = 0;
    virtual void visit(That *e) = 0;
    virtual void visit(TheOther *e) = 0;
};

 /*virtual*/void This::accept(Visitor &v)
{
  v.visit(this);
}

 /*virtual*/void That::accept(Visitor &v)
{
  v.visit(this);
}

 /*virtual*/void TheOther::accept(Visitor &v)
{
  v.visit(this);
}

// 3. Создайте производные классы Visitor для каждой
// "операции", исполняемой над "элементами"
class UpVisitor: public Visitor
{
     /*virtual*/void visit(This *e)
    {
        cout << "do Up on " + e->thiss() << '\n';
    }
     /*virtual*/void visit(That *e)
    {
```

```
        cout << "do Up on " + e->that() << '\n';
    }
     /*virtual*/void visit(TheOther *e)
    {
        cout << "do Up on " + e->theOther() << '\n';
    }
};

class DownVisitor: public Visitor
{
     /*virtual*/void visit(This *e)
    {
        cout << "do Down on " + e->thiss() << '\n';
    }
     /*virtual*/void visit(That *e)
    {
        cout << "do Down on " + e->that() << '\n';
    }
     /*virtual*/void visit(TheOther *e)
    {
        cout << "do Down on " + e->theOther() << '\n';
    }
};

int main()
{
  Element *list[] =
  {
    new This(), new That(), new TheOther()
  };

  UpVisitor up;      // 4. Клиент создает
  DownVisitor down;  //    объекты Visitor
  for (int i = 0; i < 3; i++)
    // и передает каждый
    list[i]->accept(up);
  for (i = 0; i < 3; i++)
    // в вызываемый метод accept()
    list[i]->accept(down);
}
```

Вывод программы:

```
1do Up on This do Down on This do Up on That
2do Down on That do Up on TheOther do Down on TheOther
```

Реализация паттерна Visitor: до и после

До

Интерфейс для "операций" определяется в базовом классе Color и реализуется в его подклассах.

```
class Color
{
  public:
    virtual void count() = 0;
    virtual void call() = 0;
    static void report_num()
    {
        cout << "Reds " << s_num_red << ", Blus " << s_num_blu <<
'\n';
    }
  protected:
    static int s_num_red, s_num_blu;
};
int Color::s_num_red = 0;
int Color::s_num_blu = 0;

class Red: public Color
{
  public:
    void count()
    {
        ++s_num_red;
    }
    void call()
    {
        eye();
    }
    void eye()
    {
        cout << "Red::eye\n";
    }
};

class Blu: public Color
{
  public:
    void count()
    {
        ++s_num_blu;
    }
```

```
    void call()
    {
        sky();
    }
    void sky()
    {
        cout << "Blu::sky\n";
    }
};

int main()
{
  Color *set[] =
  {
    new Red, new Blu, new Blu, new Red, new Red, 0
  };
  for (int i = 0; set[i]; ++i)
  {
    set[i]->count();
    set[i]->call();
  }
  Color::report_num();
}
```

Вывод программы:

```
1Red::eye Blu::sky Blu::sky Red::eye Red::eye Reds 3, Blus 2
```

После

Иерархия `Color` определяет единственный метод `accept()`, а методы `count()` и `call()`реализованы в виде производных классов `Visitor`. При вызове метода `accept()` объекта `Color` происходит первая диспетчеризация, а при вызове метода `visit()` объекта `Visitor` – вторая. После этого на основе типов обоих объектов могут выполняться все необходимые действия.

```
class Color
{
  public:
    virtual void accept(class Visitor*) = 0;
};

class Red: public Color
{
  public:
     /*virtual*/void accept(Visitor*);
```

```
    void eye()
    {
        cout << "Red::eye\n";
    }
};
class Blu: public Color
{
  public:
     /*virtual*/void accept(Visitor*);
    void sky()
    {
        cout << "Blu::sky\n";
    }
};

class Visitor
{
  public:
    virtual void visit(Red*) = 0;
    virtual void visit(Blu*) = 0;
};

class CountVisitor: public Visitor
{
  public:
    CountVisitor()
    {
        m_num_red = m_num_blu = 0;
    }
     /*virtual*/void visit(Red*)
    {
        ++m_num_red;
    }
     /*virtual*/void visit(Blu*)
    {
        ++m_num_blu;
    }
    void report_num()
    {
        cout << "Reds " << m_num_red << ", Blus " << m_num_blu <<
'\n';
    }
  private:
    int m_num_red, m_num_blu;
};

class CallVisitor: public Visitor
{
```

```
  public:
     /*virtual*/void visit(Red *r)
    {
        r->eye();
    }
     /*virtual*/void visit(Blu *b)
    {
        b->sky();
    }
};

void Red::accept(Visitor *v)
{
  v->visit(this);
}

void Blu::accept(Visitor *v)
{
  v->visit(this);
}

int main()
{
  Color *set[] =
  {
    new Red, new Blu, new Blu, new Red, new Red, 0
  };
  CountVisitor count_operation;
  CallVisitor call_operation;
  for (int i = 0; set[i]; i++)
  {
    set[i]->accept(&count_operation);
    set[i]->accept(&call_operation);
  }
  count_operation.report_num();
}
```

Вывод программы:

```
1Red::eye Blu::sky Blu::sky Red::eye Red::eye Reds 3, Blus 2
```