Шаблоны проектирования

Что такое шаблоны (паттерны). История вопроса. Классификация шаблонов. Описание шаблонов. Результаты применения шаблонов.

Что такое шаблоны

- При создании программных систем перед разработчиками часто встает проблема выбора тех или иных проектных решений. В этих случаях на помощь приходят шаблоны (паттерны).
  - Дело в том, что почти наверняка подобные задачи уже решались ранее и уже существуют хорошо продуманные элегантные решения, составленные экспертами.
  - Если эти решения описать и систематизировать в каталоги, то они станут доступными менее опытным разработчикам, которые после изучения смогут использовать их как шаблоны или образцы для решения задач подобного класса.
- **Шаблон проектирования или паттерн** — повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

История вопроса

- Впервые, в конце 1970-х годов Кристофером Александером был разработан каталог шаблонов, предназначенных для проектирования зданий и городов.
- В конце 1980-х годов Кент Бек и Вард Каннингем попытались перенести идеи Александра в область разработки ПО, составив 5 небольших шаблонов для проектирования пользовательских интерфейсов на языке Smalltalk.
- В 1989 Джеймс Коплиен в целях обучения C++ внутри компании AT&T составил каталог идиом C++ (разновидность шаблонов, специфичных для языка программирования), а в 1991 на его основе вышла в свет книга "Advanced C++ Programming Styles and Idioms" .
- По-настоящему популярным применение шаблонов в индустрии разработки программного обеспечения стало после того, как в 1994 был опубликован каталог, включающий 23 шаблона объектно-ориентированного проектирования. Этот каталог настолько популярен, что часто упоминается как шаблоны **GoF** ("Gang of Four" или "банда четырех" по числу авторов).
- В настоящее время шаблоны продолжают непрерывно развиваться. Появляются новые шаблоны, категории и методы их описания.

Применение шаблонов

- В области разработки программных систем существует множество шаблонов, которые отличаются областью применения, масштабом, содержимым, стилем описания. Например, в зависимости от сферы применения существуют шаблоны
  - анализа,
  - проектирования,
  - тестирования,
  - документирования,
  - организации процесса разработки,
  - планирования проектов и другие.
- В настоящее время наиболее популярными шаблонами являются **шаблоны проектирования**

Классификация шаблонов

- Одной из распространенных классификаций таких шаблонов является классификация по степени детализации и уровню абстракции рассматриваемых систем.
- Шаблоны проектирования программных систем делятся на следующие категории:
  - Архитектурные шаблоны
  - Шаблоны проектирования
  - Идиомы

Классификация шаблонов

- **Архитектурные шаблоны**, являясь наиболее высокоуровневыми шаблонами, описывают структурную схему программной системы в целом.
- В данной схеме указываются отдельные функциональные составляющие системы, называемые подсистемами, а также взаимоотношения между ними. Примером архитектурного шаблона является хорошо известная программная парадигма «модель-представление-контроллер» (model-view-controller - MVC).
- В свою очередь, подсистемы могут состоять из архитектурных единиц уровнем ниже.

Классификация шаблонов

- **Шаблоны проектирования** описывают схемы детализации программных подсистем и отношений между ними, при этом они не влияют на структуру программной системы в целом и сохраняют независимость от реализации языка программирования.
  - Шаблоны GoF относятся именно к этой категории.
  - Под шаблонами проектирования объектно-ориентированных систем понимается описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте.
- В русскоязычной литературе обычно встречаются несколько вариантов перевода оригинального названия design patterns - **паттерны проектирования**, **шаблоны проектирования**, **образцы**.

Классификация шаблонов

- **Идиомы**, являясь низкоуровневыми паттернами, имеют дело с вопросами реализации какой-либо проблемы с учетом особенностей данного языка программирования.
  - При этом часто одни и те же идиомы для разных языков программирования выглядят по-разному или не имеют смысла вовсе.
  - Например, в C++ для устранения возможных утечек памяти могут использоваться интеллектуальные указатели. Интеллектуальный указатель содержит указатель на участок динамически выделенной памяти, который будет автоматически освобожден при выходе из зоны видимости. В среде .NET такой проблемы просто не существует, так как там используется автоматическая сборка мусора.
- Обычно, для использования идиом нужно глубоко знать особенности применяемого языка программирования.

Описание шаблонов

Задача каждого шаблона - дать четкое описание проблемы и ее решения в соответствующей области. Для этого могут использоваться разные форматы описаний от художественно-описательного до строгого, академического. В общем случае описание шаблона всегда содержит следующие элементы:

1. **Название шаблона**. Представляет собой уникальное смысловое имя, однозначно определяющее данную задачу или проблему и ее решение.
2. **Решаемая задача**. Здесь дается понимание того, почему решаемая проблема действительно является таковой, четко описывает ее границы.
3. **Решение**. Здесь указывается, как именно данное решение связано с проблемой, приводится пути ее решения.
4. **Результаты использования шаблона**. Обычно приводятся достоинства, недостатки и компромиссы.

Результаты применения шаблонов

Один из соавторов GoF, Джон Влиссидес приводит следующие преимущества применения паттернов проектирования:

- Они (шаблоны) позволяют суммировать опыт экспертов и сделать его доступным рядовым разработчикам.
- Имена паттернов образуют своего рода словарь, который позволяет разработчикам лучше понимать друг друга.
- Если в документации системы указано, какие паттерны в ней используются, это позволяет читателю быстрее понять систему.
- Паттерны упрощают реструктуризацию системы независимо от того, использовались ли паттерны при ее проектировании.

Правильно выбранные паттерны проектирования позволяют сделать программную систему более гибкой, ее легче поддерживать и модифицировать, а код такой системы в большей степени соответствует концепции повторного использования.

Шаблоны проектирования GoF

Шаблоны проектирования GoF делятся на 3 основные группы:

1. Порождающие — шаблоны, отвечающие за создание объектов;
2. Структурные — определяют структуру представления классов/объектов;
3. Паттерны поведения — шаблоны для реализации действий над объектами.

Шаблоны проектирования GoF

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

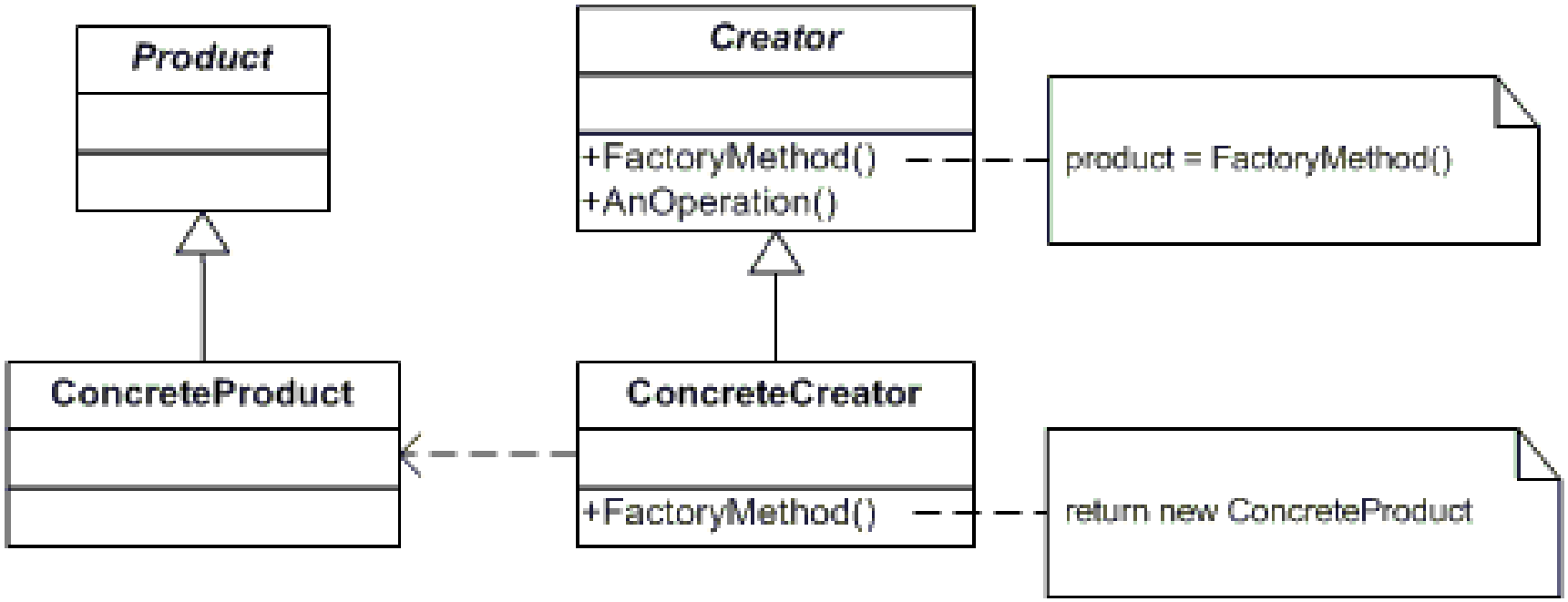
Порождающие шаблоны

- Создание новых объектов - наиболее распространенная задача при разработке программных систем.
- **Порождающие паттерны** проектирования предназначены для создания объектов, позволяя системе оставаться независимой как от самого процесса порождения, так и от типов порождаемых объектов:
  - **Factory Method (Фабричный метод)**
  - **Abstract Factory (Абстрактная Фабрика)**
  - **Builder (Строитель)**
  - **Prototype (Прототип)**
  - **Singleton (Одиночка)**

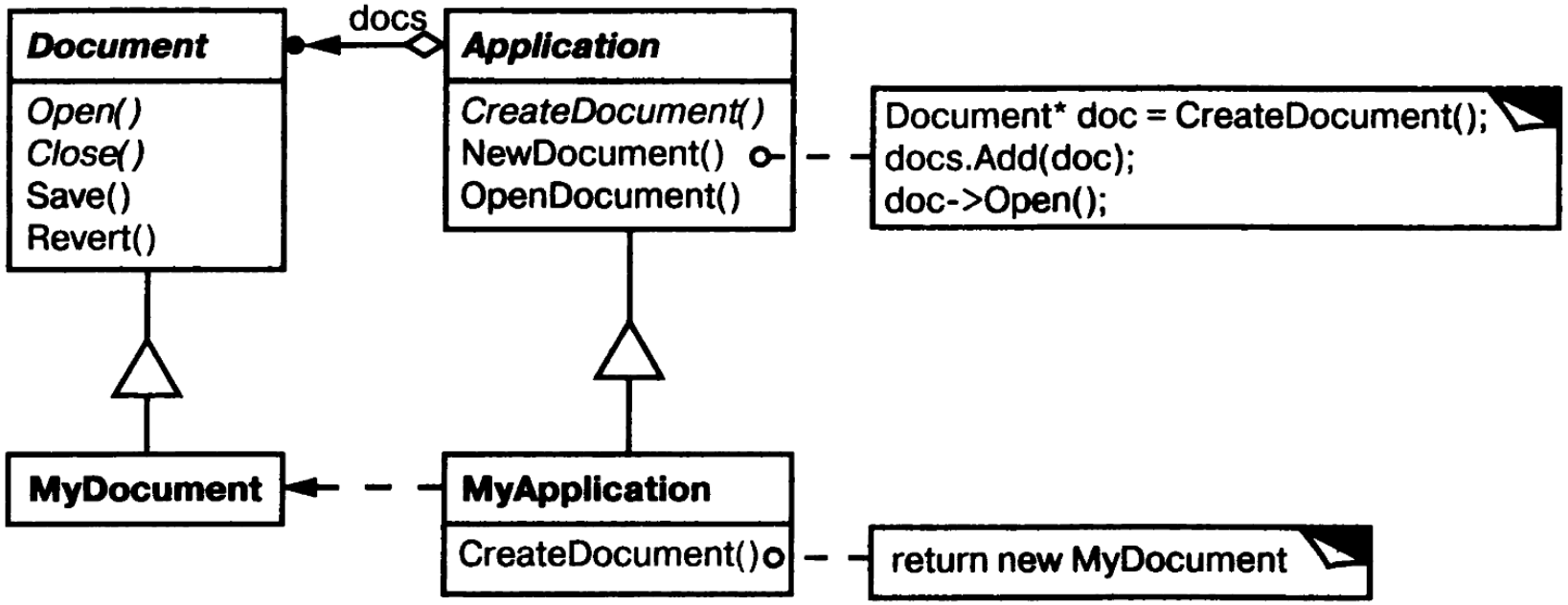
Фабричный метод

- Назначение:
  - Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой объект создавать (позволяет классу делегировать создание объектов подклассам).
- Применимость:
  - Классу заранее неизвестно, объекты каких классов ему надо создать
  - Класс спроектирован так, чтобы объекты, которые он создает, уточнялись подклассами
  - Класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и вы хотите локализовать (в вызывающем коде) знание о том, какой класс принимает эти обязанности на себя

Фабричный метод – структура



Фабричный метод – пример



Фабричный метод – пример

```
class Document //Абстрактный документ
{
  private:
    char name[20];
  public:
    Document(char *fn)
    {
        strcpy(name, fn);
    }
    virtual void Open() = 0;
    virtual void Close() = 0;
    char *GetName() {return name;}
};
```

Фабричный метод – пример

```
class MyDocument: public Document //Конкретный документ
{
  public:
    MyDocument(char *fn): Document(fn){}
    void Open()
    {
        cout << "   MyDocument: Open()" << endl;
    }
    void Close()
    {
        cout << "   MyDocument: Close()" << endl;
    }
};
```

Фабричный метод – пример

```
class Application //Абстрактный базовый класс
{
  public:
    Application() { _index = 0; cout << "Application: ctor" << endl;) }
    NewDocument(char *name)
    {
        cout << "Application: NewDocument()" << endl;
        _docs[_index] = CreateDocument(name); //вызываем метод потомка!
        _docs[_index++]->Open();
    }
    void ReportDocs();
    virtual Document *CreateDocument(char*) = 0; //для переопределения потомком
  private:
    int _index;
    Document *_docs[10]; //ссылаемся на базовый класс документа
};
```

Фабричный метод – пример

```
void Application::ReportDocs()
{
  cout << "Application: ReportDocs()" << endl;
  for (int i = 0; i < _index; i++)  cout << "    " << _docs[i]->GetName() << endl;
}

class MyApplication: public Application //Конкретный класс
{
  public:
    MyApplication() { cout << "MyApplication: ctor" << endl; }
    Document *CreateDocument(char *fn)
    {
        cout << "    MyApplication: CreateDocument()" << endl;
        return new MyDocument(fn);
    }
};
```

Фабричный метод – пример

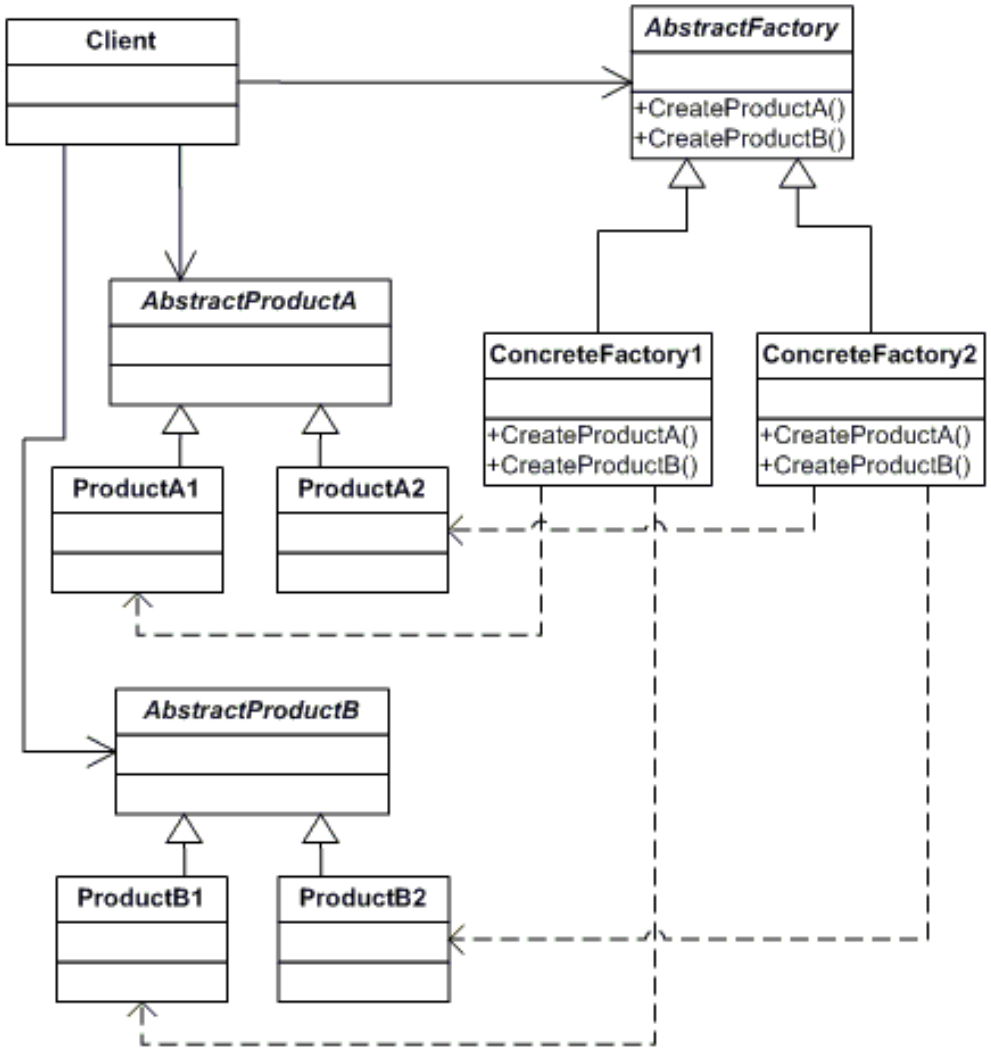
```
int main()
{
MyApplication myApp;

  myApp.NewDocument("foo");
  myApp.NewDocument("bar");
  myApp.ReportDocs();
}
```

```
Application: ctor
MyApplication: ctor
Application: NewDocument()
   MyApplication: CreateDocument()
   MyDocument: Open()
Application: NewDocument()
   MyApplication: CreateDocument()
   MyDocument: Open()
Application: ReportDocs()
   foo
   bar
```

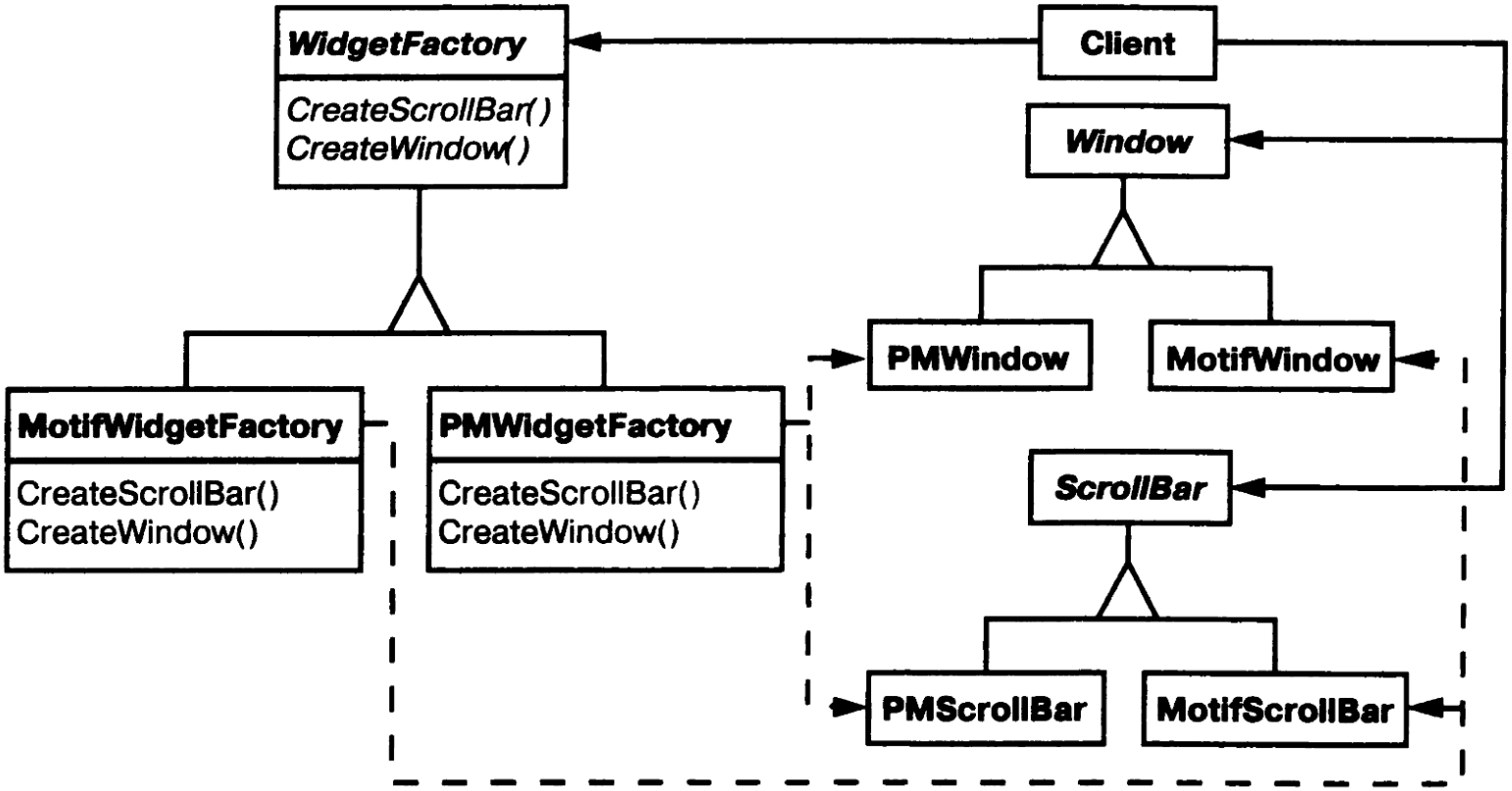
Абстрактная фабрика

- Назначение:
  - Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.
- Применимость:
  - Система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты
  - Входящие в семейство взаимосвязанные объекты должны использоваться вместе и вам необходимо обеспечить выполнение этого ограничения
  - Система должна конфигурироваться одним из семейств входящих в него объектов
  - Вы хотите представить библиотеку объектов, раскрывая только их интерфейс, но не реализацию

Абстрактная фабрика – структура



Абстрактная фабрика – пример



Абстрактная фабрика – пример

```
class Unit //Abstract product
{
  public:
    Unit () { id_ = total_++; }
    virtual void draw() = 0;
  protected:
    int id_;
    static int total_;
};
int Unit::total_ = 0;

class Warrior : public Unit //Product
{
  public:
    void draw() { cout << " Warrior " << id_ << ": draw" << endl;
}
```

Абстрактная фабрика – пример

```
class Archer : public Unit //Product
 {
  public:
    void draw() { cout << " Archer " << id_ << ": draw" << endl; }
};
class Rifleman : public Unit //Product
{
  public:
    void draw() { cout << " Rifleman " << id_ << ": draw" << endl; }
};
class Artillery : public Unit //Product
{
  public:
    void draw() { cout << " Artillery " << id_ << ": draw" << endl; }
};
```

Абстрактная фабрика – пример

```
class UnitFactory //Abstract factory
{
  public:
    virtual Unit* createMeleeUnit() = 0;
    virtual Unit* createRangedUnit() = 0;
};
class AncientEraUnitFactory : public UnitFactory { //Concrete factory
  public:
    Unit* createMeleeUnit() { return new Warrior; }
    Unit* createRangedUnit() { return new Archer; }
};
class ModernEraUnitFactory : public UnitFactory { //Concrete factory
  public:
    Unit* createMeleeUnit()   { return new Rifleman; }
    Unit* createRangedUnit() { return new Artillery; }
};
```

Абстрактная фабрика – пример

```
int main() {
  UnitFactory* factory = new
AncientEraUnitFactory;
// UnitFactory* factory = new
ModernEraUnitFactory;

  Unit* units[3];

 units[0] = factory-> createMeleeUnit();
 units[1] = factory->createRangedUnit ();
 units[2] = factory-> createMeleeUnit();

  for (int i=0; i < 3; i++) {
    units[i]->draw();
  }
}
```

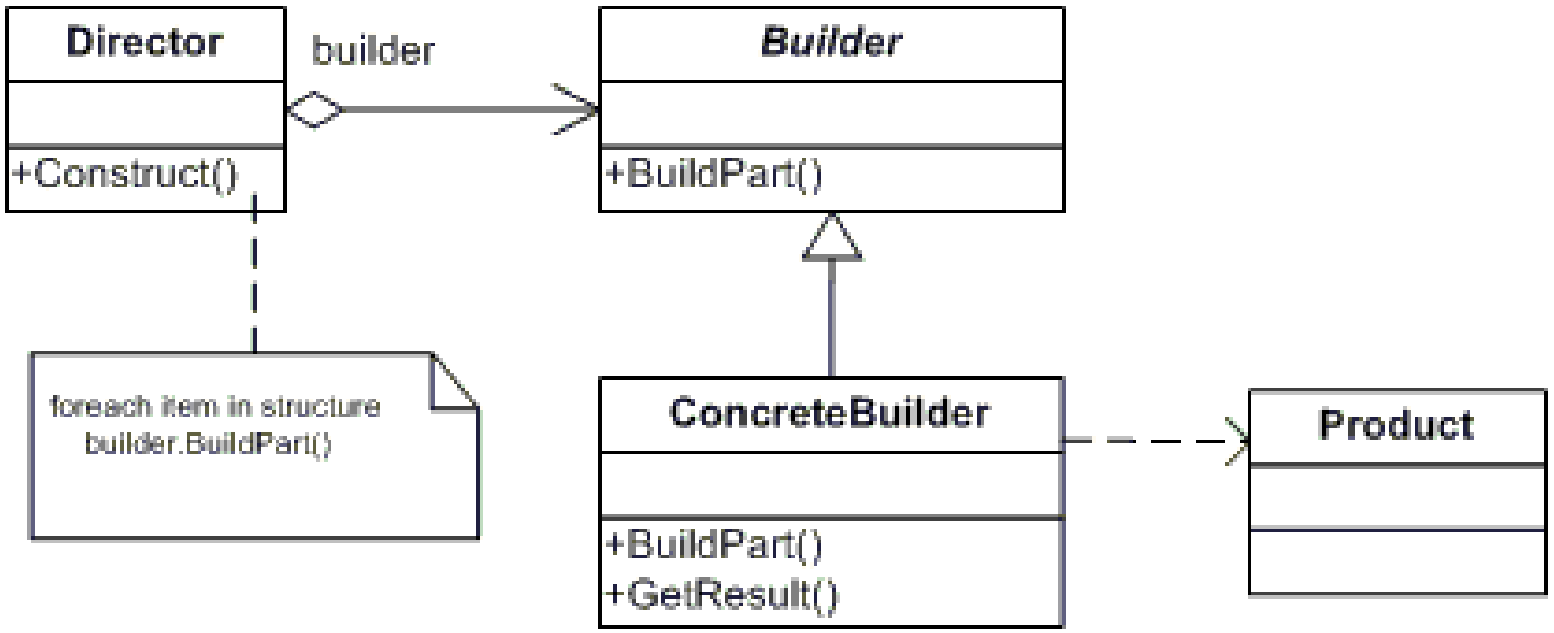
```
Warrior 0: draw
Archer 1: draw
Warrior 2: draw
```

```
Rifleman 0: draw
Artillery 1: draw
Rifleman 2: draw
```

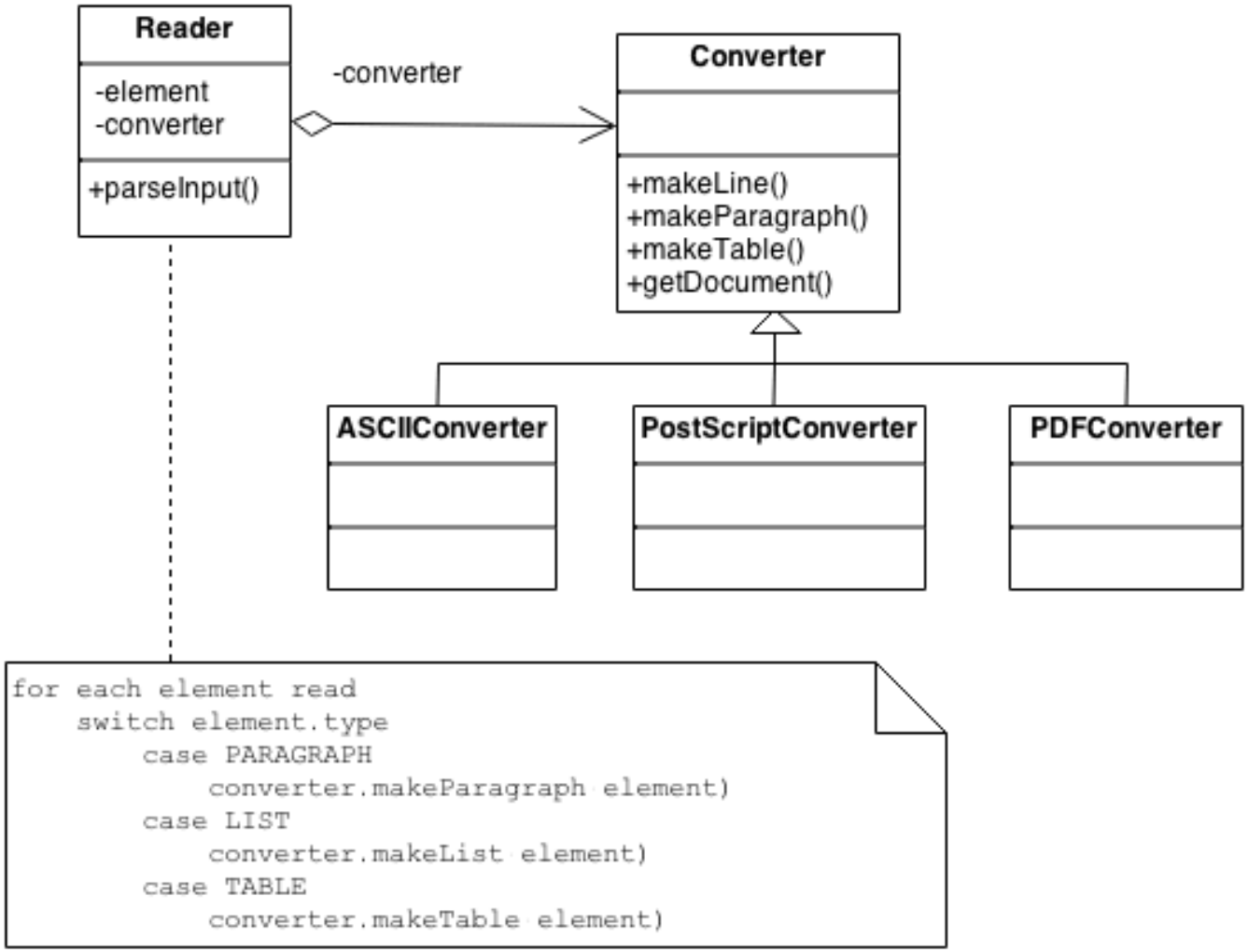
Строитель

- Назначение:
  - Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления
- Применимость:
  - Алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой
  - Процесс конструирования объекта должен обеспечивать различные представления конструируемого объекта

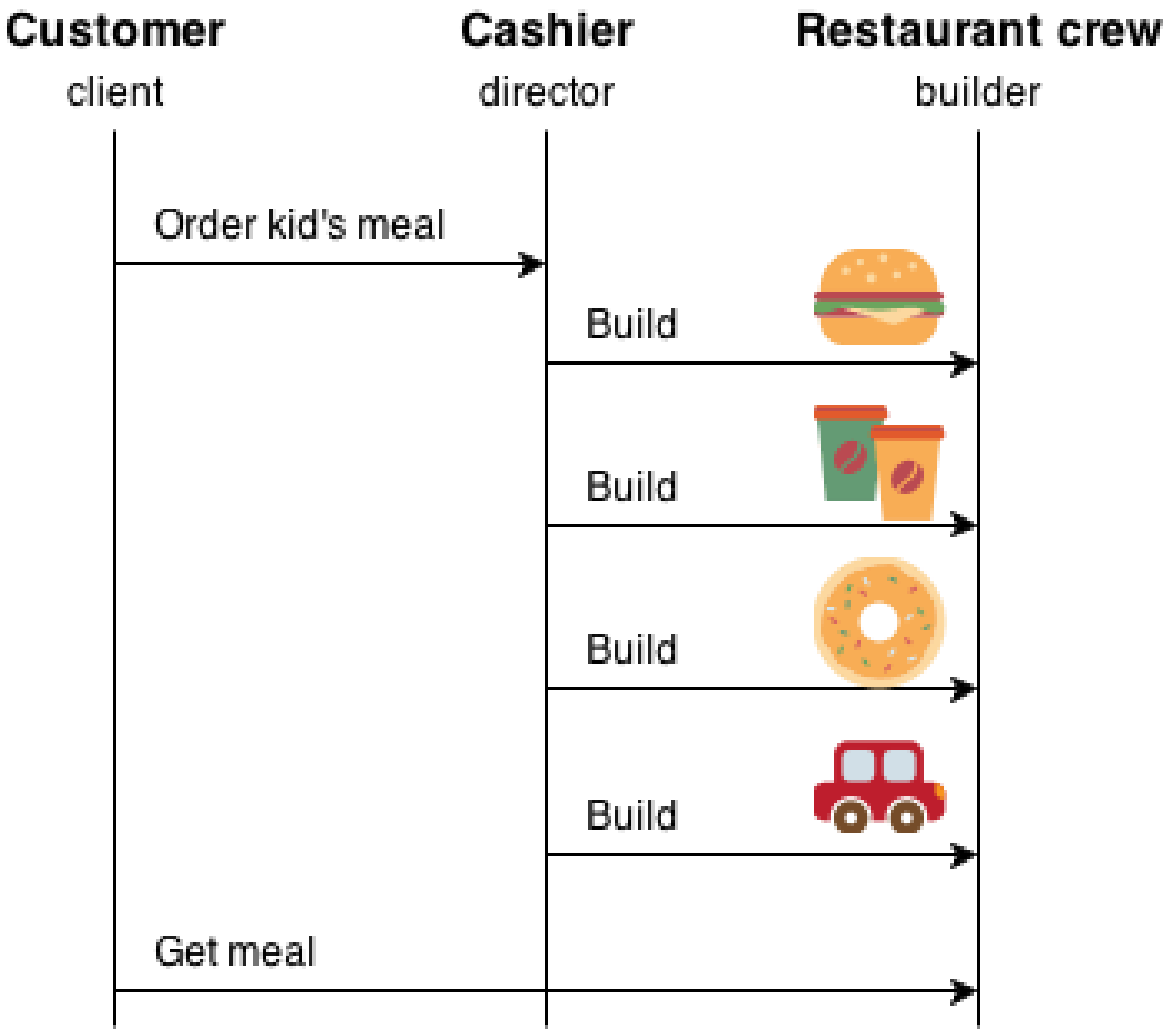
Строитель - структура



Строитель - пример



Строитель - пример



Строитель - пример

```
class Meal //Product
{
    public:
        char name[20];
        char* mainDish;
        char* sideDish;
        char* drink;
        char* dessert;

        Meal(char * _name) {  strcpy(name, _name);  }
        void Print()
        {
            cout << name << ": " << mainDish << ", " << sideDish << ", " ;
            cout << drink << " and " << dessert << endl;
        }
};
```

Строитель - пример

```
class Cashier //Director
{
    public:
        void OrderMeal(Cook* builder)
        {
            builder->BuildMainDish();
            builder->BuildSideDish();
            builder->BuildDrink();
            builder->BuildDessert();
        }
};
```

```
class Cook //Abstract Builder
{
    protected:
        Meal* meal;
    public:
        Meal* GetMeal() { return meal; };
        virtual void BuildMainDish() = 0;
        virtual void BuildSideDish()= 0;
        virtual void BuildDrink()= 0;
        virtual void BuildDessert()= 0;
};
```

Строитель - пример

```
class FastFoodCook : public Cook {//Concrete builder
    public:
        FastFoodCook() { meal = new Meal("FastFood dinner"); }
        void BuildMainDish() { meal->mainDish = "Cheeseburger"; };
        void BuildSideDish() { meal->sideDish = "Fries"; };
        void BuildDrink() { meal->drink = "Coke"; };
        void BuildDessert() { meal->dessert = "Donut"; };
};
class RussianCook : public Cook { //Concrete builder
    public:
        RussianCook() { meal = new Meal("Russian dinner"); }
        void BuildMainDish() { meal->mainDish = "Kotletki"; };
        void BuildSideDish() { meal->sideDish = "Pyureshka"; };
        void BuildDrink() { meal->drink = "Kompot"; };
        void BuildDessert() { meal->dessert = "Bulochka"; };
};
```

Строитель - пример

```
void main() {
    Cook* cook;
    Meal* meal;
    Cashier* cashier = new Cashier();

    cook = new FastFoodCook();
    cashier->OrderMeal(cook);
    meal = cook->GetMeal();
    meal->Print();

    cook = new RussianCook();
    cashier->OrderMeal(cook);
    meal = cook->GetMeal();
    meal->Print();
}
```

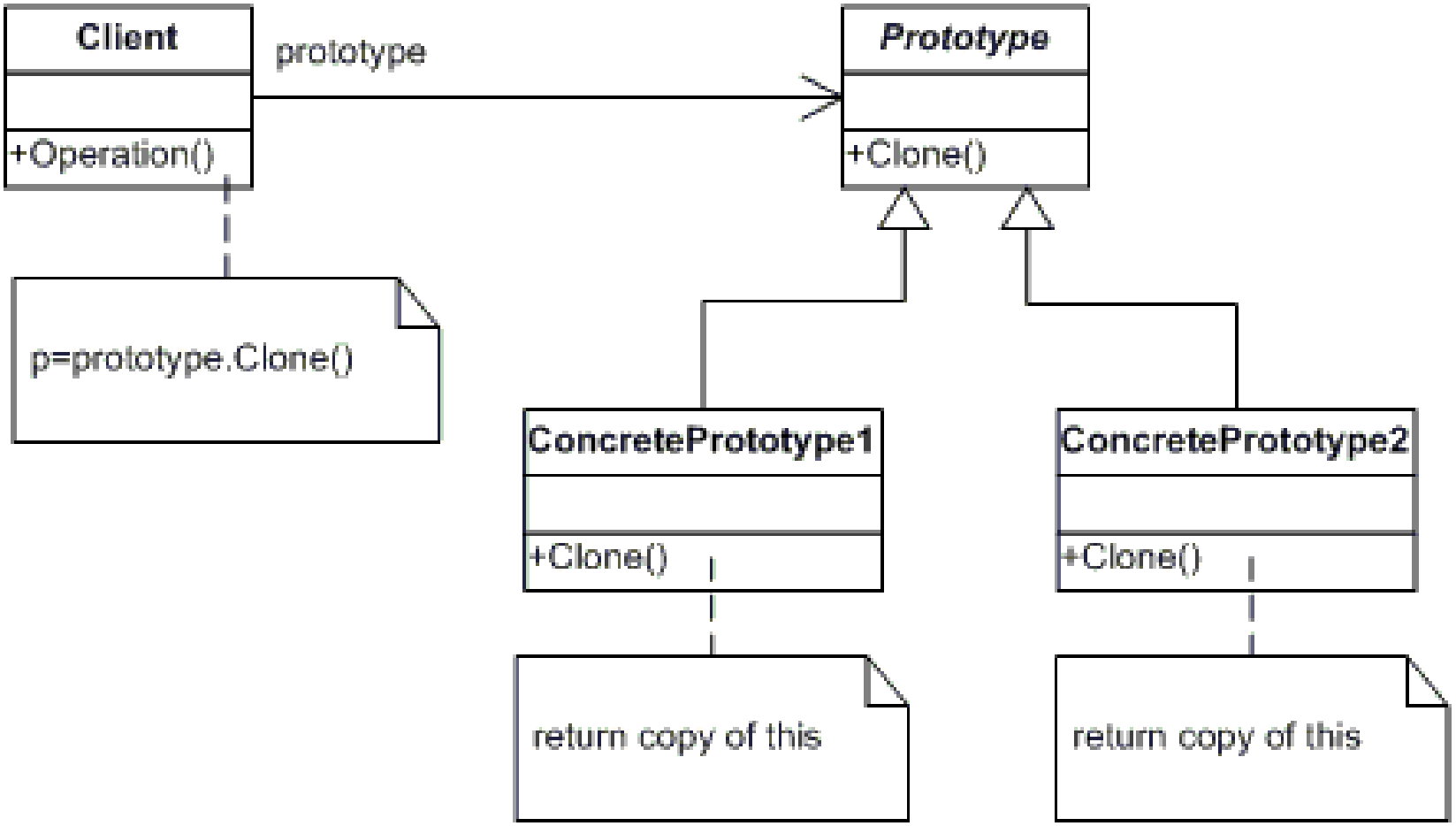
FastFood dinner: Cheeseburger, Fries, Coke and Donut

Russian dinner: Kotletki, Pyureshka, Kompot and Bulochka

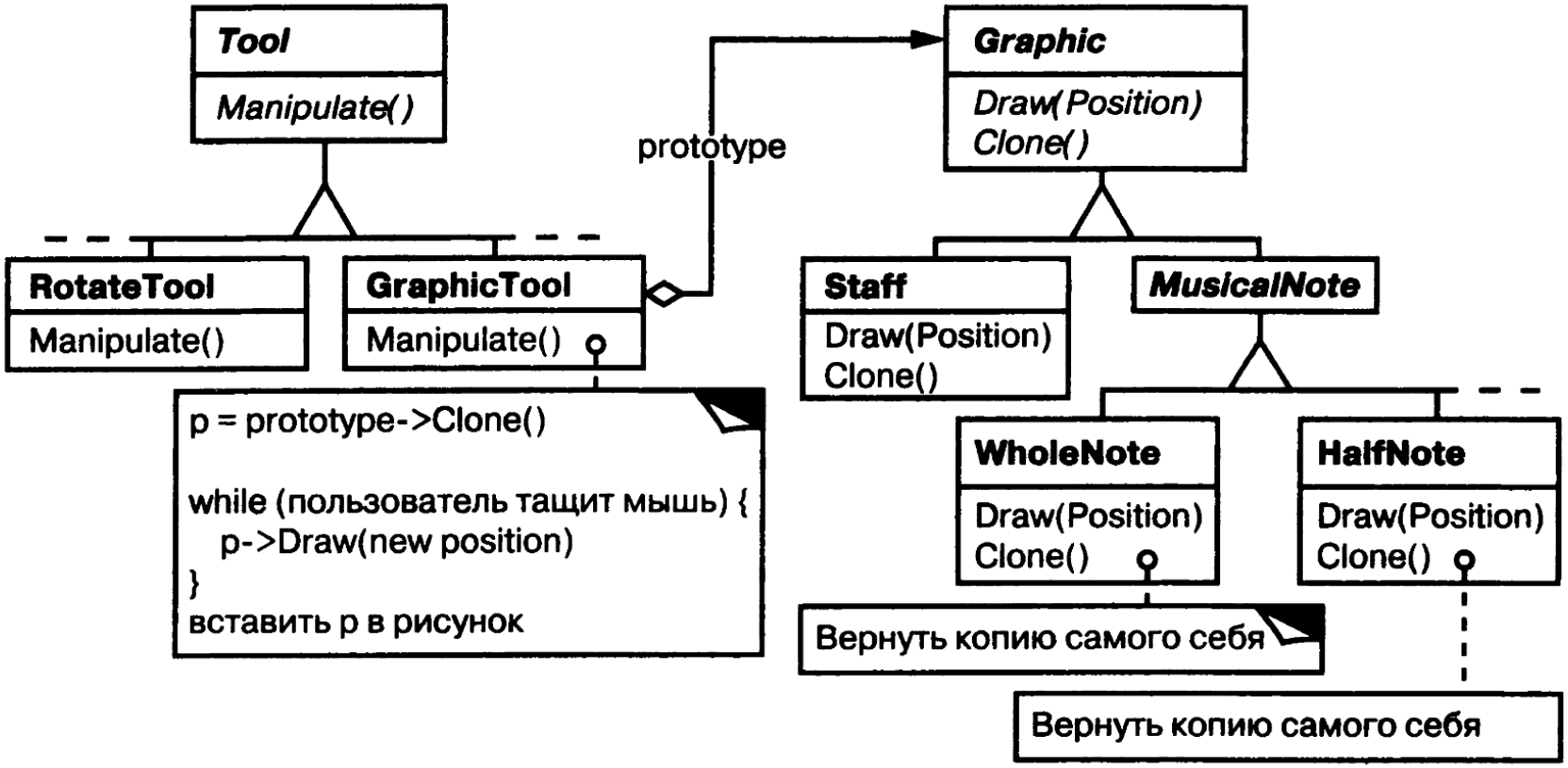
Прототип

- Назначение:
  - Задает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем копирования этого прототипа
- Применимость:
  - Инстанцируемые классы определяются по время выполнения, например, с помощью динамической загрузки
  - Для того, чтобы избежать построения иерархии фабрик, параллельной иерархии классов продуктов
  - Если экземпляры класса могут находиться в ограниченном множестве состояний

Прототип - структура



Прототип - пример



Прототип - пример

```
class Unit {
  public:
    Unit () {
        id_ = total_++;
        health=100;
    }
    virtual void Draw() = 0;
    virtual Unit* Clone() = 0;
    void TakeDamage(int dmg) { health -= dmg; }
  protected:
    int id_;
    int health;
    static int total_;
};
int Unit::total_ = 0;
```

Прототип - пример

```
class Warrior : public Unit {
  public:
    void Draw() { cout << " Warrior " << id_ << " Health: " << health << endl; }
    Unit* Clone() {
        Warrior* w = new Warrior;
        w->health = this->health;
        return w;
    }
};
```

```
int main()
{
    Unit* u1 = new Warrior;
    u1->Draw();
    u1->TakeDamage(rand() % 100 + 1);
    std::cout << "Boom!" << std::endl;
    u1->Draw();
    Unit* u2 = u1->Clone();
    u2->Draw();
}
```

```
 Warrior 0 Health: 100
Boom!
 Warrior 0 Health: 16
 Warrior 1 Health: 16
```

Одиночка

- Назначение:
  - Гарантирует, что у класса есть только один экземпляр и предоставляет к нему глобальную точку доступа.
- Применимость:
  - Необходим ровно один экземпляр некоего класса, легко доступный всем клиентам.
  - Единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

Одиночка - структура

Одиночка - пример

```
class GlobalClass{
  private:
    int m_value;
    static GlobalClass *s_instance;
    GlobalClass(int v = 0)    { m_value = v; }
  public:
    int get_value() { return m_value; }
    void set_value(int v) { m_value = v; }

    static GlobalClass *instance()
    {
        if (!s_instance)
            s_instance = new GlobalClass;
        return s_instance;
    }
};
```

Одиночка - пример

```
main: global_ptr is 0
foo: global_ptr is 1
bar: global_ptr is 2
main: global_ptr is 2
```

```
GlobalClass *GlobalClass::s_instance = 0;
void foo(void) {
  GlobalClass::instance()->set_value(1);
  std::cout << "foo: global_ptr is " << GlobalClass::instance()->get_value() << '\n';
}
void bar(void) {
  GlobalClass::instance()->set_value(2);
  std::cout << "bar: global_ptr is " << GlobalClass::instance()->get_value() << '\n';
}
int main() {
  std::cout << "main: global_ptr is " << GlobalClass::instance()->get_value() << '\n';
  foo();
  bar();
  std::cout << "main: global_ptr is " << GlobalClass::instance()->get_value() << '\n';
}
```

Резюме

- Иногда порождающие шаблоны конкурируют друг с другом:
  - иногда могут оказаться одинаково применимы как **Прототип**, так и **Абстрактная Фабрика**.
- В других случаях, они могут дополнять друг друга:
  - **Абстрактная Фабрика** может хранить набор **Прототипов**, которые она будет клонировать для построения новых объектов,
  - **Строитель** может использовать любой другой шаблон, чтобы реализовать алгоритм построения объекта.
  - **Абстрактная Фабрика**, **Строитель** и **Прототип** могут использовать **Одиночку** в своей внутренней реализации.
- **Абстрактная Фабрика**, **Строитель** и **Прототип** определяют фабричный объект, который отвечает за создание правильных экземпляров данного класса объектов, и может являться параметром системы.
  - **Абстрактная Фабрика** при этом может создавать объекты целой группы классов.
  - Фабричный объект **Строителя** умеет строить сложные объекты инкрементально, согласно сложному протоколу.
  - Фабричный объект **Прототипа** строит копию самого себя.
- Классы **Абстрактной Фабрики** часто используют **Фабричный Метод**, но могут и реализовываться через **Прототип**.
- **Абстрактная Фабрика** может использоваться как альтернатива **Фасада** чтобы скрыть платформо-специфичные классы.

Резюме

- **Строитель** в отношении процесса создания объекта выполняет роль **Стратегии** в отношении алгоритма.
- **Строитель** часто используется для построения **Компоновщика (Композита).**
- Вызовы **Фабричных Методов** часто производятся внутри **Шаблонных Методов**.
- **Фабричный Метод**: создание через наследование. **Прототип**: создание через делегирование.
- Часто, проектирование начинается с использования **Фабричного Метода** (как наиболее простого решения) и эволюционирует впоследствии к **Абстрактной Фабрике**, **Строителю** или **Прототипу** (более гибко, но более сложно), если проектировщику кажется, что необходимо больше гибкости.
- **Прототип** полезен там, где нет наследования, но есть трудоемкая операция инициализации объекта. **Фабричный Метод** полезен там, где есть наследование, но не требуется инициализация.
- Проекты, где широко используются объектные (а не классовые) шаблоны - **Компоновщик (Композит) и Декоратор** – часто выигрывают от применения Прототипа