# 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](http://cpp-reference.ru/patterns/creational-patterns/)).
* Необходимо создавать объекты, точные классы которых становятся известными уже на стадии выполнения программы.

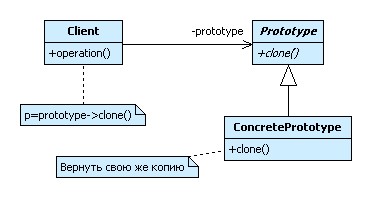
[Паттерн Factory Method](http://cpp-reference.ru/patterns/creational-patterns/factory-method/) также делает систему независимой от типов порождаемых объектов, но для этого он вводит параллельную иерархию классов: для каждого типа создаваемого объекта должен присутствовать соответствующий класс-фабрика, 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 С++ 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](http://cpp-reference.ru/patterns/creational-patterns/). Для упрощения демонстрационного кода будем создавать военные персонажи для некой абстрактной армии без учета особенностей воюющих сторон.

Также как и для [паттерна Factory Method](http://cpp-reference.ru/patterns/creational-patterns/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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112 | #include <iostream>  #include <vector>  #include <map>    // Идентификаторы всех родов войск  enum Warrior\_ID { Infantryman\_ID, Archer\_ID, Horseman\_ID };    class Warrior;  // Опережающее объявление  typedef map<Warrior\_ID, Warrior\*> Registry;    // Реестр прототипов определен в виде Singleton Мэйерса  Registry& getRegistry()  {    static Registry \_instance;    return \_instance;  }    // Единственное назначение этого класса - помощь в выборе нужного  // конструктора при создании прототипов  class Dummy { };    // Полиморфный базовый класс. Здесь также определен статический  // обобщенный конструктор для создания боевых единиц всех родов войск  class Warrior  {    public:      virtual Warrior\* clone() = 0;      virtual void info() = 0;      virtual ~Warrior() {}      // Параметризированный статический метод для создания воинов      // всех родов войск      static Warrior\* createWarrior( Warrior\_ID id ) {        Registry& r = getRegistry();        if (r.find(id) != r.end())          return r[id]->clone();        return 0;      }    protected:      // Добавление прототипа в множество прототипов      static void addPrototype( Warrior\_ID id, Warrior \* prototype ) {        Registry& r = getRegistry();        r[id] = prototype;      }      // Удаление прототипа из множества прототипов      static void removePrototype( Warrior\_ID id ) {        Registry& r = getRegistry();        r.erase( r.find( id));      }  };      // В производных классах различных родов войск в виде статических  // членов-данных определяются соответствующие прототипы  class Infantryman: public Warrior  {    public:      Warrior\* clone() {        return new Infantryman( \*this);      }      void info() {        cout << "Infantryman" << endl;      }    private:      Infantryman( Dummy ) {        Warrior::addPrototype( Infantryman\_ID, this);      }      Infantryman() {}      static Infantryman prototype;  };    class Archer: public Warrior  {    public:      Warrior\* clone() {        return new Archer( \*this);      }      void info() {        cout << "Archer" << endl;      }    private:      Archer(Dummy) {        addPrototype( Archer\_ID, this);      }      Archer() {}      static Archer prototype;  };    class Horseman: public Warrior  {    public:      Warrior\* clone() {        return new Horseman( \*this);      }      void info() {        cout << "Horseman" << endl;      }    private:      Horseman(Dummy) {        addPrototype( Horseman\_ID, this);      }      Horseman() {}      static Horseman prototype;  };      Infantryman Infantryman::prototype = Infantryman( Dummy());  Archer Archer::prototype = Archer( Dummy());  Horseman Horseman::prototype = Horseman( Dummy());      int main()  {    vector<Warrior\*> v;    v.push\_back( Warrior::createWarrior( Infantryman\_ID));    v.push\_back( Warrior::createWarrior( Archer\_ID));    v.push\_back( Warrior::createWarrior( Horseman\_ID));      for(int i=0; i<v.size(); i++)      v[i]->info();    // ...  } |

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

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

Для приведенной реализации паттерна Prototype можно отметить следующие особенности:

* Создавать новых воинов можно только при помощи обобщенного конструктора. Их непосредственное создание невозможно, так как соответствующие конструкторы объявлены со спецификатором доступа private.
* Отсутствует недостаток реализации на базе обобщенного конструктора для [паттерна Factory Method](http://cpp-reference.ru/patterns/creational-patterns/factory-method/), а именно базовый класс Warrior ничего не знает о своих подклассах.

### Реализация паттерна Prototype с помощью выделенного класса-фабрики

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82 | #include <iostream>  #include <vector>    // Иерархия классов игровых персонажей  // Полиморфный базовый класс  class Warrior  {    public:      virtual Warrior\* clone() = 0;      virtual void info() = 0;      virtual ~Warrior() {}  };      // Производные классы различных родов войск  class Infantryman: public Warrior  {      friend class PrototypeFactory;    public:      Warrior\* clone() {        return new Infantryman( \*this);      }      void info() {        cout << "Infantryman" << endl;      }    private:      Infantryman() {}  };    class Archer: public Warrior  {      friend class PrototypeFactory;    public:      Warrior\* clone() {        return new Archer( \*this);      }      void info() {        cout << "Archer" << endl;      }    private:      Archer() {}  };    class Horseman: public Warrior  {      friend class PrototypeFactory;    public:      Warrior\* clone() {        return new Horseman( \*this);      }      void info() {        cout << "Horseman" << endl;      }    private:      Horseman() {}  };      // Фабрика для создания боевых единиц всех родов войск  class PrototypeFactory  {    public:      Warrior\* createInfantrman() {        static Infantryman prototype;        return prototype.clone();      }      Warrior\* createArcher() {        static Archer prototype;        return prototype.clone();      }      Warrior\* createHorseman() {        static Horseman prototype;        return prototype.clone();      }  };      int main()  {    PrototypeFactory factory;    vector<Warrior\*> v;    v.push\_back( factory.createInfantrman());    v.push\_back( factory.createArcher());    v.push\_back( factory.createHorseman());      for(int i=0; i<v.size(); i++)      v[i]->info();      // ...  } |

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

## Результаты применения паттерна Prototype

### Достоинства паттерна Prototype

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

### Недостатки паттерна Prototype

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

**http://cpp-reference.ru/patterns/creational-patterns/prototype/**