**Практическая работа № 3**

**Шаблоны проектирования**

***Цель работы***

Цель работы заключается в знакомстве с основными шаблонами проектирования, научиться применять их при проектировании и разработке ПО. В работе приводятся примеры реализации паттернов проектирования при решении практических задач в сфере создания игр.

**Краткие теоретические сведения**

**Шаблон проектирования**или паттерн (англ. design pattern) в разработке программного обеспечения — повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код; это лишь пример решения задачи, который можно использовать в различных ситуациях. Объектно­ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие именно конечные классы или объекты приложения будут использоваться.

Сообразное использование паттернов проектирования дает разработчику ряд неоспоримых преимуществ. Приведем некоторые из них. Модель системы, построенная в терминах паттернов проектирования, фактически является структурированным выделением тех элементов и связей, которые значимы при решении поставленной задачи. Помимо этого, модель, построенная с использованием паттернов проектирования, более проста и наглядна в изучении, чем стандартная модель. Тем не менее, несмотря на простоту и наглядность, она позволяет глубоко и всесторонне проработать архитектуру разрабатываемой системы с использованием специального языка. Применение паттернов проектирования повышает устойчивость системы к изменению требований и упрощает неизбежную последующую доработку системы. Кроме того, трудно переоценить роль использования паттернов при интеграции информационных систем организации. Также следует упомянуть, что совокупность паттернов проектирования, по сути, представляет собой единый словарь проектирования, который, будучи унифицированным средством, незаменим для общения разработчиков друг другом.

Но самое главное ­ любой шаблон проектирования может стать палкой о двух концах: если он будет применен не к месту, это может обернуться катастрофой и создать вам много проблем в последующем. В то же время, реализованный в нужном месте, в нужное время, он может стать для вас настоящим спасителем.

Есть три основных вида шаблонов проектирования:

●структурные;

●порождающие;

●поведенческие.

**Структурные шаблоны**определяют различные сложные структуры, которые изменяют интерфейс уже существующих объектов или его реализацию, позволяя облегчить разработку и оптимизировать программу.

**Порождающие шаблоны**­ шаблоны проектирования, которые абстрагируют процесс инстанцирования. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять инстанцируемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту.

**Поведенческие шаблоны**определяют взаимодействие между объектами, увеличивая таким образом его гибкость.

Каждый из шаблонов предназначен для решения своего круга практических задач:

* **Порождающие шаблоны**, предназначенные для создания новых объектов в системе;
* **Структурные шаблоны**, решающие задачи компоновки системы на основе классов и объектов;
* **Шаблоны поведения**, предназначенные для распределения обязанностей между объектами в системе.

Далее рассмотрим основные шаблоны различных видов.

**Структурные шаблоны**

**Адаптер (Adapter)**

**Проблема:**необходимо обеспечить взаимодействие несовместимых интерфейсов или создать единый устойчивый интерфейс для нескольких компонентов с разными интерфейсами.

**Решение:**преобразовать исходный интерфейс компонента к другому виду с помощью промежуточного объекта ­ адаптера, то есть, добавить специальный объект с общим интерфейсом в рамках данного приложения и перенаправить связи от внешних объектов к этому объекту ­ адаптеру.

Класс ***Adapter***приводит интерфейс класса ***Adaptee***в соответствие с интерфейсом класса ***Target***(наследником которого является ***Adapter***). Это позволяет объекту ***Client***использовать объект ***Adaptee***(посредством адаптера ***Adapter***) так, словно он является экземпляром класса ***Target***.

Таким образом ***Client***обращается к интерфейсу ***Target***, реализованному в наследнике ***Adapter***, который перенаправляет обращение к ***Adaptee***.

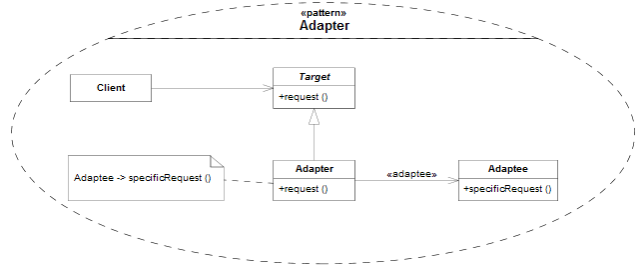


Рисунок 1 - шаблон «Адаптер»

Шаблон Адаптер позволяет включать уже существующие объекты в новые объектные структуры, независимо от различий в их интерфейсах. Этот шаблон позволяет в процессе проектирования не принимать во внимание возможные различия в интерфейсах уже существующих классов. Если есть класс, обладающий требуемыми методами и свойствами (по крайней мере, концептуально), то при необходимости всегда можно воспользоваться шаблоном Адаптер для приведения его интерфейса к нужному виду.

**Фасад (Facade)**

Шаблон “фасад” ­ структурный шаблон проектирования, позволяющий скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.

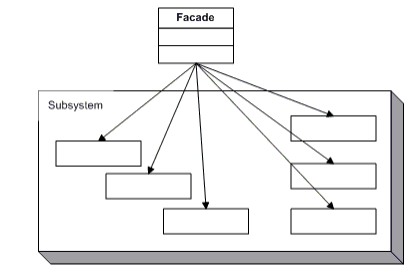


Рисунок 2 - шаблон «Фасад»

Проблема:как обеспечить унифицированный интерфейс с набором разрозненных реализаций или интерфейсов, например, с подсистемой, если нежелательно высокое связывание с этой подсистемой или реализация подсистемы может измениться? Решение:определить одну точку взаимодействия с подсистемой — фасадный объект, обеспечивающий общий интерфейс с подсистемой, и возложить на него обязанность по взаимодействию с её компонентами. Фасад — это внешний объект, обеспечивающий единственную точку входа для служб подсистемы. Реализация других компонентов подсистемы закрыта и не видна внешним компонентам.

**Заместитель (Proxy)**

Заместитель — структурный шаблон проектирования, который предоставляет объект, который контролирует доступ к другому объекту, перехватывая все вызовы. Проблема:необходимо управлять доступом к объекту так, чтобы не создавать громоздкие объекты «по требованию». Решение:создать суррогат громоздкого объекта. «Заместитель» хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту (объект класса «Заместитель» может обращаться к объекту класса «Субъект», если интерфейсы «Реального Субъекта» и «Субъекта» одинаковы). Поскольку интерфейс «Реального Субъекта» идентичен интерфейсу «Субъекта», так, что «Заместителя» можно подставить вместо «Реального Субъекта». «Заместитель» контролирует доступ к «Реальному Субъекту», может отвечать за создание или удаление «Реального Субъекта». «Субъект» определяет общий для «Реального Субъекта» и «Заместителя» интерфейс, так, что «Заместитель» может быть использован везде, где ожидается «Реальный Субъект». При необходимости запросы могут быть переадресованы «Заместителем» «Реальному Субъекту».

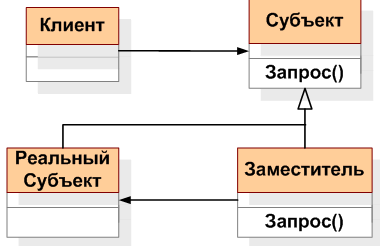


Рисунок 3 - шаблон «Заместитель»

**Декоратор**

Декоратор — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту. Шаблон Декоратор предоставляет гибкую альтернативу практике создания подклассов с целью расширения функциональности.

Задача: объект, который предполагается использовать, выполняет основные функции. Однако может потребоваться добавить к нему некоторую дополнительную функциональность, которая будет выполняться до, после или даже вместо основной функциональности объекта.

Решение: шаблон “декоратор” предусматривает расширение функциональности объекта без определения подклассов.

Класс ConcreteComponent — класс, в который с помощью шаблона Декоратор добавляется новая функциональность. В некоторых случаях базовая функциональность предоставляется классами,производными от класса ConcreteComponent. В подобных случаях класс ConcreteComponent является уже не конкретным, а абстрактным. Абстрактный класс Component определяет интерфейс для использования всех этих классов.

**Поведенческие шаблоны**

**Стратегия (Strategy)**

Стратегия — поведенческий шаблон проектирования, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Это позволяет выбирать алгоритм путем определения соответствующего класса. Шаблон Strategy позволяет менять выбранный алгоритм независимо от объектов­клиентов, которые его используют.

**Проблема:**по типу клиента (или по типу обрабатываемых данных) выбрать подходящий алгоритм, который следует применить. Если используется правило, которое не подвержено изменениям, нет необходимости обращаться к шаблону «стратегия».

**Решение:**отделение процедуры выбора алгоритма от его реализации. Это позволяет сделать выбор на основании контекста.

Класс Strategy определяет, как будут использоваться различные алгоритмы. Конкретные классы ConcreteStrategy реализуют эти различные алгоритмы. Класс Context использует конкретные классы ConcreteStrategy посредством ссылки на конкретный тип абстрактного класса Strategy. Классы Strategy и Context взаимодействуют с целью реализации выбранного алгоритма (в некоторых случаях классу Strategy требуется посылать запросы классу Context). Класс Context пересылает классу Strategy запрос, поступивший от его класса­клиента.

Наблюдатель

Наблюдатель — поведенческий шаблон проектирования. Создает механизм у класса, который позволяет получать экземпляру объекта этого класса оповещения от других объектов об изменении их состояния, тем самым наблюдая за ними.

Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

При реализации шаблона «наблюдатель» обычно используются следующие классы:

●**Observable**— интерфейс, определяющий методы для добавления, удаления и оповещения наблюдателей;

●**Observer**— интерфейс, с помощью которого наблюдатель получает оповещение;

●**ConcreteObservable**— конкретный класс, который реализует интерфейс Observable;

●**ConcreteObserver**— конкретный класс, который реализует интерфейс Observer.

Шаблон «наблюдатель» применяется в тех случаях, когда система обладает следующими свойствами:

●существует, как минимум, один объект, рассылающий сообщения;

●имеется не менее одного получателя сообщений, причём их количество и состав могут изменяться во время работы приложения;

●нет надобности очень сильно связывать взаимодействующие объекты, что полезно для повторного использования.

Данный шаблон часто применяют в ситуациях, в которых отправителя сообщений не интересует, что делают получатели с предоставленной им информацией.

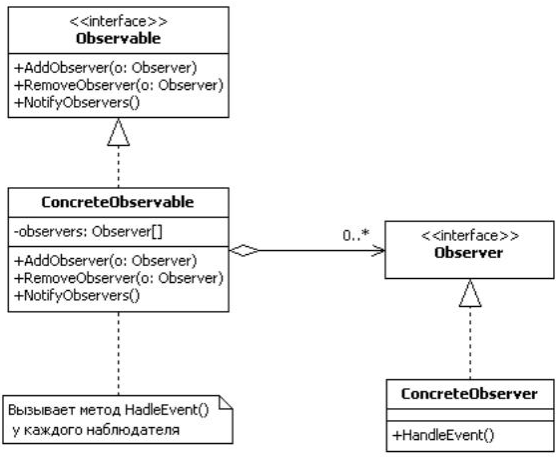


Рисунок 6 - Наблюдатель (Observer)

**Команда (Command)**

Команда — поведенческий шаблон проектирования, используемый при объектно­ориентированном программировании, представляющий действие. Объект команды заключает в себе само действие и его параметры.

Паттерн обеспечивает обработку команды в виде объекта, что позволяет сохранять её, передавать в качестве параметра методам, а также возвращать её в виде результата, как и любой другой объект.

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

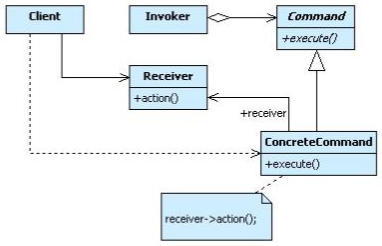


Рисунок 7 - Паттерн Command

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

В паттерне Command может быть до трех участников:

●Клиент, создающий экземпляр командного объекта.

●Инициатор запроса, использующий командный объект.

●Получатель запроса.

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

– функция регистрируется, чтобы быть вызванной позднее.

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

**Порождающие шаблоны**

Порождающие шаблоны 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 шаблонов, рассмотрим на примере типичные проблемы, встающие перед разработчиками при порождении в системе объектов новых типов.

Рассмотрим принципы работы с порождающими шаблонами на примере создания игры, где будет необходимо создавать войска разных типов:

* Пехота;
* Конница;
* Лучники.

Каждый из этих видов войска будет обладать собственными характеристиками. Проектирование архитектуры должно учитывать возможности внесения изменения в иерархию классов, создание новых видов войск, изменение их характеристик.

Простейшим решением будет создание иерархии классов для каждого вида войска:

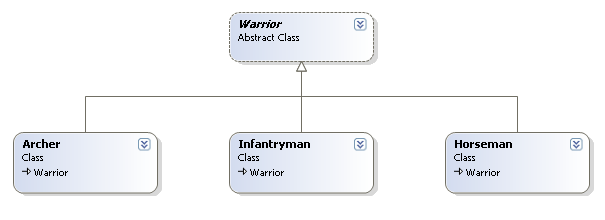


Рисунок 8 - Диаграмма классов для войск

Структура классов выглядит следующим образом:

**Класс Warrior (воин – базовый абстрактный класс для всех видов войск)**

/// <summary>

/// Воин

/// </summary>

public abstract class Warrior

{

/// <summary>

/// Иформация

/// </summary>

public abstract string Info();

}

**Класс Infantryman**

/// <summary>

/// Пехотинец

/// </summary>

public class Infantryman : Warrior

{

/// <summary>

/// Информация о воине

/// </summary>

/// <returns></returns>

public override string Info()

{

return "Infantryman";

}

}

**Класс Archer**

/// <summary>

/// Лучник

/// </summary>

public class Archer : Warrior

{

/// <summary>

/// Информация о воине

/// </summary>

/// <returns></returns>

public override string Info()

{

return "Archer";

}

}

**Класс Horseman**

/// <summary>

/// Всадник

/// </summary>

public class Horseman : Warrior

{

/// <summary>

/// Информация о воине

/// </summary>

/// <returns></returns>

public override string Info()

{

return "Horseman";

}

}

Абстрактный 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 экземпляров классов. С её помощью можно 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.

public enum WarriorTypes

{

Infantryman,

Archer,

Horseman

}

…

/// <summary>

/// Создание воина

/// </summary>

/// <param name="parWattiorType">Тип воина</param>

/// <returns></returns>

public static Warrior CreateWarrior(WarriorTypes parWattiorType)

{

Warrior warrior = null;

switch (parWattiorType)

{

case WarriorTypes.Infantryman:

warrior = new Infantryman();

break;

case WarriorTypes.Archer:

warrior = new Archer();

break;

case WarriorTypes.Horseman:

warrior = new Horseman();

break;

}

return warrior;

}

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( ).

Показав основные проблемы, которые возникают при добавлении новых типов классов, рассмотрим порождающие шаблоны и особенности каждого из них:

* **Шаблон «Фабричный метод» (Factory Method).** В данном шаблоне 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;
* **Шаблон «Абстрактная фабрика» (Abstract Factory).** Использует несколько фабричных методов и предназначен для создания целого семейства или группы взаимосвязанных объектов. Для приведённого выше примера, шаблон может предоставлять методы для создания всех типов воинов;
* **Шаблон «Строитель» (Builder).** Определяет процесс поэтапного конструирования сложного объекта, в результате которого могут получаться разные представления этого объекта;
* **Шаблон «Прототип» (Prototype).** Создаёт новые объекты с помощью прототипов (создавая копии самого себя);
* **Шаблон «Синглетон» (Singleton).** Позволяет создавать только один экземпляр класса.

**Шаблон «Фабричный метод»**

В системе часто требуется создавать объекты самых разных типов. Шаблон «Фабричный метод» (Factory Method) 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. Шаблон «Фабричный метод» 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.

Для того, чтобы система оставалась независимой от различных типов объектов, Шаблон «Фабричный метод» использует механизм полиморфизма - классы всех конечных типов наследуются 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 Шаблон «Фабричный метод» локализует создание объектов конкретных типов в специальном классе-фабрике. Методы этого класса, посредством которых создаются объекты конкретных классов, называются фабричными. Существуют две разновидности Шаблон «Фабричный метод»:

**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.

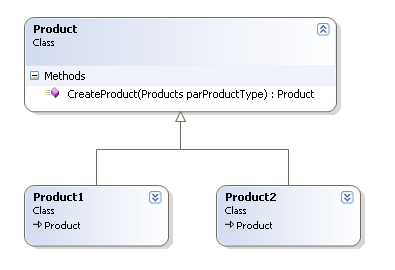


Рисунок 8 - Пример диаграммы классов для фабричного метода с обобщённым конструктором

Исходный код для данной диаграммы выглядит следующим образом:

public enum Products

{

Product1,

Product2

}

public class Product

{

public static Product CreateProduct(Products parProductType)

{

Product product = null;

switch (parProductType)

{

case Products.Product1:

product = new Product1();

break;

case Products.Product2:

product = new Product2();

break;

}

return product;

}

}

public class Product1 : Product

{

}

public class Product2 : Product

{

}

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

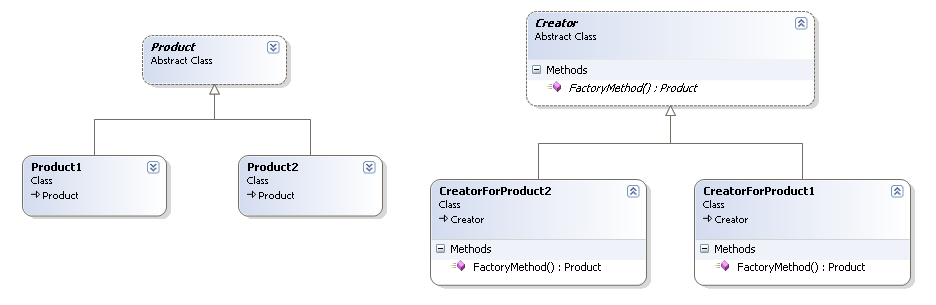


Рисунок 9 Пример классического фабричного метода

Исходный код для данной диаграммы выглядит следующим образом:

public abstract class Creator

{

public abstract Product FactoryMethod();

}

public class CreatorForProduct1 : Creator

{

public override Product FactoryMethod()

{

return new Product1();

}

}

public class CreatorForProduct2 : Creator

{

public override Product FactoryMethod()

{

return new Product2();

}

}

public abstract class Product

{

}

public class Product1 : Product

{

}

public class Product2 : Product

{

}

**Реализация шаблона «Фабричный метод»**

Рассмотрим оба варианта шаблона на примере создания различных типов воинов.

**Пример реализации для обобщённого конструктора**

Диаграмма классов будет выглядеть следующим образом:

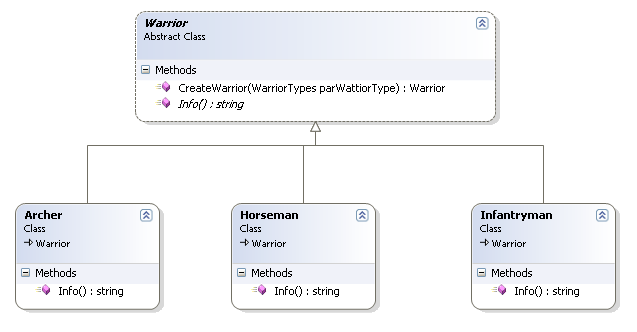


Рисунок 10 Диаграмма классов для реализации создания воинов на базе обобщённого конструктора

Исходный код:

public enum WarriorTypes

{

Infantryman,

Archer,

Horseman

}

public abstract class Warrior

{

public abstract string Info();

public static Warrior CreateWarrior(WarriorTypes parWattiorType)

{

Warrior warrior = null;

switch (parWattiorType)

{

case WarriorTypes.Infantryman:

warrior = new Infantryman();

break;

case WarriorTypes.Archer:

warrior = new Archer();

break;

case WarriorTypes.Horseman:

warrior = new Horseman();

break;

}

return warrior;

}

}

public class Infantryman : Warrior

{

public override string Info()

{

return "Infantryman";

}

}

public class Archer : Warrior

{

public override string Info()

{

return "Archer";

}

}

public class Horseman : Warrior

{

public override string Info()

{

return "Horseman";

}

}

Пример создания армии из трёх воинов будет выглядеть следующим образом:

Warrior[] army = new Warrior[3];

army[0] = Warrior.CreateWarrior(WarriorTypes.Infantryman);

army[1] = Warrior.CreateWarrior(WarriorTypes.Archer);

army[2] = Warrior.CreateWarrior(WarriorTypes.Horseman);

foreach (Warrior elWarrior in army)

elWarrior.Info();

Представленный вариант шаблона «Фабричный метод» 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.

**Пример классической реализации**

Диаграмма классов будет выглядеть следующим образом:

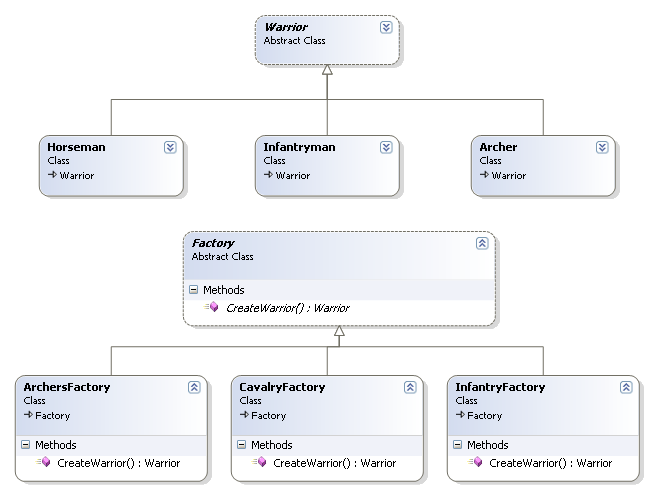


Рисунок 11- Диаграмма классов для классического примера шаблонного метода

Исходный код:

public abstract class Warrior

{

public abstract string Info();

}

public class Infantryman : Warrior

{

public override string Info()

{

return "Infantryman";

}

}

public class Archer : Warrior

{

public override string Info()

{

return "Archer";

}

}

public class Horseman : Warrior

{

public override string Info()

{

return "Horseman";

}

}

public abstract class Factory

{

public abstract Warrior CreateWarrior();

}

public class InfantryFactory : Factory

{

public override Warrior CreateWarrior()

{

return new Infantryman();

}

}

public class ArchersFactory : Factory

{

public override Warrior CreateWarrior()

{

return new Archer();

}

}

public class CavalryFactory : Factory

{

public override Warrior CreateWarrior()

{

return new Horseman();

}

}

Классический вариант шаблона «Фабричный метод» 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().

**Достоинства и недостатки**

**Достоинства:**

* Создает объекты разных типов, позволяя системе оставаться независимой как от самого процесса создания, так и от типов создаваемых объектов.

**Недостатки:**

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

**Шаблон «Абстрактная фабрика»**

Данный шаблон позволяет изменять поведение системы, оперируя создаваемыми объектами, при этом сохраняя интерфейсы. Он позволяет создавать целые группы взаимосвязанных объектов, которые, будучи созданными одной фабрикой, реализуют общее поведение. Шаблон реализуется созданием абстрактного класса Factory, который представляет собой интерфейс для создания компонентов системы (например, для оконного интерфейса он может создавать окна и кнопки). Затем пишутся классы, реализующие этот интерфейс.

Данный шаблон рекомендуется 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;
* Необходимо создавать группы или семейства взаимосвязанных объектов, исключая возможность одновременного использования объектов из разных семейств в одном контексте.

Для решения задачи по созданию семейств взаимосвязанных объектов шаблон «Абстрактная фабрика» 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.

Обобщённая диаграмма классов выглядит следующим образом:

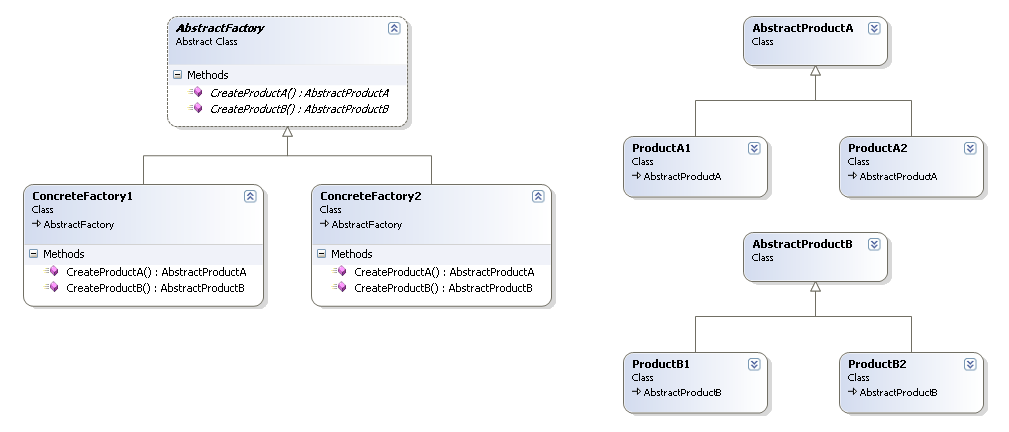


Рисунок 12 - Диаграмма классов для реализации абстрактной фабрики

Исходный код:

public class AbstractProductA

{

}

public class ProductA1 : AbstractProductA

{

}

public class ProductA2 : AbstractProductA

{

}

public class AbstractProductB

{

}

public class ProductB1 : AbstractProductB

{

}

public class ProductB2 : AbstractProductB

{

}

public abstract class AbstractFactory

{

public abstract AbstractProductA CreateProductA();

public abstract AbstractProductB CreateProductB();

}

public class ConcreteFactory1 : AbstractFactory

{

public override AbstractProductA CreateProductA()

{

return new ProductA1();

}

public override AbstractProductB CreateProductB()

{

return new ProductB1();

}

}

public class ConcreteFactory2 : AbstractFactory

{

public override AbstractProductA CreateProductA()

{

return new ProductA2();

}

public override AbstractProductB CreateProductB()

{

return new ProductB2();

}

}

В данном примере есть отдельные иерархии наследования для продуктов «A» и «B». Абстрактный класс фабрики реализует два метода по созданию объектов классов «A» и «B». Конкретные реализации фабрики переопределяют данные методы и создают каждый свой вид продуктов.

**Пример реализации абстрактной фабрики**

Рассмотрим пример реализации абстрактной фабрики для создания различных типов воинов. Сделаем классы создания армий двух разных стран: Рим и Карфаген. Для простоты, будем считать, что у каждой страны есть одинаковые типы воинов.

Диаграмма классов выглядит следующим образом:

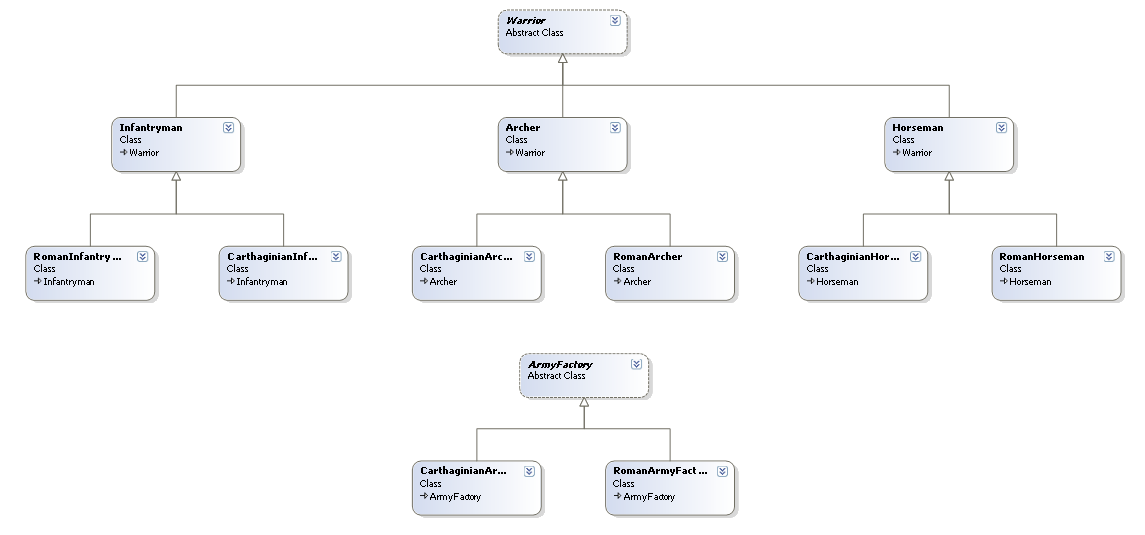


Рисунок 13 - Пример фабрики классов для создания армии из воинов

Исходный код:

public abstract class Warrior

{

public abstract string Info();

}

public class Infantryman : Warrior

{

public override string Info()

{

return "Infantryman";

}

}

public class Archer : Warrior

{

public override string Info()

{

return "Archer";

}

}

public class Horseman : Warrior

{

public override string Info()

{

return "Horseman";

}

}

public class RomanInfantryman : Infantryman

{

public override string Info()

{

return "RomanInfantryman";

}

}

public class RomanArcher : Archer

{

public override string Info()

{

return "RomanArcher";

}

}

public class RomanHorseman : Horseman

{

public override string Info()

{

return "RomanHorseman";

}

}

public class CarthaginianInfantryman : Infantryman

{

public override string Info()

{

return "CarthaginianInfantryman";

}

}

public class CarthaginianArcher : Archer

{

public override string Info()

{

return "CarthaginianArcher";

}

}

public class CarthaginianHorseman : Horseman

{

public override string Info()

{

return "CarthaginianHorseman";

}

}

public abstract class ArmyFactory

{

public abstract Infantryman CreateInfantryman();

public abstract Archer CreateArcher();

public abstract Horseman CreateHorseman();

}

public class RomanArmyFactory : ArmyFactory

{

public override Infantryman CreateInfantryman()

{

return new RomanInfantryman();

}

public override Archer CreateArcher()

{

return new RomanArcher();

}

public override Horseman CreateHorseman()

{

return new RomanHorseman();

}

}

public class CarthaginianArmyFactory : ArmyFactory

{

public override Infantryman CreateInfantryman()

{

return new CarthaginianInfantryman();

}

public override Archer CreateArcher()

{

return new CarthaginianArcher();

}

public override Horseman CreateHorseman()

{

return new CarthaginianHorseman();

}

}

public class Army

{

private ArmyFactory \_armyFactory;

private Warrior[] \_army;

public Army(ArmyFactory parFactory)

{

\_armyFactory = parFactory;

}

public Warrior[] CreateArmy()

{

\_army = new Warrior[3];

\_army[0] = \_armyFactory.CreateInfantryman();

\_army[1] = \_armyFactory.CreateArcher();

\_army[2] = \_armyFactory.CreateHorseman();

return \_army;

}

}

Для каждого типа войска создаётся своя иерархия классов, в которой выделяется отдельные классы для каждого типа войск по странам.

Создаётся базовый класс «ArmyFactory» для создания каждого из типов войск. Для армии каждой страны создаются фабрики – наследники.

Также добавляется класс «Army», который создаёт армию, используя переданную фабрику классов.

Создание армий двух стран выглядит следующим образом:

RomanArmyFactory romanArmyFactory = new RomanArmyFactory();

CarthaginianArmyFactory carthaginianArmyFactory = new CarthaginianArmyFactory();

Army romanArmy = new Army(romanArmyFactory);

Warrior[] romanWarriors = romanArmy.CreateArmy();

Army carthaginianArmy = new Army(carthaginianArmyFactory);

Warrior[] carthaginianWarriors = carthaginianArmy.CreateArmy();

**Достоинства:**

* 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:**

* Трудно добавлять новые типы создаваемых объектов или заменять существующие, так как интерфейс базового класса абстрактной фабрики фиксирован. Например, если для текущего примера нужно будет ввести новый вид военной единицы - осадные орудия, то необходимо 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.

**Шаблон «Строитель»**

Шаблон «Строитель» (Builder) позволяет не только создавать объекты, но и конструировать сложные объекты по заложенному в него алгоритму.

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.

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

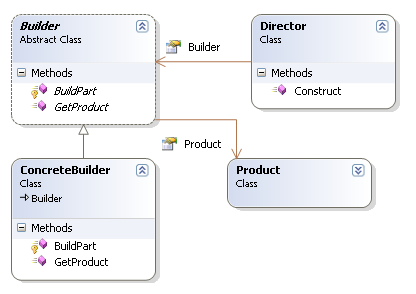


Рисунок 14 - Диаграмма классов для шаблона «Строитель»

Класс Director содержит указатель или ссылку на экземпляр одного из дочерних классов Builder, определяющего соответствующе представление. После этого, класс 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.

Исходный код:

public abstract class Builder

{

protected abstract Product BuildPart();

public Product GetProduct()

{

return BuildPart();

}

}

public class ConcreteBuilder : Builder

{

protected override Product BuildPart()

{

Product product = new Product();

BuildPart1();

BuildPart2();

return product;

}

private void BuildPart1()

{

}

private void BuildPart2()

{

}

}

public class Director

{

Builder \_builder;

public Product Construct()

{

return \_builder.GetProduct();

}

}

public class Product

{

}

**Пример реализации шаблона «Строитель»**

Приведём реализацию шаблона «Строитель» на примере построения армии.

Сделаем классы создания армий двух разных стран: Рим и Карфаген. Для простоты, будем считать, что у каждой страны есть одинаковые типы воинов.

Диаграмма классов выглядит следующим образом:

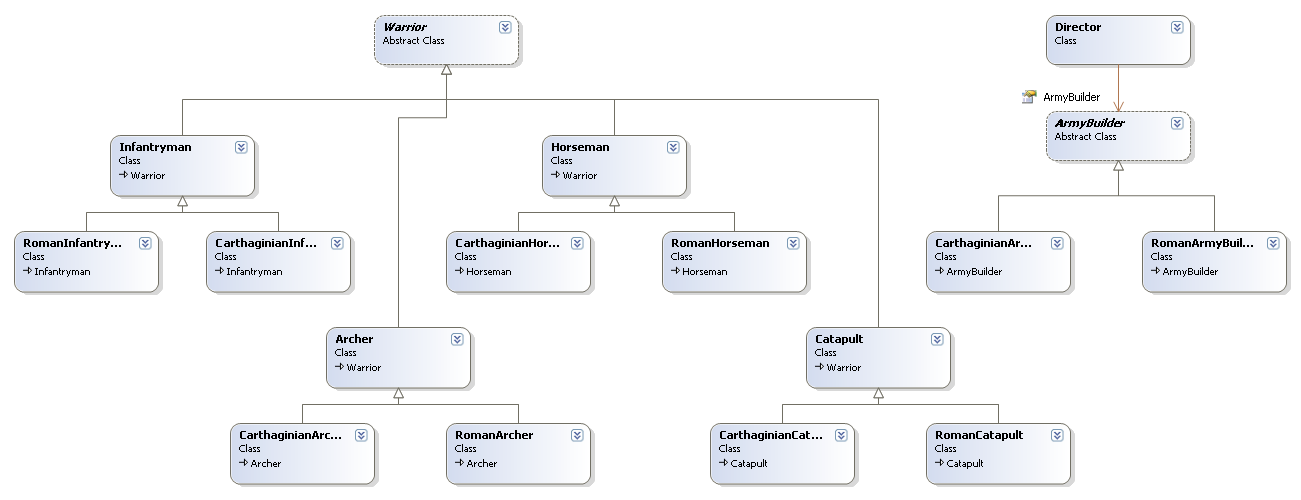


Рисунок 15 - Диаграмма классов реализации создания армии на шаблоне «Строитель»

Исходный код (не показаны классы для типов армий):

public abstract class ArmyBuilder

{

private List<Warrior> \_army;

public void CreateArmy()

{

\_army = new List<Warrior>();

}

public abstract void AddInfantry();

public abstract void AddArcher();

public abstract void AddHorseman();

public abstract void AddCatapult();

public List<Warrior> Army { get { return \_army; } }

}

public class RomanArmyBuilder : ArmyBuilder

{

public override void AddInfantry() { Army.Add(new RomanInfantryman()); }

public override void AddArcher() { Army.Add(new RomanArcher()); }

public override void AddHorseman() { Army.Add(new RomanHorseman()); }

public override void AddCatapult() { Army.Add(new RomanCatapult()); }

}

public class CarthaginianArmyBuilder : ArmyBuilder

{

public override void AddInfantry() { Army.Add(new CarthaginianInfantryman()); }

public override void AddArcher() { Army.Add(new CarthaginianArcher()); }

public override void AddHorseman() { Army.Add(new CarthaginianHorseman()); }

public override void AddCatapult() { Army.Add(new CarthaginianCatapult()); }

}

public class Director

{

public List<Warrior> CreateArmy(ArmyBuilder parArmyBuilder)

{

parArmyBuilder.CreateArmy();

parArmyBuilder.AddInfantry();

parArmyBuilder.AddArcher();

parArmyBuilder.AddHorseman();

parArmyBuilder.AddCatapult();

return parArmyBuilder.Army;

}

}

Если сравнить реализацию построения армии по шаблону «Строитель» с «Абстрактной фабрикой», то видно отличие в том, что в шаблоне «Строитель» скрыта реализация процесса создания армии.

Создание армий двух стран выглядит следующим образом:

Director director = new Director();

RomanArmyBuilder romanArmyBuilder = new RomanArmyBuilder();

CarthaginianArmyBuilder carthaginianArmyBuilder = new CarthaginianArmyBuilder();

List<Warrior> romanArmy = director.CreateArmy(romanArmyBuilder);

List<Warrior> carthaginianArmy = director.CreateArmy(carthaginianArmyBuilder);

**Достоинства:**

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

**Недостатки:**

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

**Шаблон «Прототип»**

Для создания новых объектов шаблон «Прототип» использует клонирование. 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.

Шаблон «Прототип» 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().

Данный шаблон 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;
* Необходимо создавать объекты, точные классы которых становятся известными уже на стадии выполнения программы.

Шаблон «Фабричный метод» 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. «Прототип» лишен этого недостатка.

Диаграмма классов выглядит следующим образом:

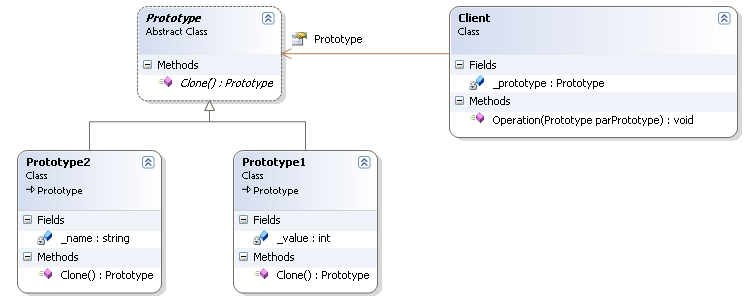


Рисунок 16 - Диаграмма классов для шаблона «Прототип»

Исходный код:

public abstract class Prototype

{

public abstract Prototype Clone();

}

public class Prototype1 : Prototype

{

private int \_value;

public override Prototype Clone()

{

Prototype1 prototype = new Prototype1();

prototype.\_value = this.\_value;

return prototype;

}

}

public class Prototype2 : Prototype

{

private string \_name;

public override Prototype Clone()

{

Prototype2 prototype = new Prototype2();

prototype.\_name = this.\_name;

return prototype;

}

}

public class Client

{

Prototype \_prototype;

public void Operation(Prototype parPrototype)

{

\_prototype = parPrototype.Clone();

}

}

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. 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.

**Пример реализации шаблона «Прототип»**

Приведем реализацию шаблона «Прототип» на примере построения армий.

В примере будут показаны два варианта реализации шаблона:

* 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.
* На базе специально выделенного класса-фабрики.

**Реализация шаблона на базе обобщённого конструктора**

Диаграмма классов:

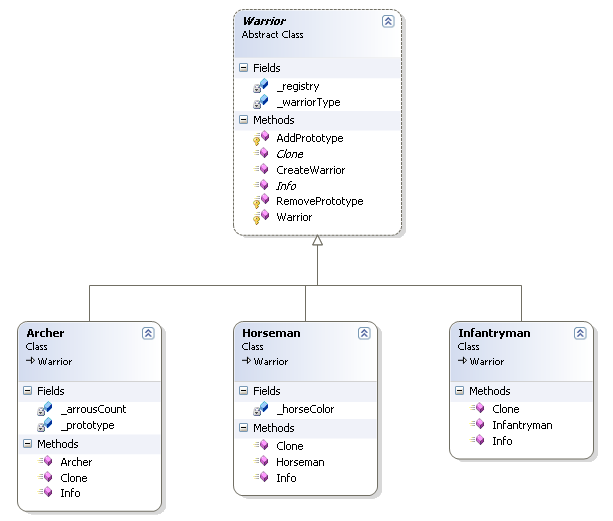


Рисунок 17 - Диаграмма классов для шаблона «Прототип» на базе обобщённого конструктора

Исходный код:

public enum WarriorTypes

{

Infantryman,

Archer,

Horseman

}

public abstract class Warrior

{

private WarriorTypes \_warriorType;

private static Hashtable \_registry = new Hashtable();

protected Warrior(WarriorTypes parWarriorType)

{

\_warriorType = parWarriorType;

}

public abstract Warrior Clone();

public static Warrior CreateWarrior(WarriorTypes parWarriorType)

{

Warrior warrior = null;

if (\_registry.Contains(parWarriorType))

{

warrior = (Warrior)\_registry[parWarriorType];

return warrior.Clone();

}

return warrior;

}

protected void AddPrototype(Warrior parWarrior)

{

\_registry[parWarrior.\_warriorType] = parWarrior;

}

protected void RemovePrototype(WarriorTypes parWarriorType)

{

\_registry.Remove(parWarriorType);

}

public abstract string Info();

}

public class Infantryman : Warrior

{

public Infantryman()

: base(WarriorTypes.Infantryman)

{

base.AddPrototype(this);

}

public override Warrior Clone()

{

Infantryman infantryman = new Infantryman();

return infantryman;

}

public override string Info() { return "Infantryman"; }

}

public class Archer : Warrior

{

private int \_arrousCount = 0;

private static Archer \_prototype;

public Archer()

: base(WarriorTypes.Archer)

{

\_arrousCount = 10;

base.AddPrototype(this);

}

public override Warrior Clone()

{

Archer archer = new Archer();

archer.\_arrousCount = this.\_arrousCount;

return archer;

}

public override string Info() { return "Archer"; }

}

public class Horseman : Warrior

{

private Color \_horseColor;

public Horseman()

: base(WarriorTypes.Horseman)

{

\_horseColor = Color.Black;

base.AddPrototype(this);

}

public override Warrior Clone()

{

Horseman horseman = new Horseman();

horseman.\_horseColor = this.\_horseColor;

return horseman;

}

public override string Info() { return "Horseman"; }

}

Пример использования:

// Инициализация клонов

Infantryman infantryman = new Infantryman();

Archer archer = new Archer();

Horseman horseman = new Horseman();

// Создание армии

List<Warrior> army = new List<Warrior>();

army.Add(Warrior.CreateWarrior(WarriorTypes.Infantryman));

army.Add(Warrior.CreateWarrior(WarriorTypes.Archer));

army.Add(Warrior.CreateWarrior(WarriorTypes.Horseman));

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 СreateWarrior. 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 хеша, каждый элемент которого представляет собой пару " тип воина" - "его прототип".

Добавление прототипов в реестр происходит автоматически при создании экземпляра класса одного из типов воинов. При этом, если в реестре существует прототип данного типа, то он будет заменён на новый.

Для приведенной реализации шаблона «Прототип» можно отметить следующие особенности:

* Создавать новых воинов можно как при помощи обобщенного конструктора, так и непосредственно (при этом в реестре автоматически будут обновляться объекты-клоны);
* Отсутствует недостаток реализации на базе обобщенного конструктора для шаблона «Фабричный метод», а именно базовый класс Warrior ничего не знает о своих подклассах.

**Реализация шаблона на базе выделенного класса-фабрики**

Диаграмма классов:

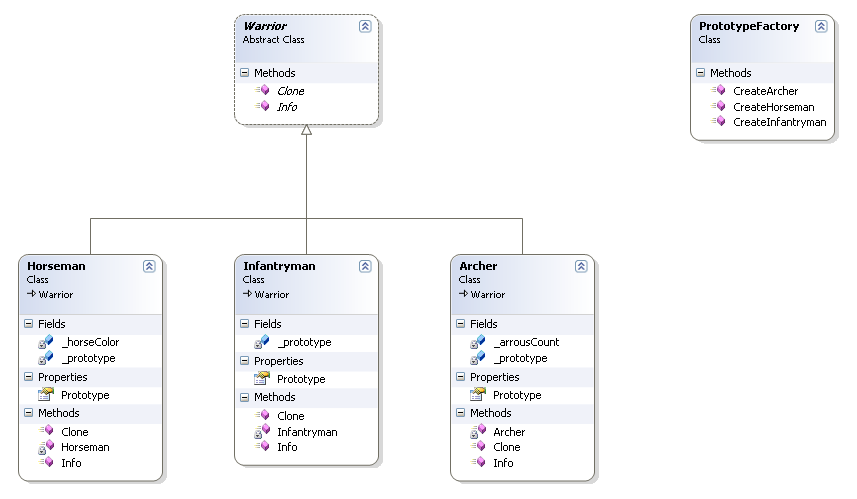


Рисунок 18 - Пример реализации шаблона «Прототип» на базе выделенного класса - фабрики

Исходный код:

public abstract class Warrior

{

public abstract Warrior Clone();

public abstract string Info();

}

public class Infantryman : Warrior

{

private static Infantryman \_prototype = new Infantryman();

private Infantryman() { }

public override Warrior Clone()

{

Infantryman infantryman = new Infantryman();

return infantryman;

}

public override string Info() { return "Infantryman"; }

public static Infantryman Prototype

{

get { return \_prototype; }

set { \_prototype = value; }

}

}

public class Archer : Warrior

{

private static Archer \_prototype = new Archer();

private int \_arrousCount = 0;

private Archer(int parArrowsCount) { \_arrowsCount = parArrowsCount; }

public override Warrior Clone()

{

Archer archer = new Archer();

archer.\_arrousCount = this.\_arrousCount;

return archer;

}

public override string Info() { return "Archer"; }

public static Archer Prototype

{

get { return \_prototype; }

set { \_prototype = value; }

}

}

public class Horseman : Warrior

{

private static Horseman \_prototype = new Horseman();

private Color \_horseColor;

private Horseman(Color parHorseColor) { \_horseColor = parHorseColor; }

public override Warrior Clone()

{

Horseman horseman = new Horseman();

horseman.\_horseColor = this.\_horseColor;

return horseman;

}

public override string Info() { return "Horseman"; }

public static Horseman Prototype

{

get { return \_prototype; }

set { \_prototype = value; }

}

}

public class PrototypeFactory

{

public static Warrior CreateInfantryman()

{

return Infantryman.Prototype.Clone();

}

public static Warrior CreateArcher()

{

return Archer.Prototype.Clone();

}

public static Warrior CreateHorseman()

{

return Horseman.Prototype.Clone();

}

}

Пример использования:

// Настройка прототипов

Infantryman infantryman = new Infantryman();

Infantryman.Prototype = infantryman;

Archer archer = new Archer(20);

Archer.Prototype = archer;

Horseman horseman = new Horseman(Color.Black);

Horseman.Prototype = horseman;

// Использование настоенных прототипов

List<Warrior> army = new List<Warrior>();

army.Add(PrototypeFactory.CreateInfantryman());

army.Add(PrototypeFactory.CreateArcher());

army.Add(PrototypeFactory.CreateHorseman());

В приведенной реализации, для упрощения кода, реестр прототипов не ведется. Воины всех родов войск создаются при помощи соответствующих методов фабричного класса PrototypeFactory.

**Достоинства:**

* 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:**

* 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.

**Шаблон «Одиночка»**

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.

Шаблон «Одиночка» (Singleton) предоставляет такие возможности.

Для того, чтобы у класса существовал единственный экземпляр, необходимо запретить создавать экземпляр класса извне и контролировать единственность внутри кода самого класса.

Это обеспечивается при помощи следующего решения:

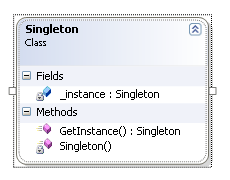


Рисунок 19 - Диаграмма классов для реализации шаблона «Одиночка»

Исходный код:

public class Singleton

{

private static Singleton \_instance = null;

private Singleton()

{

}

public static Singleton GetInstance()

{

if (\_instance == null)

\_instance = new Singleton();

return \_instance;

}

}

В данном примере реализован закрытый конструктор, который не даст создавать экземпляр класса извне.

Создание единственного экземпляра контролируется закрытым статическим полем \_instance, которое инициализируется при первом обращении к получению экземпляра класса через метод GetInstance(), который предоставляет клиентам единственный экземпляр класса.

**Достоинства:**

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

**Недостатки:**

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

**Задания для практической работы**

В каждом из вариантов (можно выбрать свой, предварительно обсудив с преподавателем) указан шаблон для реализации и проект, использующий этот шаблон. Необходимо **нарисовать диаграмму классов** возможной реализации программы (проектирование) для выбранной задачи.

**Вариант №1**

Шаблон “Стратегия”. Проект “Принтеры”. В проекте должны быть реализованы разные модели принтеров, которые выполняют разные виды печати.

**Вариант №2**

Шаблон “Наблюдатель”. Проект “Оповещение постов ГИБДД”. В проекте должна быть реализована отправка сообщений всем постам ГИБДД.

**Вариант №3**

Шаблон “Декоратор”. Проект “Универсальная электронная карта”. В проекте должна быть реализована универсальная электронная карта, в которой есть функции паспорта, страхового полиса, банковской карты и т. д.

**Вариант №4**

Шаблон “Фабричный метод”. Проект “Фабрика смартфонов”. В проекте должно быть реализовано создание смартфонов с различными характеристиками.

**Вариант №5**

Шаблон “Абстрактная фабрика”. Проект “Заводы по производству автомобилей”. В проекте должно быть реализована возможность создавать автомобили различных типов на разных заводах.

**Вариант №6**

Шаблон “Команда”. Проект “Клавиатура настраимаемого калькулятора”. Цифровые и арифметические кнопки имеют фиксированную функцию, а остальные могут менять своё назначение.

**Для тех, кто хочет выполнить болеее сложный вариант задания, необходимо выбрать один из порождающих шаблонов и для ранее разработанной иерархии классов, представленной в тексте, реализовать данный шаблон на языке JAVA, в виде работоспособной программы.**