Тема 3. Порождающие паттерны

Пожалуй, создание новых объектов является наиболее распространенной задачей, встающей перед разработчиками программных систем. Порождающие паттерны проектирования предназначены для создания объектов, позволяя системе оставаться независимой как от самого процесса порождения, так и от типов порождаемых объектов. Прежде чем рассматривать особенности каждого из порождающих паттернов, рассмотрим на примере типичные проблемы, встающие перед разработчиками при порождении в системе объектов новых типов.

Пусть мы разрабатываем многообещающую стратегическую игру под названием "Пунические войны", описывающую великое военное противостояние между Римской Республикой и Карфагеном (264 — 146 г. до н. э.). Персонажами игры могут быть воины трех типов: пехота, конница и лучники. Каждый из этих видов обладает своими отличительными характеристиками, такими как внешний вид, боевая мощь, скорость передвижения и степень защиты. Несмотря на такие отличия, у всех видов боевых единиц есть общие черты. Например, все они могут передвигаться по игровому полю в различных направлениях, хотя всадники делают это быстрее всех. Или каждая боевая единица имеет свой уровень здоровья, и если он становится равным нулю, воин погибает. При этом уничтожить лучника значительно проще, чем другие виды воинов.

В будущем, если игра окажется успешной, мы будем развивать ее дальше. Например, мы могли бы добавить новые виды воинов, такие как боевые слоны, или усовершенствовать существующие, разделив пехоту на легковооруженных и тяжеловооруженных пехотинцев. Для внесения подобных изменений без модификации существующего кода, мы должны уже сейчас постараться сделать игру максимально независимой от конкретных типов персонажей. Казалось бы, для этого достаточно использовать следующую иерархию классов.

```
1    class Warrior
2    {
3      public:
4        virtual void info() = 0;
5        virtual ~Warrior() {}
6    };
7
8    class Infantryman: public Warrior
```

```
9   {
10    public:
11       void info() { cout << "Infantryman" << endl; }
12  };
13
14  class Archer: public Warrior
15  {
16    public:
17      void info() { cout << "Archer" << endl; }
18  };
19
20  class Horseman: public Warrior
21  {
22    public:
23      void info() { cout << "Horseman" << endl; }
24  };
```

Полиморфный базовый класс Warrior определяет общий интерфейс, а производные от него классы Infantryman, Archer и Horseman реализуют особенности каждого вида воина. Сложность заключается в том, что хотя код системы и оперирует готовыми объектами через соответствующие общие интерфейсы, в процессе игры требуется создавать новые персонажи, непосредственно указывая их конкретные типы. Если код их создания рассредоточен по всему приложению, то добавлять новые типы персонажей или заменять существующие будет затруднительно.

В таких случаях на помощь приходит **фабрика объектов**, локализующая создание объектов. Работа фабрики объектов напоминает функционирование виртуального конструктора, - мы можем создавать объекты нужных классов, не указывая напрямую их типы. В самом простом случае, для этого используются идентификаторы типов. Следующий пример демонстрирует простейший вариант фабрики объектов - **фабричную функцию**.

```
1   enum Warrior_ID { Infantryman_ID=0, Archer_ID, Horseman_ID };
2
3   Warrior * createWarrior( Warrior_ID id  )
4   {
5      Warrior * p;
6      switch (id)
7      {
```

```
8           case Infantryman_ID:
9               p = new Infantryman();
10              break;
11          case Archer_ID:
12              p = new Archer();
13              break;
14          case Horseman_ID:
15              p = new Horseman();
16              break;
17          default:
18              assert( false);
19      }
20      return p;
21  }
```

Теперь, скрывая детали, код создания объектов разных типов игровых персонажей сосредоточен в одном месте, а именно, в фабричной функции createWarrior( ). Эта функция получает в качестве аргумента тип объекта, который нужно создать, создает его и возвращает соответствующий указатель на базовый класс.

Несмотря на очевидные преимущества, у этого варианта фабрики также существуют недостатки. Например, для добавления нового вида боевой единицы необходимо сделать несколько шагов - завести новый идентификатор типа и модифицировать код фабричной функции createWarrior( ).

Познакомившись с основными проблемами, возникающими при создании объектов новых типов, кратко рассмотрим особенности каждого из порождающих паттернов (шаблонов).

Паттерн Factory Method развивает тему фабрики объектов дальше, перенося создание объектов в специально предназначенные для этого классы. В его классическом варианте вводится полиморфный класс Factory, в котором определяется интерфейс фабричного метода, подобного createWarrior( ), а ответственность за создание объектов конкретных классов переносится на производные от Factory классы, в которых этот метод переопределяется.
Паттерн Abstract Factory использует несколько фабричных методов и предназначен для создания целого семейства или группы взаимосвязанных объектов. Для случая нашей стратегической игры такими взаимосвязанными объектами могли бы быть объекты всех персонажей (воины и крестьяне) для конкретной расы.

Паттерн Builder определяет процесс поэтапного конструирования сложного объекта, в результате которого могут получаться разные представления этого объекта.
Паттерн Prototype создает новые объекты с помощью прототипов. Прототип - некоторый объект, умеющий создавать по запросу копию самого себя.
Паттерн Singleton контролирует создание единственного экземпляра некоторого класса и предоставляет доступ к нему.
Паттерн Object Pool используется в случае, когда создание объекта требует больших затрат или может быть создано только ограниченное количество объектов некоторого класса.

Паттерн 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).

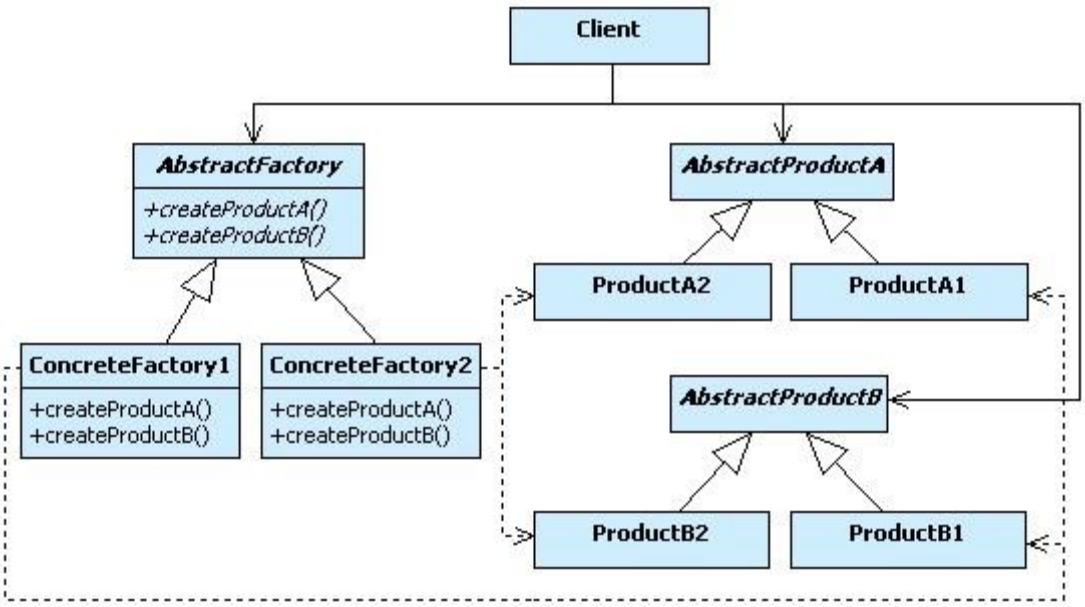
Любое семейство или группа взаимосвязанных объектов характеризуется несколькими общими типами создаваемых продуктов, при этом сами продукты таких типов будут различными для разных семейств. Например, для случая стратегической игры общими типами создаваемых продуктов будут пехота, лучники и конница, при 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.

```
1    #include <iostream>
2    #include <vector>
3
4    // Абстрактные базовые классы всех возможных видов воинов
5    class Infantryman
6    {
7      public:
8        virtual void info() = 0;
9        virtual ~Infantryman() {}
10   };
11
12   class Archer
13   {
14     public:
15       virtual void info() = 0;
```

```
16      virtual ~Archer() {}
17  };
18
19  class Horseman
20  {
21    public:
22      virtual void info() = 0;
23      virtual ~Horseman() {}
24  };
25
26
27  // Классы всех видов воинов Римской армии
28  class RomanInfantryman: public Infantryman
29  {
30    public:
31      void info() {
32        cout << "RomanInfantryman" << endl;
33      }
34  };
35
36  class RomanArcher: public Archer
37  {
38    public:
39      void info() {
40        cout << "RomanArcher" << endl;
41      }
42  };
43
44  class RomanHorseman: public Horseman
45  {
46    public:
47      void info() {
48        cout << "RomanHorseman" << endl;
49      }
50  };
51
52
53  // Классы всех видов воинов армии Карфагена
54  class CarthaginianInfantryman: public Infantryman
```

```
55  {
56   public:
57     void info() {
58       cout << "CarthaginianInfantryman" << endl;
59     }
60  };
61
62  class CarthaginianArcher: public Archer
63  {
64   public:
65     void info() {
66       cout << "CarthaginianArcher" << endl;
67     }
68  };
69
70  class CarthaginianHorseman: public Horseman
71  {
72   public:
73     void info() {
74       cout << "CarthaginianHorseman" << endl;
75     }
76  };
77
78
79  // Абстрактная фабрика для производства воинов
80  class ArmyFactory
81  {
82   public:
83     virtual Infantryman* createInfantryman() = 0;
84     virtual Archer* createArcher() = 0;
85     virtual Horseman* createHorseman() = 0;
86     virtual ~ArmyFactory() {}
87  };
88
89
90  // Фабрика для создания воинов Римской армии
91  class RomanArmyFactory: public ArmyFactory
92  {
93   public:
```

```
94      Infantryman* createInfantryman() {
95        return new RomanInfantryman;
96      }
97      Archer* createArcher() {
98        return new RomanArcher;
99      }
100     Horseman* createHorseman() {
101       return new RomanHorseman;
102     }
103 };
104
105
106 // Фабрика для создания воинов армии Карфагена
107 class CarthaginianArmyFactory: public ArmyFactory
108 {
109   public:
110     Infantryman* createInfantryman() {
111       return new CarthaginianInfantryman;
112     }
113     Archer* createArcher() {
114       return new CarthaginianArcher;
115     }
116     Horseman* createHorseman() {
117       return new CarthaginianHorseman;
118     }
119 };
120
121
122 // Класс, содержащий всех воинов той или иной армии
123 class Army
124 {
125   public:
126    ~Army() {
127       int i;
128       for(i=0; i<vi.size(); ++i)  delete vi[i];
129       for(i=0; i<va.size(); ++i)  delete va[i];
130       for(i=0; i<vh.size(); ++i)  delete vh[i];
131     }
132     void info() {
```

```
133       int i;
134       for(i=0; i<vi.size(); ++i)  vi[i]->info();
135       for(i=0; i<va.size(); ++i)  va[i]->info();
136       for(i=0; i<vh.size(); ++i)  vh[i]->info();
137     }
138     vector<Infantryman*> vi;
139     vector<Archer*> va;
140     vector<Horseman*> vh;
141   };
142
143
144   // Здесь создается армия той или иной стороны
145   class Game
146   {
147     public:
148     Army* createArmy( ArmyFactory& factory ) {
149       Army* p = new Army;
150       p->vi.push_back( factory.createInfantryman());
151       p->va.push_back( factory.createArcher());
152       p->vh.push_back( factory.createHorseman());
153       return p;
154     }
155   };
156
157
158   int main()
159   {
160     Game game;
161     RomanArmyFactory ra_factory;
162     CarthaginianArmyFactory ca_factory;
163
164     Army * ra = game.createArmy( ra_factory);
165     Army * ca = game.createArmy( ca_factory);
166     cout << "Roman army:" << endl;
167     ra->info();
168     cout << "\nCarthaginian army:" << endl;
169     ca->info();
170     // ...
171   }
```

Вывод программы будет следующим:

```
1  Roman army:
2  RomanInfantryman
3  RomanArcher
4  RomanHorseman
5
6  Carthaginian army:
7  CarthaginianInfantryman
8  CarthaginianArcher
9  CarthaginianHorseman
```

Результаты применения паттерна Abstract Factory

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 Строитель

Назначение паттерна

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

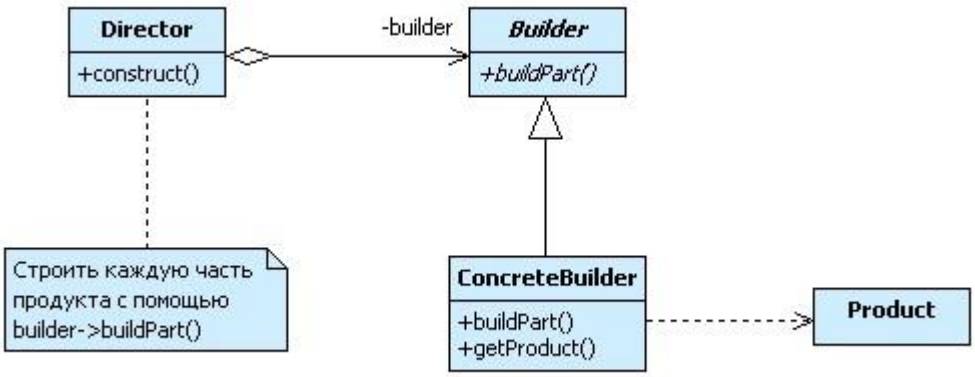
Паттерн Builder отделяет алгоритм поэтапного конструирования сложного продукта (объекта) от его внешнего представления так, что с помощью одного и того же алгоритма можно получать разные представления этого продукта.

> 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.

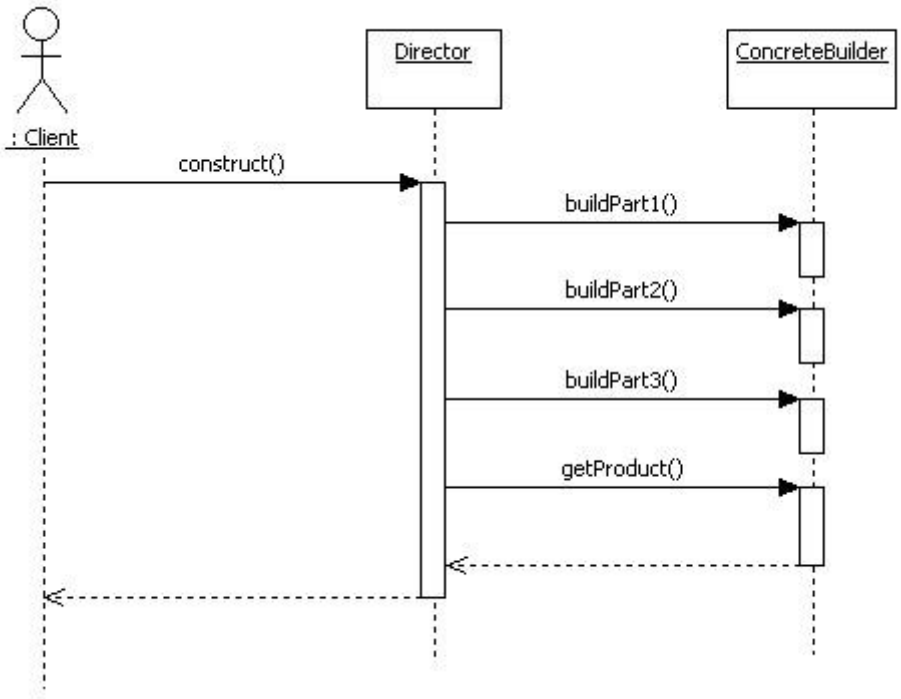
UML-диаграмма классов паттерна Builder



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.

```
1   #include <iostream>
2   #include <vector>
3
4   // Классы всех возможных родов войск
5   class Infantryman
6   {
7    public:
8     void info() {
9       cout << "Infantryman" << endl;
10    }
11  };
12
13  class Archer
14  {
15   public:
16    void info() {
17      cout << "Archer" << endl;
18    }
19  };
20
21  class Horseman
22  {
23   public:
```

```
24      void info() {
25        cout << "Horseman" << endl;
26      }
27  };
28
29  class Catapult
30  {
31    public:
32      void info() {
33        cout << "Catapult" << endl;
34      }
35  };
36
37  class Elephant
38  {
39    public:
40      void info() {
41        cout << "Elephant" << endl;
42      }
43  };
44
45
46  // Класс "Армия", содержащий все типы боевых единиц
47  class Army
48  {
49    public:
50      vector<Infantryman> vi;
```

```
51      vector<Archer>      va;
52      vector<Horseman>    vh;
53      vector<Catapult>    vc;
54      vector<Elephant>    ve;
55      void info() {
56        int i;
57        for(i=0; i<vi.size(); ++i)  vi[i].info();
58        for(i=0; i<va.size(); ++i)  va[i].info();
59        for(i=0; i<vh.size(); ++i)  vh[i].info();
60        for(i=0; i<vc.size(); ++i)  vc[i].info();
61        for(i=0; i<ve.size(); ++i)  ve[i].info();
62      }
63  };
64
65
66  // Базовый класс ArmyBuilder объявляет интерфейс для поэтапного
67  // построения армии и предусматривает его реализацию по
    умолчанию
68
69
70  class ArmyBuilder
71  {
72    protected:
73      Army* p;
74    public:
75      ArmyBuilder(): p(0) {}
76      virtual ~ArmyBuilder() {}
77      virtual void createArmy() {}
```

```
78      virtual void buildInfantryman() {}
79      virtual void buildArcher() {}
80      virtual void buildHorseman() {}
81      virtual void buildCatapult() {}
82      virtual void buildElephant() {}
83      virtual Army* getArmy() { return p; }
84  };
85
86
87  // Римская армия имеет все типы боевых единиц кроме боевых
88  слонов
89  class RomanArmyBuilder: public ArmyBuilder
90  {
91    public:
92      void createArmy() { p = new Army; }
93      void buildInfantryman() { p->vi.push_back( Infantryman()); }
94      void buildArcher() { p->va.push_back( Archer()); }
95      void buildHorseman() { p->vh.push_back( Horseman()); }
96      void buildCatapult() { p->vc.push_back( Catapult()); }
97  };
98
99
100 // Армия Карфагена имеет все типы боевых единиц кроме
101 катапульт
102 class CarthaginianArmyBuilder: public ArmyBuilder
103 {
104   public:
      void createArmy() { p = new Army; }
```

```
105     void buildInfantryman() { p->vi.push_back( Infantryman()); }
106     void buildArcher() { p->va.push_back( Archer()); }
107     void buildHorseman() { p->vh.push_back( Horseman()); }
108     void buildElephant() { p->ve.push_back( Elephant()); }
109 };
110
111
112 // Класс-распорядитель, поэтапно создающий армию той или иной
    стороны.
113 // Именно здесь определен алгоритм построения армии.
114 class Director
115 {
116   public:
117     Army* createArmy( ArmyBuilder & builder )
118     {
119         builder.createArmy();
120         builder.buildInfantryman();
121         builder.buildArcher();
122         builder.buildHorseman();
123         builder.buildCatapult();
124         builder.buildElephant();
125         return( builder.getArmy());
126     }
127 };
128
129
130
131 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:

```
Roman army:
Infantryman
Archer
Horseman
Catapult

Carthaginian army:
Infantryman
Archer
Horseman
Elephant
```

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 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.

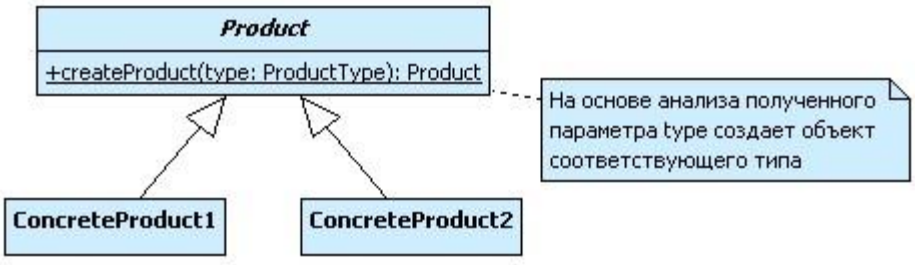
Описание паттерна Factory Method

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:

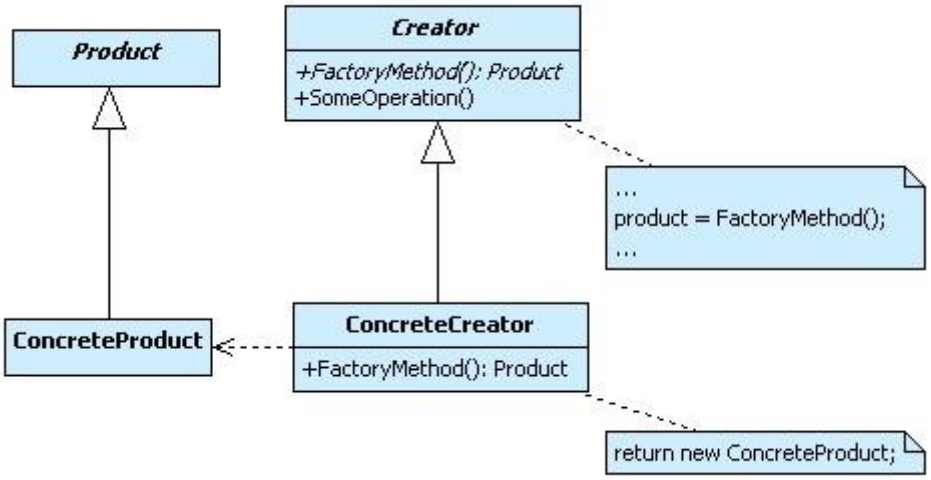
Обобщенный конструктор, когда в том же самом полиморфном базовом классе, от которого наследуют производные классы всех создаваемых в системе типов, определяется статический фабричный метод. В качестве параметра в этот метод должен передаваться идентификатор типа создаваемого объекта.

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

```
1   // #include <iostream>
2   #include <vector>
3
4   enum Warrior_ID { Infantryman_ID=0, Archer_ID, Horseman_ID };
5
6   // Иерархия классов игровых персонажей
7   class Warrior
8   {
9     public:
10      virtual void info() = 0;
11      virtual ~Warrior() {}
12      // Параметризированный статический фабричный метод
13      static Warrior* createWarrior( Warrior_ID id );
```

```
14  };
15
16  class Infantryman: public Warrior
17  {
18    public:
19      void info() {
20        cout << "Infantryman" << endl;
21      }
22  };
23
24  class Archer: public Warrior
25  {
26    public:
27      void info() {
28        cout << "Archer" << endl;
29      }
30  };
31
32  class Horseman: public Warrior
33  {
34    public:
35      void info() {
36        cout << "Horseman" << endl;
37      }
38  };
39
40
```

```
// 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;
```

```
68      v.push_back( Warrior::createWarrior( Infantryman_ID));
69      v.push_back( Warrior::createWarrior( Archer_ID));
70      v.push_back( Warrior::createWarrior( Horseman_ID));
71
72      for(int i=0; i<v.size(); i++)
73          v[i]->info();
74      // ...
    }
```

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

```
1   //
2   #include <iostream>
3   #include <vector>
4
5   // Иерархия классов игровых персонажей
6   class Warrior
7   {
```

```
8     public:
9       virtual void info() = 0;
10      virtual ~Warrior() {}
11  };
12
13  class Infantryman: public Warrior
14  {
15    public:
16      void info() {
17        cout << "Infantryman" << endl;
18      };
19  };
20
21  class Archer: public Warrior
22  {
23    public:
24      void info() {
25        cout << "Archer" << endl;
26      };
27  };
28
29  class Horseman: public Warrior
30  {
31    public:
32      void info() {
33        cout << "Horseman" << endl;
34      };
35  };
36
37
38  // Фабрики объектов
39  class Factory
40  {
41    public:
42      virtual Warrior* createWarrior() = 0;
43      virtual ~Factory() {}
44  };
45
46  class InfantryFactory: public Factory
```

```
47  {
48    public:
49      Warrior* createWarrior() {
50        return new Infantryman;
51      }
52  };
53
54  class ArchersFactory: public Factory
55  {
56    public:
57      Warrior* createWarrior() {
58        return new Archer;
59      }
60  };
61
62  class CavalryFactory: public Factory
63  {
64    public:
65      Warrior* createWarrior() {
66        return new Horseman;
67      }
68  };
69
70
71  // Создание объектов при помощи фабрик объектов
72  int main()
73  {
74      InfantryFactory* infantry_factory = new InfantryFactory;
75      ArchersFactory*  archers_factory  = new ArchersFactory ;
76      CavalryFactory*  cavalry_factory  = new CavalryFactory ;
77
78      vector<Warrior*> v;
79      v.push_back( infantry_factory->createWarrior());
80      v.push_back( archers_factory->createWarrior());
81      v.push_back( cavalry_factory->createWarrior());
82
83      for(int i=0; i<v.size(); i++)
84          v[i]->info();
85      // ...
```

```
86  }
```

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 Отложенная инициализация

Отложенная (ленивая) инициализация (англ. *Lazy initialization*). Приём в программировании, когда некоторая ресурсоёмкая операция (создание объекта, вычисление значения) выполняется непосредственно перед тем, как будет использован её результат. Таким образом, инициализация выполняется «по требованию», а не заблаговременно. Аналогичная идея находит применение в самых разных областях: например, компиляция «на лету» и логистическая концепция «Точно в срок».

Частный случай ленивой инициализации — создание объекта в момент обращения к нему — является одним из порождающих шаблонов

проектирования. Как правило, он используется в сочетании с такими шаблонами как Фабричный метод, Одиночка и Заместитель.

Достоинства

- Инициализация выполняется только в тех случаях, когда она действительно необходима;
- ускоряется начальная инициализация.

Недостатки

- Невозможно явным образом задать порядок инициализации объектов;
- возникает задержка при обращении к объекту.

Примеры реализации

Java

```
public class SomeObject {
        static SomeObject singleInstance = null;
        String msg;

        private SomeObject() {
          msg = "Instance of SomeObject have been created.";
        }

        public String toString() {
                return msg;
        }

        // Метод возвращает экземпляр SomeObject, при этом он
        // создаёт его, если тот ещё не существует
        public static SomeObject getObject() {
                if (singleInstance == null)
                        singleInstance = new SomeObject();

                return singleInstance;
        }
}

public class Main {

        private static long totalMemory;
```

```
    // Этот флаг определяет, было ли выполнено вычисление
    private static boolean isCalc = false;

    private static long getTotalMemory() {
        // Определяем значение totalMemory только при первом вызове
        if (!isCalc) {
            totalMemory = Runtime.getRuntime().totalMemory() / 1024;
            isCalc = true;
        }

        return totalMemory;
    }

    public static void main(String[] args) {

        // Здесь значение будет вычислено
        System.out.println("Total amount of memory: " + getTotalMemory() + " KB");

        // Здесь будет использовано значение, сохранённое в переменной
        if (getTotalMemory() > 1024) {
            System.out.println(SomeObject.getObject());
        }
    }
```

Паттерн Мультитон

Шаблон "Пул одиночек" позволяет создать определенное число своих экземпляров и предоставляет точку доступа для работы с ними. При этом каждый экземпляр связан с уникальным идентификатором.

Пул одиночек может использоваться в случае, когда необходимо предоставить доступ к определенным данным из различных блоков приложения. Другой случай – взаимодействие с аппаратным обеспечением через экземпляры одного и того же класса. Например, обмен данными с сетью контроллеров, опрос группы серверов или рабочих станций в сети. Все эти примеры

объединяет одно: число экземпляров класса может (и даже должно) быть ограничено и они глобальные для всего приложения.

Данный шаблон можно рассматривать как объединение идей Одиночки и Пула объектов. Исходя их этого можно определить его свойства:

1. Шаблон может использоваться как с жестко заданным списком экземпляров, так и с созданием по требованию.
2. Если список фиксированный, то возможно создание всех экземпляров при старте программы или обращению к любому из них.
3. Возможны два варианта реакции на запрос экземпляра с неизвестным идентификатором: отказ или создание нового.
4. Минусом шаблона является возможность появления большого числа зависимых от него частей приложения. Однако, как и в случае с Одиночкой, это можно смягчить используя Внедрение зависимостей (Dependency injection).

Схожие шаблоны и их отличия

Пул одиночек	**Одиночка**	**Пул объектов**
Создает и содержит заданное число экземпляров своего класса. Обеспечивает их идентификацию и точку доступа к ним.	Создает единственный экземпляр своего класса и обеспечивает доступ к нему.	Содержит и содержит определенное число экземпляров заданного класса, не зная ничего о их сути.
Контролирует количество экземпляров.	Гарантирует существование единственного экземпляра.	Не огран��чивает создание других экземпляров заданного класса вне шаблона и ничего не знает о них.

Реализация шаблона в общем виде

- определяем параметры использования шаблона:
  - условие, ограничивающее общее число экземпляров;
  - реакцию на запрос с неизвестным идентификатором;

    - создаются ли экземпляры сразу или только по запросу;
- объявляем только закрытый конструктор, чтобы запретить создание экземпляров извне;
- создаем закрытый контейнер для размещения экземпляров;
- предоставляем доступ к нему через метод;
- клиентский код использует этот метод для получения нужного экземпляра класса.

Идентификаторами будут выступать значения типа *string*. Ограничений по списку экземпляров не будет. Это значит, что на любой запрос будет возвращен существующий объект или создан новый. В качестве контейнера объектов возьмем *ConcurrentDictionary*, входящий в .NET 4. От обычного *Dictionary* этот вариант отличается потокобезопасностью.

Давайте посмотрим исходный код:

```
?
1     /// <summary>Thread-safe .NET4 multiton.</summary>
2     public sealed class Multiton
3     {
4         /// <summary>Container for multiton instances.</summary>
5         private static readonly ConcurrentDictionary<string, Multiton> _instances
6             = new ConcurrentDictionary<string, Multiton>();
7
8         /// <summary>Initializes a new instance of the
9         /// <see cref="Multiton&lt;TKey&gt;"/> class.</summary>
10        /// <param name="key">The key of the instance.</param>
11        private Multiton(string key) { /* SKIPPED */ }
12
13        /// <summary>Gets the instance associated with the specified key .</summary>
14        /// <param name="key">The key of the instance to get.</param>
15        /// <returns>The instance for the key.</returns>
16        public static Multiton GetInstance(string key)
17        {
18            return Multiton._instances.GetOrAdd(key, (x) => new Multiton(x));
19        }
20    }
```

Все очень просто:

- экземпляры класса хранятся в контейнере *_instances*;

- создать самостоятельно экземпляр не возможно, т.к. конструктор закрыт;
- на конструктор возложена обязанность создавать экземпляры в зависимости от указанного идентификатора;
- метод *GetInstance()* возвращает экземпляр по указанному ключу. При этом вызываемый им метод *GetOrAdd()* проверяет есть ли готовый объект в контейнере и, если такой не найден, создает новый.

Остается добавить в класс нужную функциональность и использовать его в деле.

```
?
1    Multiton obj1 = Multiton.GetInstance("instance-id-1");
2    obj1.DoSomething();
```

Но что если нужно контролировать по каким идентификаторам создаются экземпляры? В данном варианте это возможно сделать через исключения в конструкторе.

Паттерн Объектный пул

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

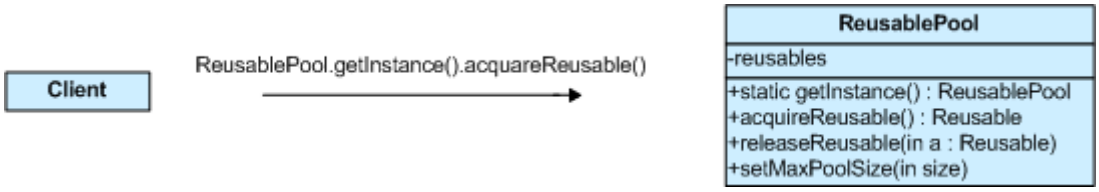
Процессы запрашивают объекты из пула объектов. Когда эти объекты больше не нужны, они возвращаются в пул для дальнейшего повторного использования.

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 Получение ресурса есть инициализация

Получение ресурса есть инициализация (шаблон проектирования) - программная идиома объектно-ориентированного программирования, смысл которой заключается в том, что получение некоторого ресурса совмещается с инициализацией, а освобождение — с уничтожением объекта.

Получение доступа к ресурсу происходит в конструкторе, а освобождение в деструкторе. Поскольку деструктор автоматической переменной вызывается при выходе её из области видимости, то ресурс гарантированно освобождается при уничтожении переменной. Это справедливо и в ситуациях, в которых возникают исключения. Это делает RAII ключевой концепцией для написания безопасного при исключениях кода.

Применения

Эта концепция может использоваться для любых разделяемых объектов или ресурсов:

- для выделения памяти,
- для открытия файлов или устройств,
- для мьютексов или критических секций и т. д.

Важный случай использования RAII — «умные указатели»: классы, инкапсулирующие владение памятью. Например, в стандартной библиотеке шаблонов языка C++ для этой цели существует класс auto_ptr.

```
#include <cstdio>

#include <stdexcept>

class file {

public:

file( const char* filename ) : m_file_handle(std::fopen(filename, "w+"))

{

if( !m_file_handle )

throw std::runtime_error("file open failure") ;

}

~file()

{

if( std::fclose(m_file_handle) != 0 )
```

```
{
// fclose() может вернуть ошибку при записи на диск последних изменений
}
}
void write( const char* str )
{
if( std::fputs(str, m_file_handle) == EOF )
throw std::runtime_error("file write failure") ;
}
private:
std::FILE* m_file_handle ;
// Копирование и присваивание не реализовано. Предотвратим их использование,
// объявив соответствующие методы закрытыми.
file( const file & ) ;
file & operator=( const file & ) ;
};
// пример использования этого класса
void example_usage() {
// открываем файл (захватываем ресурс)
file logfile("logfile.txt");
logfile.write("hello logfile!") ;
// продолжаем использовать logfile...
```

// Можно возбуждать исключения или выходить из функции не беспокоясь о закрытии файла;

// он будет закрыт автоматически когда переменная logfile выйдет из области видимости.

}

Суть идиомы RAII в том, что класс инкапсулирует владение (захват и освобождение) некоторого ресурса — например, открытого файлового дескриптора. Когда объекты-экземпляры такого класса являются автоматическими переменными, гарантируется, что когда они выйдут из области видимости, будет вызван их деструктор — а значит, ресурс будет освобождён. В данном примере файл будет закрыт корректно, даже если вызов std::fopen() вернёт ошибку и будет возбуждено исключение. Более того, если конструктор класса file завершился корректно, это гарантирует то, что файл действительно открыт. В случае ошибки при открытии файла конструктор возбуждает исключение.

При помощи RAII и автоматических переменных можно просто управлять владением нескольких ресурсов. Порядок вызова деструкторов является обратным порядку вызова конструкторов; деструктор вызывается только если объект был полностью создан (то есть если конструктор не возбудил исключение).

Использование RAII упрощает код и помогает обеспечить корректность работы программы.

Возможна реализация без исключений (например, это необходимо во встраиваемых приложениях). В этом случае используется конструктор по умолчанию, обнуляющий file handler, а для открытия файла используется отдельный метод типа bool FileOpen(const char *). Смысл использования класса сохраняется, особенно если точек выхода из метода, где создаётся объект класса, несколько. Естественно, в этом случае в деструкторе проверяется необходимость закрытия файла.

Паттерн Прототип

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.

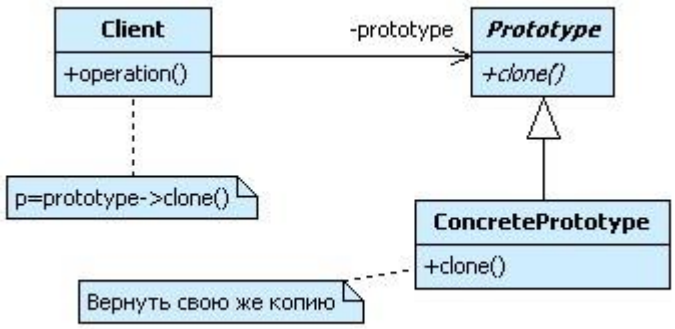
> Паттерн Factory Method также делает систему независимой от типов порождаемых объектов, но для этого он вводит параллельную иерархию классов: для каждого типа создаваемого объекта должен присутствовать соответствующий класс-фабрика, что может быть нежелательно. Паттерн Prototype лишен этого недостатка.

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

```
1       #include <iostream>
```

```
2  #include <vector>
3  #include <map>
4
5  // Идентификаторы всех родов войск
6  enum Warrior_ID { Infantryman_ID, Archer_ID, Horseman_ID };
7
8  class Warrior;  // Опережающее объявление
9  typedef map<Warrior_ID, Warrior*> Registry;
10
11 // Реестр прототипов определен в виде Singleton Мэйерса
12 Registry& getRegistry()
13 {
14   static Registry _instance;
15   return _instance;
16 }
17
18 // Единственное назначение этого класса - помощь в выборе
19 нужного
20 // конструктора при создании прототипов
21 class Dummy { };
22
23 // Полиморфный базовый класс. Здесь также определен
24 статический
25 // обобщенный конструктор для создания боевых единиц всех
26 родов войск
27 class Warrior
28 {
   public:
```

```
29      virtual Warrior* clone() = 0;
30      virtual void info() = 0;
31      virtual ~Warrior() {}
32      // Параметризированный статический метод для создания
33  воинов
34      // всех родов войск
35      static Warrior* createWarrior( Warrior_ID id ) {
36        Registry& r = getRegistry();
37        if (r.find(id) != r.end())
38          return r[id]->clone();
39        return 0;
40      }
41    protected:
42      // Добавление прототипа в множество прототипов
43      static void addPrototype( Warrior_ID id, Warrior * prototype ) {
44        Registry& r = getRegistry();
45        r[id] = prototype;
46      }
47      // Удаление прототипа из множества прототипов
48      static void removePrototype( Warrior_ID id ) {
49        Registry& r = getRegistry();
50        r.erase( r.find( id));
51      }
52  };
53
54
55  // В производных классах различных родов войск в виде
    статических
```

```
56  // членов-данных определяются соответствующие прототипы
57  class Infantryman: public Warrior
58  {
59    public:
60      Warrior* clone() {
61        return new Infantryman( *this);
62      }
63      void info() {
64        cout << "Infantryman" << endl;
65      }
66    private:
67      Infantryman( Dummy ) {
68        Warrior::addPrototype( Infantryman_ID, this);
69      }
70      Infantryman() {}
71      static Infantryman prototype;
72  };
73
74  class Archer: public Warrior
75  {
76    public:
77      Warrior* clone() {
78        return new Archer( *this);
79      }
80      void info() {
81        cout << "Archer" << endl;
82      }
```

```
83      private:
84        Archer(Dummy) {
85          addPrototype( Archer_ID, this);
86        }
87        Archer() {}
88        static Archer prototype;
89    };
90
91    class Horseman: public Warrior
92    {
93      public:
94        Warrior* clone() {
95          return new Horseman( *this);
96        }
97        void info() {
98          cout << "Horseman" << endl;
99        }
100     private:
101       Horseman(Dummy) {
102         addPrototype( Horseman_ID, this);
103       }
104       Horseman() {}
105       static Horseman prototype;
106   };
107
108
109   Infantryman Infantryman::prototype = Infantryman( Dummy());
```

```
110  Archer Archer::prototype = Archer( Dummy());
111  Horseman Horseman::prototype = Horseman( Dummy());
112
113
114  int main()
115  {
116    vector<Warrior*> v;
117    v.push_back( Warrior::createWarrior( Infantryman_ID));
118    v.push_back( Warrior::createWarrior( Archer_ID));
119    v.push_back( Warrior::createWarrior( Horseman_ID));
120
121    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

```
1   #include <iostream>
2   #include <vector>
3
4   // Иерархия классов игровых персонажей
5   // Полиморфный базовый класс
6   class Warrior
7   {
8     public:
9       virtual Warrior* clone() = 0;
10      virtual void info() = 0;
11      virtual ~Warrior() {}
12  };
13
14
15  // Производные классы различных родов войск
16  class Infantryman: public Warrior
17  {
18      friend class PrototypeFactory;
19    public:
```

```
20      Warrior* clone() {
21        return new Infantryman( *this);
22      }
23      void info() {
24        cout << "Infantryman" << endl;
25      }
26    private:
27      Infantryman() {}
28  };
29
30  class Archer: public Warrior
31  {
32      friend class PrototypeFactory;
33    public:
34      Warrior* clone() {
35        return new Archer( *this);
36      }
37      void info() {
38        cout << "Archer" << endl;
39      }
40    private:
41      Archer() {}
42  };
43
44  class Horseman: public Warrior
45  {
46      friend class PrototypeFactory;
```

```
47    public:
48      Warrior* clone() {
49        return new Horseman( *this);
50      }
51      void info() {
52        cout << "Horseman" << endl;
53      }
54    private:
55      Horseman() {}
56  };
57
58
59  // Фабрика для создания боевых единиц всех родов войск
60  class PrototypeFactory
61  {
62    public:
63      Warrior* createInfantrman() {
64        static Infantryman prototype;
65        return prototype.clone();
66      }
67      Warrior* createArcher() {
68        static Archer prototype;
69        return prototype.clone();
70      }
71      Warrior* createHorseman() {
72        static Horseman prototype;
73        return prototype.clone();
```

```
74      }
75  };
76
77
78  int main()
79  {
80    PrototypeFactory factory;
81    vector<Warrior*> v;
82    v.push_back( factory.createInfantrman());
83    v.push_back( factory.createArcher());
84    v.push_back( factory.createHorseman());
85
86    for(int i=0; i<v.size(); i++)
87      v[i]->info();
88      // ...
89  }
```

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 Одиночка

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.

```
1   // Singleton.h
2   class Singleton
3   {
4     private:
5       static Singleton * p_instance;
6       // Конструкторы и оператор присваивания недоступны клиентам
7       Singleton() {}
8       Singleton( const Singleton& );
9       Singleton& operator=( Singleton& );
10    public:
11      static Singleton * getInstance() {
12          if(!p_instance)
13              p_instance = new Singleton();
14          return p_instance;
15      }
16  };
17
18  // Singleton.cpp
```

```
19  #include "Singleton.h"
20
21  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

```
1   // Singleton.h
2   class Singleton
3   {
4   private:
5       Singleton() {}
6       Singleton( const Singleton&);
7       Singleton& operator=( Singleton& );
8   public:
9       static Singleton& getInstance() {
10          static Singleton  instance;
11          return instance;
12      }
13  };
```

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
31  Singleton * Singleton::p_instance = 0;
32  SingletonDestroyer Singleton::destroyer;
33
34  SingletonDestroyer::~SingletonDestroyer() {
35      delete p_instance;
36  }
37  void SingletonDestroyer::initialize( Singleton* p ) {
38      p_instance = p;
39  }
40  Singleton& 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.

```
1   // Singleton.h
2   class Singleton1
3   {
```

```
4     private:
5       Singleton1() { }
6       Singleton1( const Singleton1& );
7       Singleton1& operator=( Singleton1& );
8     public:
9       static Singleton1& getInstance() {
10          static Singleton1 instance;
11          return instance;
12      }
13  };
14
15  class Singleton2
16  {
17    private:
18      Singleton2( Singleton1& instance): s1( instance) { }
19      Singleton2( const Singleton2& );
20      Singleton2& operator=( Singleton2& );
21      Singleton1& s1;
22    public:
23      static Singleton2& getInstance() {
24          static Singleton2 instance( Singleton1::getInstance());
25          return instance;
26      }
27  };
28
29  // main.cpp
30  #include "Singleton.h"
```

```
31
32  int main()
33  {
34      Singleton2& s = Singleton2::getInstance();
35      return 0;
36  }
```

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.