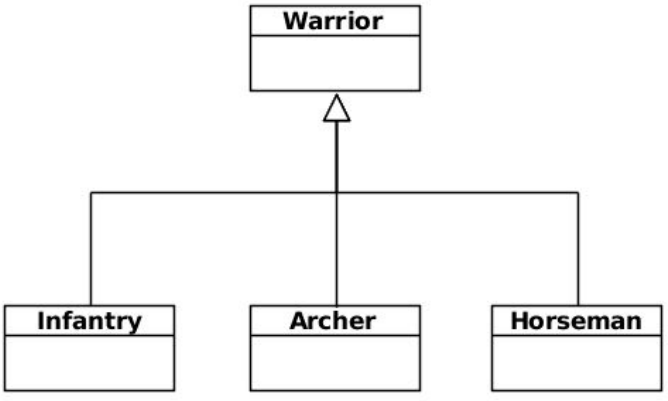
Создание новых объектов является одной из наиболее распространенных задач, встающих перед разработчиками программных систем. Порождающие паттерны проектирования предназначены для абстрагирования процесса создания объектов, позволяя системе оставаться независимой как от самого процесса порождения, так и от типов порождаемых объектов.

Порождающие шаблоны инкапсулируют знания о конкретных классах, которые применяются в системе. Они скрывают детали того, как эти классы создаются и стыкуются. Единственная информация об объектах, известная системе, — это их интерфейсы, определенные с помощью абстрактных классов. Следовательно, порождающие шаблоны обеспечивают большую гибкость при решении вопроса о том, что создается, кто это создает, как и когда. Можно собрать систему из «готовых» объектов с самой различной структурой и функциональностью статически (на этапе компиляции) или динамически (во время выполнения).

Пусть для примера мы разрабатываем стратегическую игру. 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 мы планируем развивать игру 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.



Полиморфный базовый класс Warrior определяет общий интерфейс, а производные от него классы Infantry, 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 новых персонажей, непосредственно указывая их конкретные

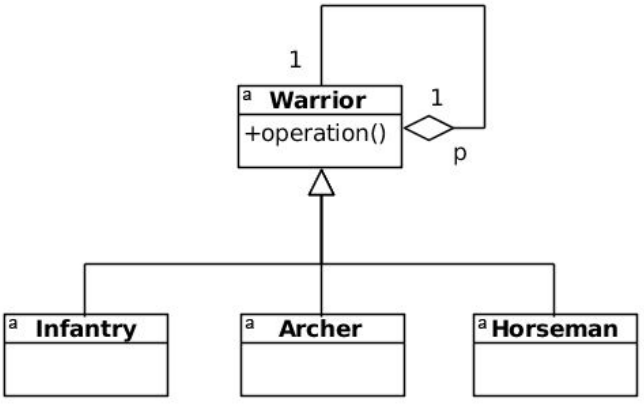
типы. Система должна оставаться расширяемой путем добавления объектов новых типов. Непосредственное использование выражения new является нежелательным, так как в этом случае код создания объектов с указанием конкретных типов может получиться разбросанным по всему приложению. В таком случае добавлять новые типы персонажей или заменять существующие будет весьма затруднительно. Чтобы бороться с этой проблемой, используют идиому виртуального конструктора, а также шаблоны “Фабричный метод” и “Абстрактная фабрика”.

Виртуальный конструктор

Для начала вспомним, что дает использование виртуальных функций. Механизм полиморфизма на базе виртуальных функций и наследования позволяет автоматически направлять вызовы производным классам, ничего о них не зная. Если у пользователя есть ссылка или указатель на базовый класс, значением которого является адрес объекта производного класса, то для вызова некоторого метода производного класса пользователю достаточно знать об интерфейсе базового класса.

Однако, подобное поведение "ломается" непосредственно при создании объектов производных классов, а именно, при присваивании значению ссылке или указателю на базовый класс адреса объекта производного класса обязательно нужно указывать этот производный класс. Наличие виртуального конструктора позволило бы клиентам с помощью конструктора базового класса создавать объекты производных классов, ничего не зная об их конкретных типах. Как известно, в распространённых языках программирования нет прямой поддержки виртуального конструктора, однако, существует идиома, с помощью которой можно имитировать его работу.

Для этого опять же используют парадигму класса-обёртки, которую мы обсуждали при рассмотрении структурных паттернов ранее. Класс обёртки представляет собой базовый класс, содержащий указатель на объект того же базового типа. На практике этот указатель будет указывать на объект некоторого производного класса. Когда клиент хочет создать объект некоторого типа, то он должен передать идентификатор этого типа в конструктор класса-обёртки.



На основании этого идентификатора конструктор создает в куче объект соответствующего производного класса, адрес которого становится значением указателя на внутренний объект. В дальнейшем все запросы от пользователя перенаправляются ему. В случае нашего примера со стратегической игрой, код бы выглядел как-то так:

```
enum Warrior_ID { Infantryman_ID, Archer_ID, Horseman_ID };

class Warrior
{
 public:
   Warrior( Warrior_ID id )
   {
      if (id == Infantryman_ID) p = new Infantryman;
      else if (id == Archer_ID) p = new Archer;
      else if (id == Horseman_ID) p = new Horseman;
      else assert( false);
   }
   virtual void info() { p->info(); }
   virtual ~Warrior() { delete p;  p=0; }
 private:
   Warrior* p;
};

class Infantry: public Warrior
{
 public:
   void info() { cout << "Infantry" << endl; }
 private:
   Infantryman(): Warrior() {}
   Infantryman(Infantryman&);
   friend class Warrior;
};
 // ...

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

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

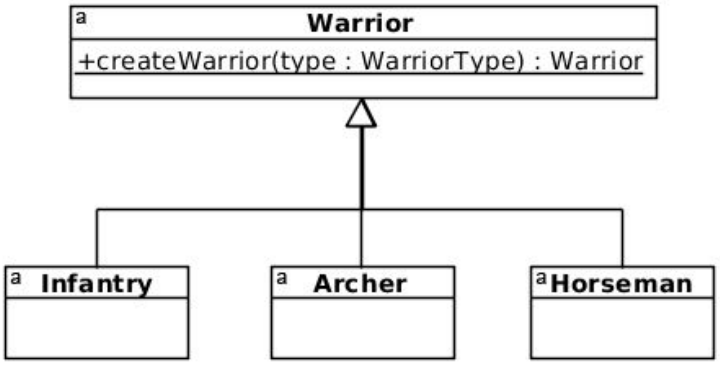
Рассмотрим особенности реализации идиомы виртуального конструктора:

- Класс Warrior одновременно является основным классом-обёрткой и базовым классом для подклассов Infantry, Archer, Horseman.

- Пользователи не могут непосредственно создавать воинов разных родов войск, так как конструкторы подклассов воинов не являются общедоступными.
- Для создания воина некоторого типа используется конструктор базового класса Warrior(Warrior_ID id). По полученному идентификатору типа в куче создается объект-воин, адрес которого и присваивается указателю на внутренний объект.
- Метод info() в базовом классе должен быть объявлен виртуальным для того, чтобы обёртка могла автоматически перенаправить этот вызов в соответствующий объект по указателю на объект базового класса.
- При разрушении обёртки его деструктор освобождает память, занимаемую внутренним объектом.

Фабричный метод

Шаблон “Фабричный метод” продолжает идею виртуального конструктора. Первый его вариант по сути реализуется ровно так же, только вместо конструктора в базовом классе используются обычный статический (фабричный) метод, создающий нужный объект.



Реализуется этот метод так же через набор условий или switch. Этот вариант паттерна Factory Method пользуется популярностью благодаря своей простоте. С точки зрения "чистоты" объектно-ориентированного кода у этого варианта есть следующие недостатки:

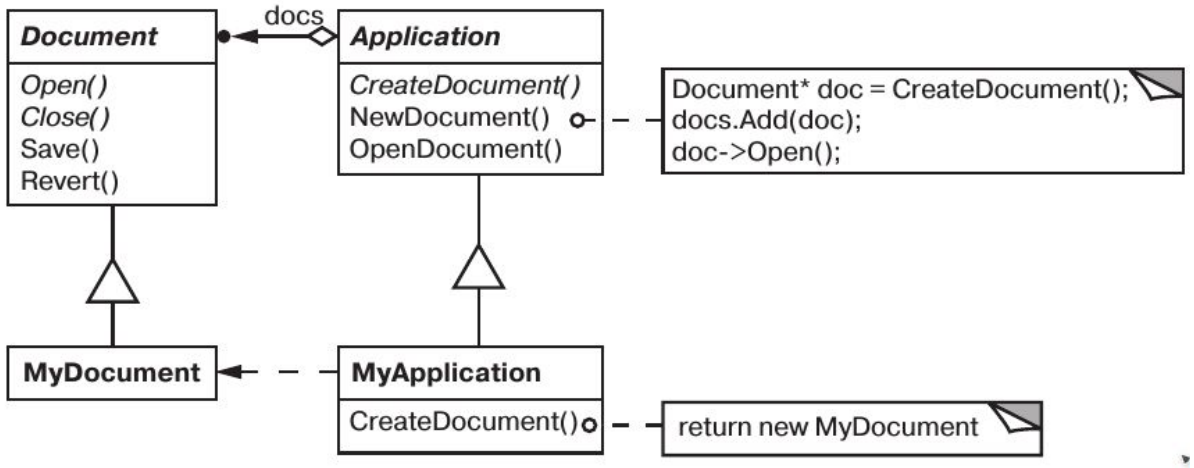
- так как код по созданию объектов всех возможных типов сосредоточен в статическом фабричном методе класса Warrior, то базовый класс Warrior обладает знанием обо всех производных от него классах, что является нетипичным для объектно-ориентированного подхода;
- подобное использование оператора switch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

Эти недостатки исправляет другой вариант шаблона, который определяет лишь интерфейс для создания объекта и делегирует создание реальных объектов подклассам.

Рассмотрим фреймворк для приложений, способных представлять пользователю сразу несколько документов. Две основных абстракции в таком фреймворке – это классы Application и Document. Оба класса абстрактные, поэтому клиенты должны порождать от них подклассы для создания специфичных для приложения реализаций. Например, чтобы создать приложение для рисования, мы определим классы

DrawingApplication и DrawingDocument. Класс Application отвечает за управление документами и создает их по мере необходимости, допустим, когда пользователь выбирает из меню пункт Open или New.

Поскольку решение о том, какой подкласс класса Document инстанцировать, зависит от приложения, то Application не может «предсказать», что именно понадобится. Этому классу известно лишь, *когда* нужно инстанцировать новый документ, а не *какой* документ создать. Возникает дилемма: фреймворк должен инстанцировать классы, но знает он лишь об абстрактных классах, которые инстанцировать нельзя. Вот тут и помогает фабричный метод, в котором инкапсулируется создание экземпляра подкласса Document, и это знание выводится за пределы фреймворка.



Подклассы класса Application переопределяют абстрактную операцию CreateDocument таким образом, чтобы она возвращала подходящий подкласс класса Document. Таким образом, класс Application может инстанцировать специфические для приложения документы, ничего не зная об их классах.

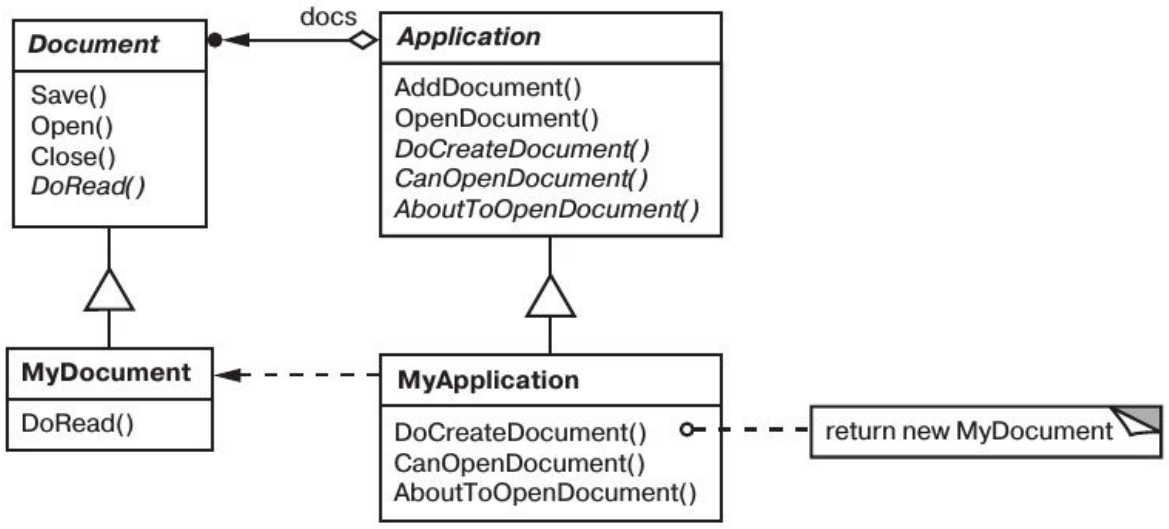
К недостаткам последнего варианта шаблона можно отнести то, что иногда приходится создавать новый класс с единственной целью переопределить шаблонный метод.

Итого, паттерн “Фабричный метод” хорошо применим, когда:

- классу заранее неизвестно, объекты каких классов ему нужно создавать;
- класс спроектирован так, чтобы объекты, которые он создает, специфицировались подклассами;
- класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и вы планируете локализовать знание о том, какой класс принимает эти обязанности на себя.

Шаблонный метод

Вообще стоит заметить, что в этом примере фабричный метод используется вместе с другим паттерном -- шаблонным методом. Рассмотрим более подробно то, как может быть устроен абстрактный класс Application.



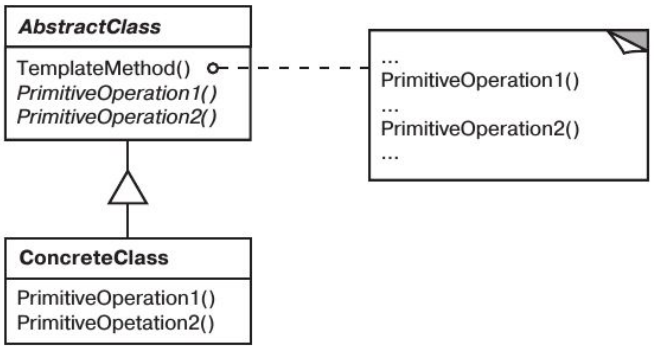
В нём определен алгоритм открытия и считывания документа в операции OpenDocument:

```
void Application::OpenDocument (const char* name) {
   if (!CanOpenDocument(name)) {
       return;
   }
   Document* doc = DoCreateDocument();

   if (doc) {
       _docs->AddDocument(doc);
       AboutToOpenDocument(doc);
       doc->Open();
       doc->DoRead();
   }
}
```

Операция OpenDocument определяет все шаги открытия документа. Она проверяет, можно ли открыть документ, создает объект класса Document, добавляет его к набору документов и считывает документ из файла.

Операцию вида OpenDocument мы будем называть шаблонным методом, описывающим алгоритм в терминах абстрактных операций, которые замещены в подклассах для получения нужного поведения. Подклассы класса Application выполняют проверку возможности открытия (CanOpenDocument) и создания документа (DoCreateDocument). Подклассы класса Document считывают документ (DoRead). Шаблонный метод определяет также операцию, которая позволяет подклассам Application получить информацию о том, что документ вот-вот будет открыт (AboutToOpenDocument). Определяя некоторые шаги алгоритма с помощью абстрактных операций, шаблонный метод фиксирует их последовательность, но позволяет реализовать их в подклассах классов Application и Document.



Шаблонные методы – один из фундаментальных приемов повторного использования кода. Они особенно важны в библиотеках классов, поскольку предоставляют возможность вынести общее поведение в библиотечные классы. Шаблонные методы приводят к инвертированной структуре кода, когда родительский класс вызывает операции подкласса, а не наоборот (“не звоните нам, мы сами вам позвоним”).

Часто шаблонные методы вызывают фабричные методы или операции-зацепки (hook operations), реализующие поведение по умолчанию, которое может быть расширено в подклассах. Часто такая операция по умолчанию не делает ничего. Важно, чтобы в шаблонном методе четко различались операции-зацепки (которые *можно* замещать) и абстрактные операции (которые *нужно* замещать). Чтобы повторно использовать абстрактный класс с максимальной эффективностью, авторы подклассов должны понимать, какие операции предназначены для замещения.

Паттерн шаблонный метод следует использовать:

- чтобы однократно использовать инвариантные части алгоритма, оставляя реализацию изменяющегося поведения на усмотрение подклассов;
- когда нужно вычленить и локализовать в одном классе поведение, общее для всех подклассов, дабы избежать дублирования кода. Сначала идентифицируются различия в существующем коде, а затем они выносятся в отдельные операции. В конечном итоге различающиеся фрагменты кода заменяются шаблонным методом, из которого вызываются новые операции;
- для управления расширениями подклассов. Можно определить шаблонный метод так, что он будет вызывать операции-хуки в определенных точках, разрешив тем самым расширение только в этих точках.

Абстрактная фабрика

На основе шаблонного метода реализуется другой паттерн, Абстрактная фабрика, который позволяет целостно создавать семейства связанных друг с другом объектов, исключая возможность одновременного использования объектов из разных семейств в одном контексте.

Вернёмся к примеру с текстовым редактором. Предположим, что мы хотим поддержать разные стили отображения графических элементов. К примеру, мы хотим сделать интерфейс, в котором виджеты будут иметь скругления а-ля Mac OS, другой вариант будет сугубо квадратный, а третий вообще будет оформлен в стиле псевдографики. Разумеется, мы хотим это сделать максимально гибко, чтобы не

пришлось перепроектировать и переделывать всё приложение, да к тому же чтобы поддержку новых стилей отображения можно было добавлять быстро и удобно. К тому же, было бы неплохо сделать поддержку смены внешнего облика приложения во время его выполнения.

Будем предполагать, что имеется два набора классов виджетов, с помощью которых реализуются стандарты внешнего облика:

- набор абстрактных подклассов класса Glyph для каждой категории виджетов. Например, абстрактный класс ScrollBar будет дополнять интерфейс глифа с целью получения операций прокрутки общего вида, а Button – это абстрактный класс, добавляющий операции для работы с кнопками;
- набор конкретных подклассов для каждого абстрактного подкласса, в которых реализованы стандарты внешнего облика. Так, у ScrollBar могут быть подклассы RoundedScrollBar и PseudoScrollBar, реализующие полосы прокрутки в стиле со скруглёнными углами и в стиле псевдографики соответственно.

Редактор должен различать виджеты для разных стилей внешнего оформления. Например, когда необходимо поместить в интерфейс кнопку, редактор должен инстанцировать подкласс класса Glyph для нужного стиля кнопки (RoundedButton, SquareButton и т.д.). Как мы уже обсуждали выше, это нельзя сделать непосредственно, например, вызвав конструктор. При этом была бы жестко закодирована кнопка одного конкретного стиля, значит, выбрать нужный стиль во время выполнения оказалось бы невозможно. Кроме того, мы были бы вынуждены отслеживать и изменять каждый такой вызов конструктора при поддержке других стилей редактора. А ведь кнопки – это лишь один элемент пользовательского интерфейса. Загромождение кода вызовами конструкторов для разных классов внешнего облика вызывает существенные неудобства при сопровождении. Стоит что-нибудь пропустить – и в приложении, составленном из скруглённых виджетов, появится квадратная кнопка!

Необходим какой-то способ определить нужный стандарт внешнего облика для создания подходящих виджетов. При этом надо не только постараться избежать явных вызовов конструкторов, но и уметь без труда заменять весь набор виджетов.

Итак, для создания экземпляра виджета полосы прокрутки со скруглёнными краями обычно было достаточно написать следующий код:

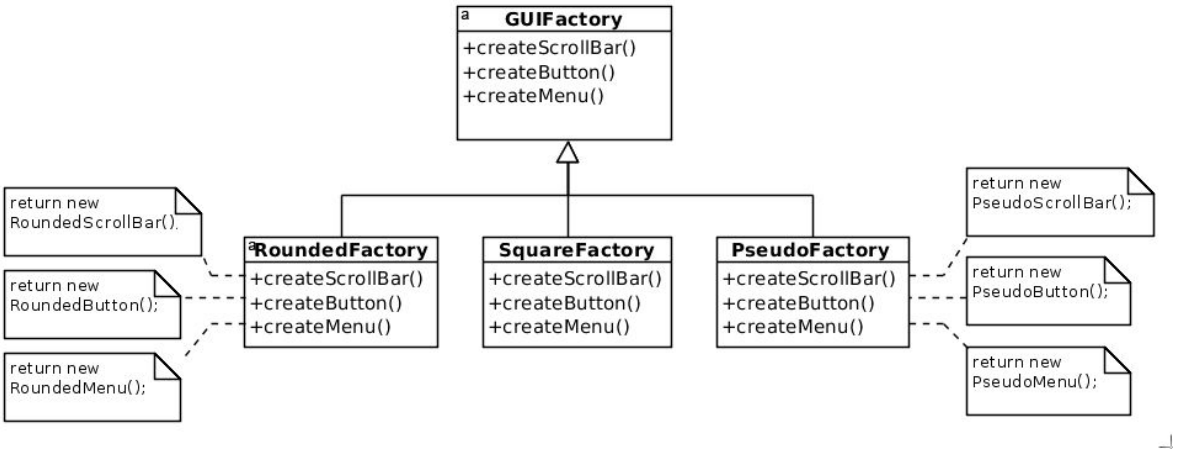
```
ScrollBar* bar = new RoundedScrollBar;
```

Но так как такого кода мы хотим избегать (так как хотим минимизировать зависимость редактора от стандарта внешнего облика), объект bar можно проинициализировать так:

```
ScrollBar* bar = guiFactory->createScrollBar();
```

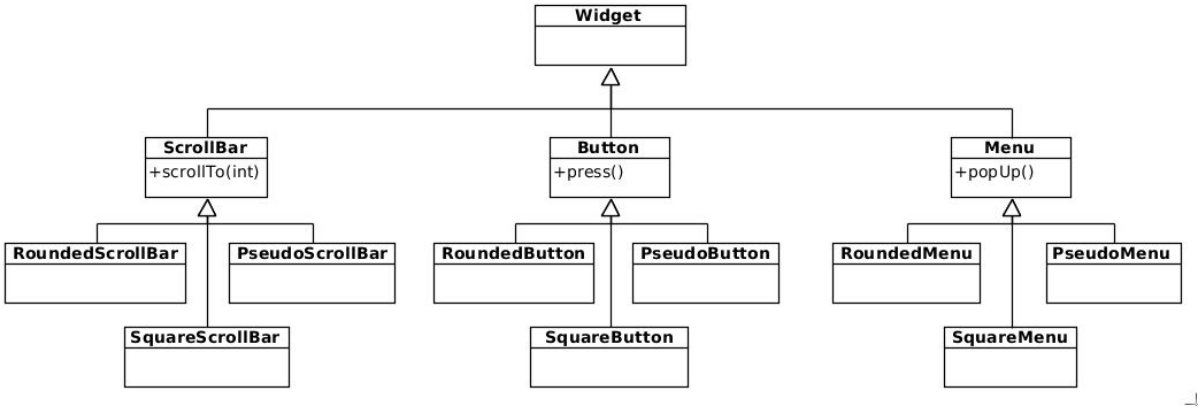
Тут guiFactory -- объект класса RoundedFactory. Его метод createScrollBar() возвращает новый экземпляр подходящего подкласса ScrollBar, который соответствует желательному варианту внешнего облика, в нашем случае – а-ля Mac OS. С точки зрения клиентов результат тот же самый, что и при прямом обращении к конструктору RoundedScrollBar. Но есть и существенное отличие: нигде в коде больше

не упоминается имя Rounded. Объект guiFactory абстрагирует процесс создания не только полос прокрутки для скруглённых виджетов, но и любых других. Более того, guiFactory не ограничен изготовлением только полос прокрутки. Его можно применять для производства любых виджетов, включая кнопки, поля ввода, меню и т.д.



Все это стало возможным, поскольку RoundedFactory является подклассом GUIFactory – абстрактного класса, который определяет общий интерфейс для создания виджетов. В нем есть такие операции, как CreateScrollBar() и CreateButton(), для инстанцирования различных видов виджетов. Подклассы GUIFactory реализуют эти операции, возвращая виджеты вроде RoundedScrollBar и SquareButton, которые имеют нужный внешний облик и поведение.

Мы говорим, что фабрики изготавливают объекты. Продукты, изготовленные фабриками, связаны друг с другом; в нашем случае все такие продукты – это виджеты, имеющие один и тот же внешний облик.



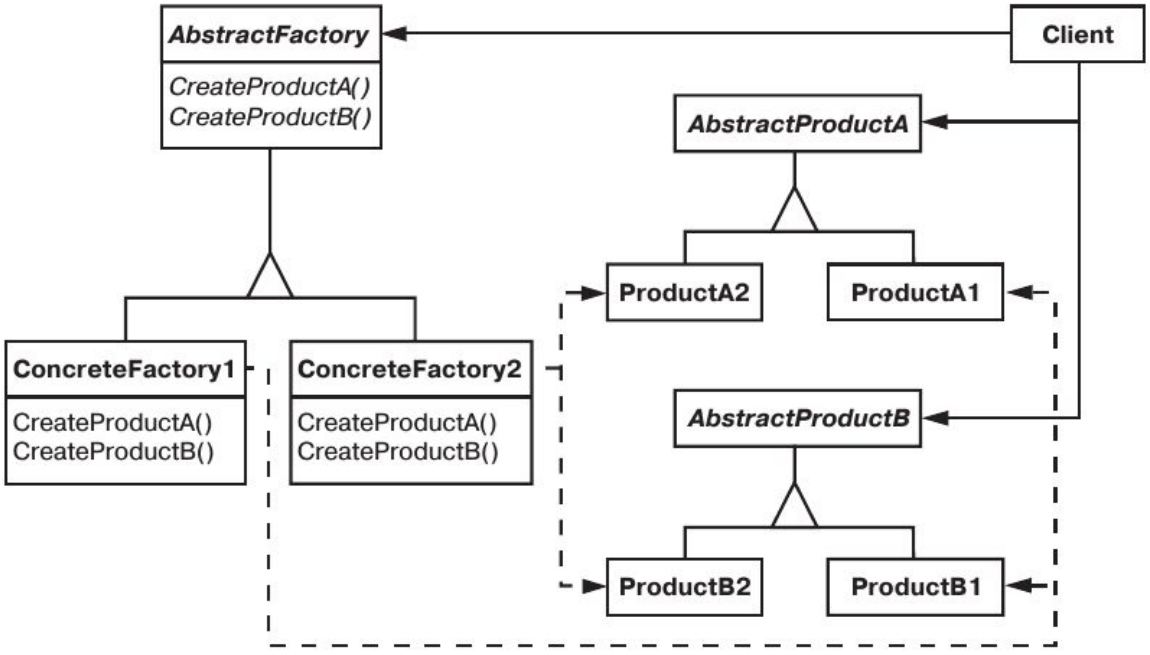
Экземпляр класса GUIFactory может быть создан любым способом. Переменная guiFactory может быть глобальной или статическим членом хорошо известного класса или даже локальной, если весь пользовательский интерфейс создается внутри одного класса или функции. В принципе, в приложении редко требуется более одной фабрики виджетов, поэтому часто этот шаблон используют совместно с шаблоном Одиночка (будет рассмотрен далее). Важно, однако, чтобы фабрика guiFactory была

инициализирована до того, как начнет использоваться для производства объектов, но после того, как стало известно, какой внешний облик нужен.

Когда вариант внешнего облика известен на этапе компиляции, то guiFactory можно инициализировать простым присваиванием в начале программы. Если же пользователю разрешается задавать внешний облик с помощью строки-параметра при запуске, значения в реестре или файле настроек, придётся писать условные операторы или switch. Но вся остальная работа с виджетами будет лишена этих конструкций. Если требуется часто переключаться между фабриками, можно создать их все и хранить в ассоциативном массиве по значению строк или их идентификаторов типов.

Используйте паттерн Абстрактная фабрика, когда:

- система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты;
- система должна конфигурироваться одним из семейств составляющих ее объектов;
- входящие в семейство взаимосвязанные объекты должны использоваться вместе и вам необходимо обеспечить выполнение этого ограничения;
- вы хотите предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.



Паттерн абстрактная фабрика обладает следующими плюсами и минусами:

- *изолирует конкретные классы*. Помогает контролировать классы объектов, создаваемых приложением. Поскольку фабрика инкапсулирует ответственность за создание классов и сам процесс их создания, то она изолирует клиента от деталей реализации классов. Клиенты манипулируют экземплярами через их абстрактные интерфейсы. Имена изготавливаемых классов известны только конкретной фабрике, в коде клиента они не упоминаются;
- *упрощает замену семейств продуктов*. Класс конкретной фабрики появляется в приложении только один раз: при инстанцировании. Это облегчает замену

используемой приложением конкретной фабрики. Приложение может изменить конфигурацию продуктов, просто подставив новую конкретную фабрику. Поскольку абстрактная фабрика создает все семейство продуктов, то и заменяется сразу все семейство. В нашем примере пользовательского интерфейса перейти от закруглённых виджетов к квадратным можно, просто переключившись на продукты соответствующей фабрики и заново создав пользовательский интерфейс;

- *гарантирует сочетаемость продуктов*. Если продукты некоторого семейства спроектированы для совместного использования, то важно, чтобы приложение в каждый момент времени работало только с продуктами единственного семейства. Класс AbstractFactory позволяет легко соблюсти это ограничение;
- *поддержать новый вид продуктов трудно*. Расширение абстрактной фабрики для изготовления новых видов продуктов – непростая задача. Интерфейс AbstractFactory фиксирует набор продуктов, которые можно создать. Для поддержки новых продуктов необходимо расширить интерфейс фабрики, то есть изменить класс AbstractFactory и все его подклассы.

Одиночка

Для некоторых классов важно, чтобы существовал только один их экземпляр. Например, в системе должен быть только один реестр с настройками, одна файловая система или единственный оконный менеджер. Как гарантировать, что у класса есть единственный экземпляр и что этот экземпляр легко доступен? Глобальная переменная дает доступ к объекту, но не запрещает инстанцировать класс в нескольких экземплярах.

Более удачное решение – сам класс контролирует то, что у него есть только один экземпляр, может запретить создание дополнительных экземпляров, перехватывая запросы на создание новых объектов, и он же способен предоставить доступ к своему экземпляру. Это и есть назначение паттерна Одиночка.

В принципе, вот и весь шаблон, тут даже диаграммы никакой не нарисуешь. Все тонкости тут в реализации.

Самая простая синглтона реализация на Java выглядит, наверное, как-то так:

```
public class Singleton {
   private static Singleton instance;

   private Singleton (){}

   public static Singleton getInstance(){
       if (instance == null){
           instance = new Singleton();
       }
       return instance;
   }
}
```

Решение, в принципе, неплохое, но хорошо подходит исключительно для однопоточных приложений. Другой вариант решает проблему многопоточного доступа, но при этом теряет ленивую инициализацию (Объект instance будет создан classloader-ом во время инициализации класса) и возможность обработки исключений во время вызова конструктора.

```
public class Singleton {
   private static Singleton instance = new Singleton();

   private Singleton (){}

   public static Singleton getInstance(){
       return instance;
   }
}
```

Есть ещё один очень элегантный вариант, предложенный Джошуа Блохом в Effective Java:

```
public enum Singleton {
   INSTANCE;
}
```

Автоматически получаем потокобезопасность, сериализацию и ещё много чего, но опять же ленивой инициализации нет.

Теперь попытаемся получить потокобезопасность, сохранив ленивую инициализацию. Самое простое решение -- сделать метод getInstance() synchronized:

```
public class Singleton {
  private static Singleton instance;

  public static synchronized Singleton getInstance() {
     if (instance == null) {
        instance = new Singleton();
     }
     return instance;
  }
}
```

У этого варианта есть только один недостаток. Синхронизация полезна только один раз, при первом обращении к getInstance(), после этого каждый раз при обращении этому методу синхронизация просто забирает время. Если вызов getInstance() не происходит достаточно часто, то этот метод имеет преимущество перед остальными – прост, понятен, лениво инициализируется, дает возможность обрабатывать исключительные ситуации в конструкторе. Те, кто много смотрели в исходники стандартной библиотеки Java, говорят, что этот вариант реализации там чаще всего встречается.

Наиболее распространенный способ победить эту проблему -- так называемый Double-Checked Locking. Оставив дискуссии о Java Memory Model в стороне, заметим, что без ключевого слова volatile этот вариант вообще использовать не стоит.

```
public class Singleton {
   private static volatile Singleton instance;

   public static Singleton getInstance() {
       Singleton localInstance = instance;
       if (localInstance == null) {
           synchronized (Singleton.class) {
               localInstance = instance;
               if (localInstance == null) {
                   instance = localInstance = new Singleton();
               }
           }
       }
       return localInstance;
   }
}
```

Тут ещё много примеров реализации синглтона на разных языках программирования.

Данный паттерн очень активно критикуют, а некоторые даже считают антипаттерном. Основные замечания к нему такие:

- Синглтон нарушает SRP (Single Responsibility Principle) — класс синглтона помимо того, чтобы выполнять свои непосредственные обязанности, занимается еще и контролированием количества своих экземпляров.
- Зависимость обычного класса от синглтона не видна в публичном контракте класса. Так как обычно экземпляр синглтона не передается в параметрах метода, а получается напрямую, через getInstance(), то для выявления зависимости класса от синглтона надо залезть в тело каждого метода -- просто просмотреть публичный контракт объекта недостаточно. Как следствие: сложность рефакторинга при последующей замене синглтона на объект, содержащий несколько экземпляров.
- Глобальное состояние. Про вред глобальных переменных вроде бы уже все знают, но тут та же самая проблема. Когда мы получаем доступ к экземпляру класса, мы не знаем текущее состояние этого класса, и кто и когда его менял, и это состояние может быть вовсе не таким, как ожидается. Иными словами, корректность работы с синглтоном зависит от порядка обращений к нему, что вызывает неявную зависимость подсистем друг от друга и, как следствие, серьезно усложняет разработку.
- Наличие синглтона понижает тестируемость приложения в целом и классов, которые используют синглтон, в частности. Во-первых, вместо синглтона нельзя подпихнуть Mock-объект, а во-вторых, если синглтон имеет интерфейс для изменения своего состояния, то тесты начинают зависеть друг от друга.

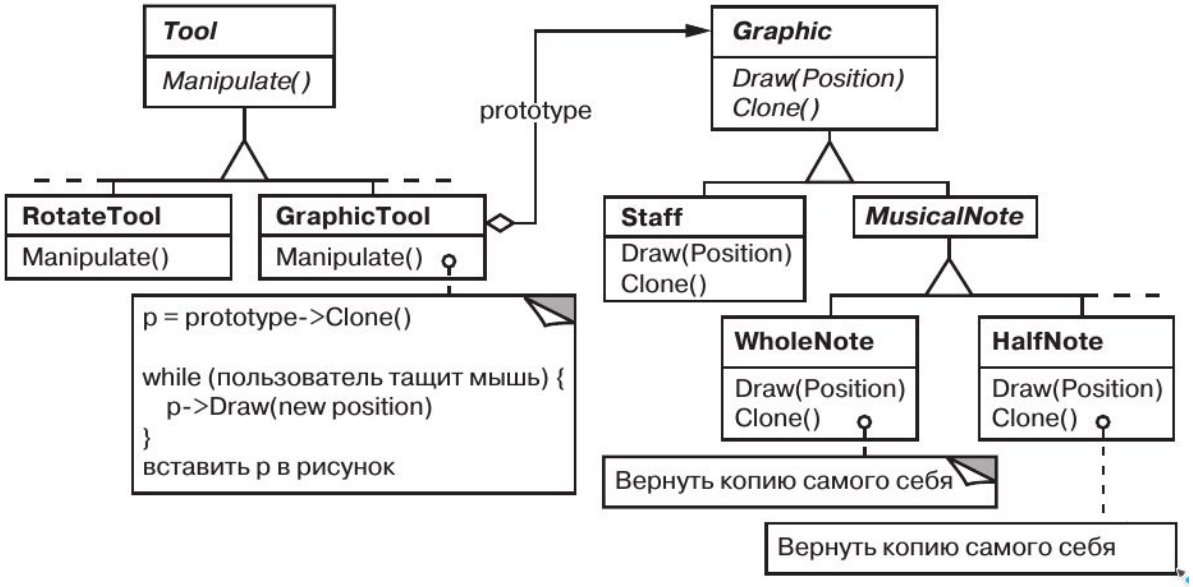
Со всеми этими аргументами сложно спорить, но всё же бывают ситуации, когда использование шаблона оправдано и полезно. Использовать его или нет -- выбор каждого.

Прототип

Рассмотрим теперь для разнообразия пример музыкального редактора.

Построить музыкальный редактор удалось бы путем адаптации общего фреймворка графических редакторов и добавления новых объектов, представляющих ноты, паузы и нотный стан. Во фреймворке редактора может присутствовать палитра инструментов для добавления в партитуру этих музыкальных объектов. Палитра может также содержать инструменты для выбора, перемещения и иных манипуляций с объектами. Так, пользователь, щелкнув, например, по значку четверти поместил бы ее тем самым в партитуру. Или, применив инструмент перемещения, сдвигал бы ноту на стане вверх или вниз, чтобы изменить ее высоту.

Предположим, что фреймворк предоставляет абстрактный класс Graphic для графических компонентов вроде нот и нотных станов, а также абстрактный класс Tool для определения инструментов в палитре. Кроме того, имеется предопределенный подкласс GraphicTool для инструментов, которые создают графические объекты и добавляют их в документ.

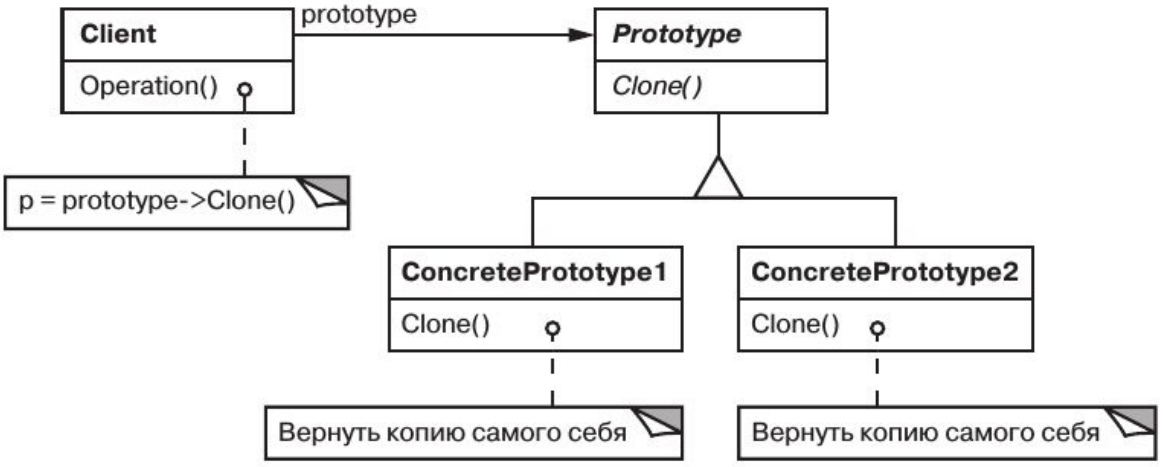


Однако класс GraphicTool создает некую проблему для проектировщика фреймворка. Классы нот и нотных станов специфичны для нашего приложения, а класс GraphicTool принадлежит фреймворку. Этому классу ничего неизвестно о том, как создавать экземпляры наших музыкальных классов и добавлять их в партитуру. Можно было бы породить от GraphicTool подклассы для каждого вида музыкальных объектов, но тогда оказалось бы слишком много классов, отличающихся только тем, какой музыкальный объект они инстанцируют. Мы знаем, что гибкой альтернативой порождению подклассов является композиция. Вопрос в том, как фреймворк мог бы воспользоваться ею для параметризации экземпляров GraphicTool классом того объекта Graphic, который предполагается создать.

Решение – заставить GraphicTool создавать новый графический объект, копируя или «клонируя» экземпляр подкласса класса Graphic. Этот экземпляр мы будем называть прототипом. GraphicTool параметризуется прототипом, который он должен клонировать и добавить в документ. Если все подклассы Graphic поддерживают операцию Clone, то GraphicTool может клонировать любой вид графических объектов.

Итак, в нашем музыкальном редакторе каждый инструмент для создания музыкального объекта – это экземпляр класса GraphicTool, инициализированный тем или иным прототипом. Любой экземпляр GraphicTool будет создавать музыкальный объект, клонируя его прототип и добавляя клон в партитуру.

При использовании данного шаблона для порождения объекта типа в системе должен существовать его прототип. 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.



У прототипа те же самые результаты, что у Абстрактной фабрики и Строителя: он скрывает от клиента конкретные классы продуктов, уменьшая тем самым число известных клиенту имен. Кроме того, все эти паттерны позволяют клиентам работать со специфичными для приложения классами без модификаций. При этом Прототип позволяет включать новый конкретный класс продуктов в систему, просто сообщив клиенту о новом экземпляре-прототипе. Это несколько более гибкое решение по сравнению с тем, что удастся сделать с помощью других порождающих паттернов, ибо клиент может устанавливать и удалять прототипы во время выполнения.

Используйте паттерн прототип, когда система не должна зависеть от того, как в ней создаются, компонуются и представляются продукты:

- инстанцируемые классы определяются во время выполнения, например, с помощью динамической загрузки;
- для того чтобы избежать построения иерархий классов или фабрик, параллельных иерархии классов продуктов;
- экземпляры класса могут находиться в одном из не очень большого числа различных состояний. Может оказаться удобнее установить соответствующее

число прототипов и клонировать их, а не инстанцировать каждый раз класс вручную в подходящем состоянии.

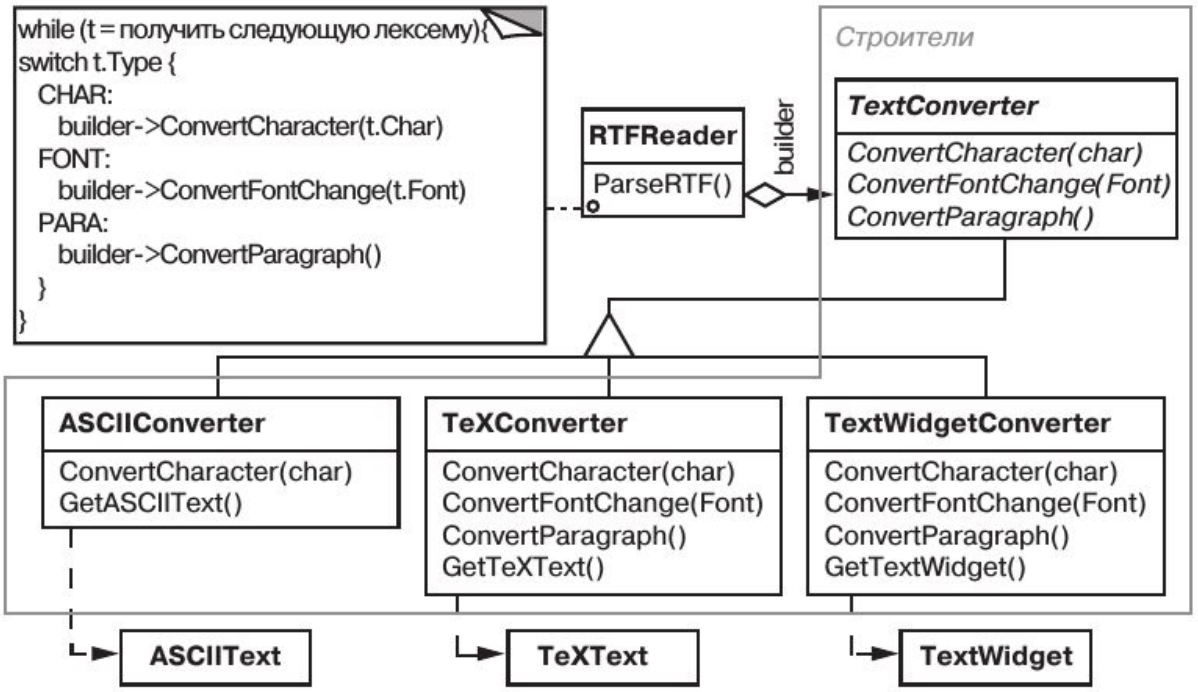
Основной недостаток паттерна прототип заключается в том, что каждый подкласс класса Prototype должен реализовывать операцию Clone, а это далеко не всегда просто. Например, сложно добавить операцию Clone, когда рассматриваемые классы уже существуют. Проблемы возникают и в случае, если во внутреннем представлении объекта есть другие объекты или наличествуют круговые ссылки.

Строитель

Рассмотрим пример подсистемы-конвертера, которая будет читать документ (например, в формате RTF) и сохранять его в другие форматы. При этом, разумеется, заранее неизвестно, какие конкретные целевые форматы будут выбраны, да и добавлять их во время выполнения тоже было бы неплохо.

Пусть у нас будет класс RTFReader, читающий и разбирающий входной файл. Этот класс можно параметризовать с помощью объекта TextConverter, который бы уже преобразовывал RTF в другой текстовый формат. При разборе документа в формате RTF класс RTFReader вызывает TextConverter для выполнения преобразования. Всякий раз, как RTFReader распознает лексему RTF (простой текст или управляющее слово), для ее преобразования объекту TextConverter посылается запрос. Объекты TextConverter отвечают как за преобразование данных, так и за представление лексемы в конкретном формате.

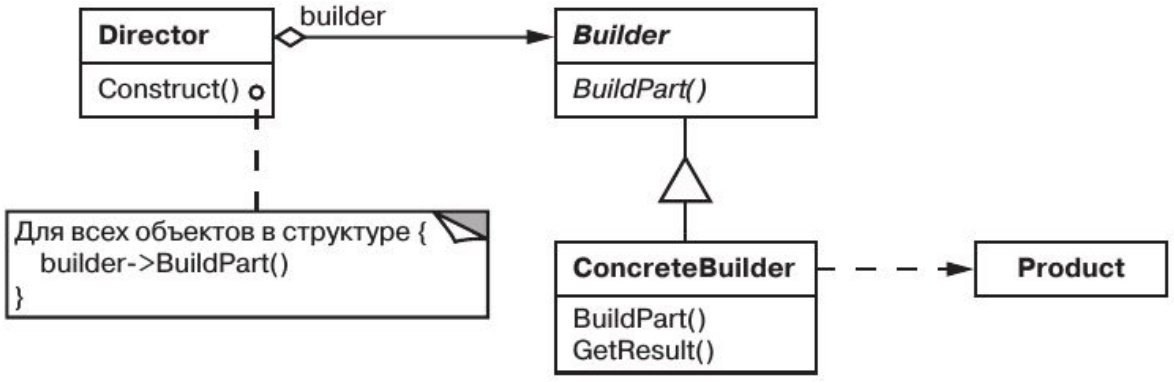
Подклассы TextConverter специализируются на различных преобразованиях и форматах. Например, ASCIIConverter игнорирует запросы на преобразование чего бы то ни было, кроме простого текста. С другой стороны, TeXConverter будет реализовывать все запросы для получения представления в формате редактора TEX, собирая по ходу необходимую информацию о стилях. А TextWidgetConverter станет строить сложный объект пользовательского интерфейса, который позволит пользователю просматривать и редактировать текст.



Класс каждого конвертера принимает механизм создания и сборки сложного объекта и скрывает его за абстрактным интерфейсом. Конвертор отделен от загрузчика, который отвечает за синтаксический разбор RTFдокумента.

В паттерне строитель абстрагированы все эти отношения. В нем любой класс конвертера называется строителем, а загрузчик – распорядителем. В применении к рассмотренному примеру строитель отделяет алгоритм интерпретации формата текста (то есть анализатор RTF-документов) от того, как создается и представляется документ в преобразованном формате. Это позволяет повторно использовать алгоритм разбора, реализованный в RTFReader, для создания разных текстовых представлений RTF-документов; достаточно передать в RTFReader различные подклассы класса TextConverter.

Общая схема шаблона выглядит так:



Ещё один пример (опять же от Джошуа Блоха) применения шаблона Строитель -- борьба с так называемыми телескопическими конструкторами.

У конструкторов и статических методов есть одно общее ограничение: они плохо масштабируют большое количество необязательных параметров. Рассмотрим такой случай: класс, представляющий собой этикетку с информацией о питательности на упаковке с продуктами питания. На этих этикетках есть несколько обязательных полей -- размер порции, количество порций в упаковке, калорийность, а также ряд необязательных параметров -- общее содержание жиров, содержание насыщенных жиров, содержание холестерина, натрия и т.д. У большинства продуктов ненулевыми будут только несколько из этих необязательных значений.

Какой конструктор или какие методы нужно использовать для написания данного класса? Традиционно, программисты будут использовать набор конструкторов: конструктор с одними обязательными параметрами, конструктор с одним необязательным параметром, конструктор с двумя обязательными параметрами и т.д., до тех пор пока не будет конструктора со всеми необязательными параметрами. Вот как это выглядит на практике (для краткости мы будем использовать только 4 необязательных параметра):

```
    public class NutritionFacts {
        private final int servingSize; // (ml_) required
        private final int servings: // (per container) required
        private final int calories; // optional
        private final int fat; // (g) optional
        private final int sodium; // (mg) optional
        private final int carbohydrate; // (g) optional
        public NutritionFacts(int servingSize, int servings) {
            this(servingSize, servings, 0);
        }
        public NutritionFacts(int servingSize, int servings,
                              int calories) {
            this(servingSize, servings, calories, 0);
        }
        public NutritionFacts(int servingSize, int servings,
                              int calories, int fat) {
            this(servingSize, servings, calories, fat, 0);
        }
        public NutritionFacts(int servingSize, int servings,
                              int calories, int fat, int sodium) {
            this(servingSize, servings, calories, fat, sodium, 0
        }
        public NutritionFacts(int servingSize, int servings,
                              int calories, int fat, int sodium, int carbohydrate)
{
            this.servingSize = servingSize;
            this.servings = servings;
            this.calories = calories;
            this.fat = fat;
            this.sodium = sodium;
            this.carbohydrate = carbohydrate;
        }
    }
```

Обычно для вызова таких конструкторов требуется передавать множество параметров, которые вы не хотите устанавливать, но вы в любом случае вынуждены передать для них значение. Поскольку мы имеем только шесть параметров, может показаться, что это не так уж и плохо, но ситуация выходит из-под контроля, когда число параметров увеличивается.

Короче говоря, шаблоны телескопических конструкторов нормально работают, но становится трудно писать код программы-клиента, когда имеется много параметров, а еще труднее этот код читать. Читателю остается только гадать, что означают все эти значения, и нужно тщательно высчитывать позицию параметра, чтобы выяснить, к какому полю он относится. Длинные последовательности одинаково типизированных параметров могут приводить к тонким ошибкам. Если клиент случайно перепутает два из таких параметров, то компиляция будет успешной, но программа будет неправильно работать при выполнении.

Второй вариант, когда вы столкнулись с конструктором со многими параметрами, — это вариант, где вы вызываете конструктор без параметров, чтобы создать объект, а затем вызываете сеттеры для установки обязательных и всех интересующих необязательных параметров. Данный шаблон лишен недостатков шаблона телескопических конструкторов. Он прост, хотя и содержит большое количество слов, но получившийся код легко читается.

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

К сожалению, этот подход не лишен серьезных недостатков. Поскольку его конструкция разделена между несколькими вызовами, объект может находиться в неустойчивом состоянии частично из-за такой конструкции. У класса нет возможности принудительно обеспечить стабильность простой проверкой действительности параметров конструктора. Попытка использования объекта, если он находится в неустойчивом состоянии, может привести к ошибкам выполнения даже после удаления ошибки из кода, что создает трудности при отладке. Схожим недостатком является то, что подобный подход исключает возможность сделать класс неизменяемым, что требует дополнительных усилий со стороны программиста для обеспечения безопасности в многопоточной среде.

Помочь решить эти проблемы может как раз шаблон Строитель. Вместо непосредственного создания желаемого объекта клиент вызывает конструктор (или статический метод) со всеми необходимыми параметрами и получает объект Builder. Затем клиент вызывает сеттеры на этом объекте для установки всех интересующих параметров. Наконец, клиент вызывает метод build для генерации объекта, который будет являться неизменным. Строитель является статическим внутренним классом в классе, который он создает. Вот как это выглядит на практике:

```
public class NutritionFacts {
```

```
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        // Required parameters
        private final int servingSize;
        private final int servings;
        // Optional parameters - initialized to default values
        private int calories = 0;
        private int fat = 0;
        private int carbohydrate = 0;
        private int sodium = 0;
        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;
            this.servings = servings;
        }
        public Builder calories(int val)
        { calories = val; return this; }
        public Builder fat(int val)
        { fat = val; return this; }
        public Builder carbohydrate(int val)
        { carbohydrate = val; return this; }
        public Builder sodium(int val)
        { sodium = val; return this; }
        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    }

    private NutritionFacts(Builder builder) {
        servingSize = builder.servingSize;
        servings = builder.servings;
        calories = builder.calories;
        fat = builder.fat;
        sodium = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }
}
```

Обратите внимание, что NutritionFacts является неизменным и что все значения параметров по умолчанию находятся в одном месте. Сеттеры объекта-строителя возвращают сам этот строитель. Поэтому вызовы можно объединять в цепочку. Вот как выглядит код клиента:

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8).
        calories(100).sodium(35).carbohydrate(27).build();
```

Стоит ещё заметить, что совершенно необязательно конструировать объект сразу в одном месте в коде. Можно сохранять промежуточное состояние объекта-строителя и добавлять ему параметры по мере их поступления, создав целевой объект в самом конце.

Этот клиентский код легко писать и, что еще важнее, легко читать. А сам объект-строитель может проверять инварианты на свои параметры в методе build().

Итого, используйте паттерн строитель, когда:

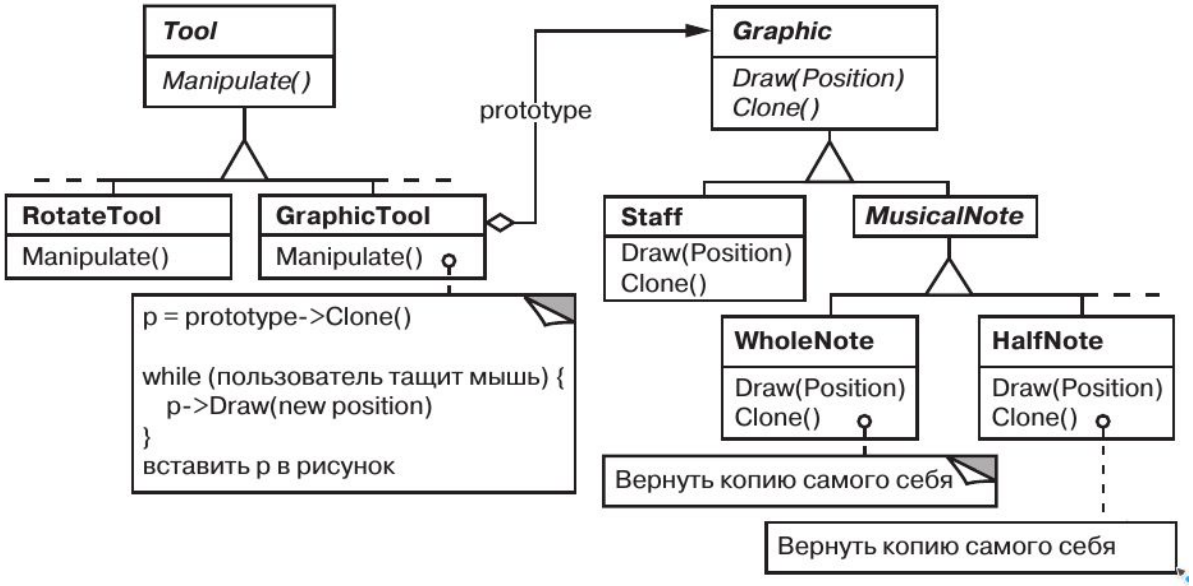
- алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой;
- процесс конструирования должен обеспечивать различные представления конструируемого объекта;
- хочется конструировать сложный объект по частям, сохраняя его целостность.

Обсуждение порождающих паттернов

Есть два наиболее распространенных способа параметризовать систему классами создаваемых ей объектов. Первый способ – порождение подклассов от класса, создающего объекты. Он соответствует паттерну фабричный метод. Основной недостаток метода: требуется создавать новый подкласс лишь для того, чтобы задать создание продукта. И таких изменений может быть очень много. Например, если создатель продукта сам создается фабричным методом, то придется замещать и создателя тоже.

Другой способ параметризации системы в большей степени основан на композиции объектов. Вы определяете объект, которому известно о классах объектов-продуктов, и делаете его параметром системы. Это ключевой аспект таких паттернов, как абстрактная фабрика, строитель и прототип. Для всех трех характерно создание «фабричного объекта», который изготавливает продукты. В абстрактной фабрике фабричный объект производит объекты разных классов. Фабричный объект строителя постепенно создает сложный продукт, следуя специальному протоколу. Фабричный объект прототипа изготавливает продукт путем копирования объекта-прототипа. В последнем случае фабричный объект и прототип – это одно и то же, поскольку именно прототип отвечает за возврат продукта.

Рассмотрим структуру фреймворка графических редакторов, описанного при обсуждении паттерна прототип.



Есть несколько способов параметризовать класс GraphicTool классом продукта:

- применить паттерн фабричный метод. Тогда для каждого подкласса класса Graphic в палитре будет создан свой подкласс GraphicTool. В классе GraphicTool будет присутствовать операция NewGraphic, переопределяемая каждым подклассом;
- использовать паттерн абстрактная фабрика. Возникнет иерархия классов GraphicsFactories, по одной для каждого подкласса Graphic. В этом случае каждая фабрика создает только один продукт: CircleFactory – окружности Circle, LineFactory – отрезки Line и т.д. GraphicTool параметризуется фабрикой для создания подходящих графических объектов;
- применить паттерн прототип. Тогда в каждом подклассе Graphic будет реализована операция Clone, а GraphicTool параметризуется прототипом создаваемого графического объекта.

Выбор паттерна зависит от многих факторов. В нашем примере фреймворка графических редакторов, на первый взгляд, проще всего воспользоваться фабричным методом. Определить новый подкласс GraphicTool легко, а экземпляры GraphicTool создаются только в момент определения палитры. Основной недостаток такого подхода заключается в комбинаторном росте числа подклассов GraphicTool, причем все они почти ничего не делают.

Абстрактная фабрика лишь немногим лучше, поскольку требует создания равновеликой иерархии классов GraphicsFactory. Абстрактную фабрику следует предпочесть фабричному методу лишь тогда, когда уже и так существует иерархия классов GraphicsFactory: либо потому, что ее автоматически строит компилятор (как в Smalltalk или Objective C), либо она необходима для другой части системы.

Очевидно, целям фреймворка графических редакторов лучше всего отвечает паттерн прототип, поскольку для его применения требуется лишь реализовать операцию Clone в каждом классе Graphics. Это сокращает число подклассов, а Clone можно с пользой применить и для решения других задач (например, для реализации пункта меню Duplicate), а не только для инстанцирования.

В случае применения паттерна фабричный метод проект в большей степени поддается настройке и оказывается лишь немногим более сложным. Другие паттерны нуждаются в создании новых классов, а фабричный метод – только в создании одной новой операции. Часто этот паттерн рассматривается как стандартный способ создания объектов, но вряд ли его стоит рекомендовать в ситуации, когда инстанцируемый класс никогда не изменяется или когда инстанцирование выполняется внутри операции, которую легко можно заместить в подклассах (например, во время инициализации).

Проекты, в которых используются паттерны абстрактная фабрика, прототип или строитель, оказываются еще более гибкими, чем те, где применяется фабричный метод, но за это приходится платить повышенной сложностью. Часто в начале работы над проектом за основу берется фабричный метод, а позже, когда проектировщик обнаруживает, что решение получается недостаточно гибким, он выбирает другие паттерны. Владение разными паттернами проектирования открывает перед вами широкий выбор при оценке различных критериев.