ЧАСТЬ 1 Лабораторная работа № 1. Введение в паттерны проектирования

Проектирование в рамках ООП - нелегкое дело, а если их нужно *использовать повторно,* то все становится еще сложнее. Необходимо подобрать подходящие объекты, отнести их к различным классам, соблюдая разумную степень детализации, определить интерфейсы классов и иерархию наследования и установить существенные отношения между классами. Дизайн должен, с одной стороны, соответствовать решаемой задаче, с другой - быть общим, чтобы удалось учесть все требования, которые могут возникнуть в будущем. Хотелось бы также избежать вовсе или, по крайней мере, свести к минимуму необходимость перепроектирования. Поднаторевшие в ОО-проектировании разработчики скажут вам, что обеспечить «правильный», то есть в достаточной мере гибкий и пригодный для повторного использования дизайн, с первого раза очень трудно, если вообще возможно. Прежде чем считать цель достигнутой, они обычно пытаются опробовать найденное решение на нескольких задачах, и каждый раз модифицируют его.

И все же опытным проектировщикам удается создать хороший дизайн системы. В то же время новички испытывают шок от числа возможных вариантов и нередко возвращаются к привычным не ОО-методикам. Проходит немало времени перед тем, как становится понятно, что же такое удачный ОО-дизайн. Опытные проектировщики, очевидно, знают какие-то тонкости, ускользающие от новичков. Так что же это?

Прежде всего, опытному разработчику понятно, что *не нужно* решать каждую новую задачу с нуля. Вместо этого он старается повторно воспользоваться теми решениями, которые оказались удачными в прошлом. Отыскав хорошее решение один раз, он будет прибегать к нему снова и снова. Именно благодаря накопленному опыту проектировщик и становится экспертом в своей области. Во многих объектно-ориентированных системах вы встретите повторяющиеся паттерны, состоящие из классов и взаимодействующих объектов. С их помощью решаются конкретные задачи проектирования, в результате чего ОО-дизайн становится более гибким, элегантным, и им можно воспользоваться повторно. Проектировщик, знакомый с паттернами, может сразу же применять их к решению новой задачи, не пытаясь каждый раз изобретать велосипед.

Поясним нашу мысль через аналогию. Писатели редко выдумывают совершенно новые сюжеты. Вместо этого они берут за основу уже отработанные схемы, жанры и образы. Например, трагический герой - Макбет, Гамлет и т.д., мотив убийства - деньги, месть, ревность и т.п. Точно так же в ОО-проектировании используются такие паттерны, как «представление состояния с помощью объектов» или «декорирование объектов, чтобы было проще добавлять и удалять их свойства».

Паттерны проектирования упрощают повторное использование удачных проектных и архитектурных решений. Представление прошедших проверку временем методик в виде паттернов проектирования облегчает доступ к ним со стороны разработчиков новых систем. С помощью паттернов можно улучшить качество документации и сопровождения существующих систем, позволяя явно описать взаимодействия классов и объектов, а также причины, по которым система была построена так, а не иначе. Проще говоря, паттерны проектирования дают разработчику возможность быстрее найти «правильный» путь.

1.1. Паттерн проектирования

По словам Кристофера Александра, «любой паттерн описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем таким образом, что это решение можно потом использовать миллион раз, ничего не изобретая заново». Хотя Александр имел в виду паттерны, возникающие при проектировании зданий и городов, но его слова верны и в отношении паттернов объектно-ориентированного проектирования. Наши решения выражаются замечания в терминах объектов и интерфейсов, а не стен и дверей, но в обоих случаях смысл паттерна - предложить решение определенной задачи в конкретном контексте. В общем случае паттерн состоит из четырех основных элементов:

1. *Имя.* Сославшись на него, можно сразу описать проблему проектирования, ее решения и их последствия. Присваивание паттернам имен позволяет проектировать на более высоком уровне абстракции. С помощью словаря паттернов можно вести обсуждение с коллегами, упоминать паттерны в документации, в тонкостях представлять дизайн системы. Нахождение хороших имен было одной из самых трудных задач при составлении каталога.
2. *Задача.* Описание того, когда следует применять паттерн. Необходимо сформулировать задачу и ее контекст. Может описываться конкретная проблема проектирования, например способ представления алгоритмов в виде объектов. Иногда отмечается, какие структуры классов или объектов свидетельствуют о негибком дизайне. Также может включаться перечень условий, при выполнении которых имеет смысл применять данный паттерн.
3. *Решение.* Описание элементов дизайна, отношений между ними, функций каждого элемента. Конкретный дизайн или реализация не имеются в виду, поскольку паттерн - это шаблон, применимый в самых разных ситуациях. Просто дается абстрактное описание задачи проектирования и того, как она может быть решена с помощью некоего весьма обобщенного сочетания элементов (в нашем случае классов и объектов).
4. *Результаты* - это следствия применения паттерна и разного рода компромиссы. Хотя при описании проектных решений о последствиях часто не упоминают, знать о них необходимо, чтобы можно было выбрать между различными вариантами и оценить преимущества и недостатки данного паттерна. Здесь речь идет и о выборе языка и реализации. Поскольку в объектно-ориентированном проектировании повторное использование зачастую является важным фактором, то к результатам следует относить и влияние на степень гибкости, расширяемости и переносимости системы. Перечисление всех последствий поможет вам понять и оценить их роль.

То, что один воспринимает как паттерн, для другого просто строительный блок. *Паттерны проектирования* — это не то же самое, что связанные списки или хэш-таблицы, которые можно реализовать в виде класса и повторно использовать без каких бы то ни было модификаций. Но это и не сложные, предметно-ориентированные решения для целого приложения или подсистемы. Здесь под паттернами проектирования понимается *описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте.*

Паттерн проектирования именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения, которые и позволяют применить его для создания повторно используемого дизайна. Он вычленяет участвующие классы и экземпляры, их роль и отношения, а также функции. При описании каждого паттерна внимание акцентируется на конкретной задаче объектно-ориентированного проектирования. Анализируется, когда следует применять паттерн, можно ли его использовать с учетом других проектных ограничений, каковы будут последствия **применения** метода. Поскольку любой проект в конечном итоге предстоит реализовывать, в состав паттерна включается пример кода на языке C++, иллюстрирующего реализацию.

Хотя паттерны используются в проектировании, они основаны на практических решениях, реализованных на языках ООП типа Smalltalk и C++, а не на процедурных (Pascal, C, Ada и т.п.) или OO-языках с динамической типизацией (CLOS, Dylan, Self).

1.2. Паттерны проектирования в схеме MVC в языке Smalltalk

В Smalltalk-80 для построения интерфейсов пользователя применяется тройка классов модель/вид/контроллер (Model/View/Controller - MVC) [KP88]. Знакомство с паттернами проектирования, встречающимися в схеме MVC, поможет вам разобраться в том, что понимаемается под словом «паттерн».

MVC состоит из объектов трех видов. *Модель* - это объект приложения, а *вид* - экранное представление. *Контроллер* описывает, как интерфейс реагирует на управляющие воздействия пользователя. До появления схемы MVC эти объекты в пользовательских интерфейсах смешивались. MVC отделяет их друг от друга, за счет чего повышается гибкость и улучшаются возможности повторного использования.

MVC отделяет вид от модели, устанавливая между ними протокол взаимодействия «подписка/оповещение». Вид должен гарантировать, что внешнее представление отражает состояние модели. При каждом изменении внутренних данных модель оповещает все зависящие от нее виды, в результате чего вид обновляет себя. Такой подход позволяет присоединить к одной модели несколько видов, обеспечив тем самым различные представления. Можно создать новый вид, не переписывая модель.

На рисунке ниже показана одна модель и три вида. (Для простоты опущены контроллеры.) Модель содержит некоторые данные, которые могут быть представлены в виде электронной таблицы, гистограммы и круговой диаграммы.

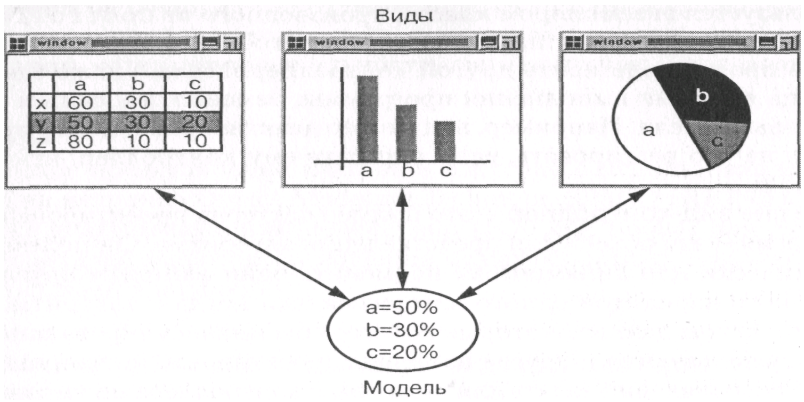

Рисунок 1.1. Модель MVC

Модель оповещает свои виды при каждом изменении значений данных, а виды обращаются к модели для получения новых значений.

На первый взгляд, в примере продемонстрирован просто дизайн, отделяющий вид от модели. Но этот принцип применим и к более общей задаче: разделение объектов таким образом, что изменение одного отражается сразу на нескольких других, причем изменившийся объект не имеет информации о деталях реализации объектов, на которые он оказал воздействие. Этот более общий подход описывается паттерном наблюдатель.

Еще одно свойство MVC заключается в том, что виды могут быть вложенными. Например, панель управления, состоящую из кнопок, допустимо представить как составной вид, содержащий вложенные, - по одной кнопке на каждый. Пользовательский интерфейс инспектора объектов может состоять из вложенных видов, используемых также и в отладчике. MVC поддерживает вложенные виды с помощью класса CompositeView, являющегося подклассом View. Объекты класса CompositeView ведут себя так же, как объекты класса View, поэтому могут использоваться всюду, где и виды. Но еще они могут содержать вложенные виды и управлять ими.

Здесь можно было бы считать, что этот дизайн позволяет обращаться с составным видом, как с любым из его компонентов. Но тот же дизайн применим и в ситуации, когда мы хотим иметь возможность группировать объекты и рассматривать группу как отдельный объект. Такой подход описывается паттерном компоновщик. Он позволяет создавать иерархию классов, в которой некоторые подклассы определяют примитивные объекты (например, Button - кнопка), а другие - составные объекты (CompositeView), группирующие примитивы в более сложные структуры.

MVC позволяет также изменять реакцию вида на действия пользователя. При этом визуальное

представление остается прежним. Например, можно изменить реакцию на нажатие клавиши или использовать всплывающие меню вместо командных клавиш. MVC инкапсулирует механизм определения реакции в объекте Controller. Существует иерархия классов контроллеров, и это позволяет без труда создать новый контроллер как вариант уже существующего.

Вид пользуется экземпляром класса, производного от Controller, для реализации конкретной стратегии реагирования. Чтобы реализовать иную стратегию, нужно просто подставить другой контроллер. Можно даже заменить контроллер вида во время выполнения программы, изменив тем самым реакцию на действия пользователя. Например, вид можно деактивировать, так что он вообще не будет ни на что реагировать, если передать ему контроллер, игнорирующий события ввода.

Отношение вид-контроллер - это пример паттерна проектирования стратегия. Стратегия - это объект для представления алгоритма. Он полезен, когда вы хотите статически или динамически подменить один алгоритм другим, если существует много вариантов одного алгоритма или когда с алгоритмом связаны сложные структуры данных, которые хотелось бы инкапсулировать.

В MVC используются и другие паттерны проектирования, например фабричный метод, позволяющий задать для вида класс контроллера по умолчанию, и декоратор для добавления к виду возможности прокрутки. Но основные отношения в схеме MVC описываются паттернами наблюдатель, компоновщик и стратегия.

1.3. Описание паттернов проектирования

Как описывать паттерны проектирования? Графических обозначений недостаточно. Они просто символизируют конечный продукт процесса проектирования в виде отношений между классами и объектами. Чтобы повторно воспользоваться дизайном, необходимо документировать решения, альтернативные варианты и компромиссы, которые привели к нему. Важны также конкретные примеры, поскольку они позволяют увидеть применение паттерна.

При описании паттернов проектировании придерживаются единого принципа. Описание каждого паттерна разбито на разделы, перечисленные ниже. Такой подход позволяет единообразно представить информацию, облегчает изучение, сравнение и применение паттернов.

Название и классификация паттерна

Название паттерна должно четко отражать его назначение.

Назначение

Лаконичный ответ на следующие вопросы: каковы функции паттерна, его обоснование и назначение, какую конкретную задачу проектирования можно решить с его помощью.

Известен также под именем

Другие распространенные названия паттерна, если таковые имеются.

Мотивация

Сценарий, иллюстрирующий задачу проектирования и то, как она решается данной структурой класса или объекта. Благодаря мотивации можно лучше понять последующее, более абстрактное описание паттерна.

Применимость

Описание ситуаций, в которых можно применять данный паттерн. Примеры проектирования, которые можно улучшить с его помощью. Распознавание такихситуаций.

Структура

Графическое представление классов в паттерне с использованием нотации, основанной на методике Object Modeling Technique (OMT). Также используются диаграммы взаимодействий для иллюстрации последовательностей запросов и отношений между объектами.

Участники

Классы или объекты, задействованные в данном паттерне проектирования, и их функции.

Отношения

Взаимодействие участников для выполнения своих функций.

Результаты

Насколько паттерн удовлетворяет поставленным требованиям? Результаты применения, компромиссы, на которые приходится идти. Какие аспекты поведения системы можно независимо изменять, используя данный паттерн?

Реализация

Сложности и так называемые подводные камни при реализации паттерна. Советы и рекомендуемые приемы. Есть ли у данного паттерна зависимость от языка программирования?

Пример кода

Фрагмент кода, иллюстрирующий вероятную реализацию на языке C++.

Известные применения

Возможности применения паттерна в реальных системах. Даются, по меньшей мере, два примера из различных областей.

Родственные паттерны

Связь других паттернов проектирования с данным. Важные различия. Использование данного паттерна в сочетании с другими.

2. Порождающие паттерны

Порождающие паттерны проектирования абстрагируют процесс инстанцирования. Они помогут сделать

систему независимой от способа создания, композиции и представления объектов. Паттерн, порождающий классы, использует наследование, чтобы варьировать инстанцируемый класс, а паттерн, порождающий объекты, делегирует инстанцирование другому объекту.

Эти паттерны оказываются важны, когда система больше зависит от композиции объектов, чем от наследования классов. Получается так, что основной упор делается не на жестком кодировании фиксированного набора поведений, а на определении небольшого набора фундаментальных поведений, с помощью композиции которых можно получать любое число более сложных. Таким образом, для создания объектов с конкретным поведением требуется нечто большее, чем простое инстанцирование класса.

Для порождающих паттернов актуальны две темы. Во-первых, эти паттерны инкапсулируют знания о конкретных классах, которые применяются в системе. Во-вторых, 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 паттерны 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).

Иногда допустимо выбирать между тем или иным порождающим паттерном. Например, есть случаи, когда с пользой для дела можно использовать как прототип, так и абстрактную фабрику. В других ситуациях порождающие паттерны дополняют друг друга. Так, применяя строитель, можно использовать другие паттерны для решения вопроса о том, какие компоненты нужно строить, а прототип часто реализуется вместе с одиночкой.

В качестве примера - построение лабиринта для компьютерной игры. Иногда целью игры станет просто отыскание выхода из лабиринта; тогда у игрока будет лишь один локальный вид помещения. В других случаях в лабиринтах могут встречаться задачки, которые игрок должен решить, и опасности, которые предстоит преодолеть. В подобных играх может отображаться карта того участка лабиринта, который уже был исследован.

Опустим детали того, что может встречаться в лабиринте и сколько игроков принимают участие в забаве, а сосредоточимся лишь на принципах создания лабиринта. Лабиринт определим как множество комнат. Любая комната «знает» о своих соседях, в качестве которых могут выступать другая комната, стена или дверь в другую комнату.

Классы Room (комната), Door (дверь) и Wall (стена) определяют компоненты лабиринта и используются во всех примерах. Определим только те части этих классов, которые важны для создания лабиринта. Не будем рассматривать игроков, операции отображения и блуждания в лабиринте и другие важные функции, не имеющие отношения к построению структуры.

На диаграмме ниже показаны отношения между классами Room, Door и Wall.

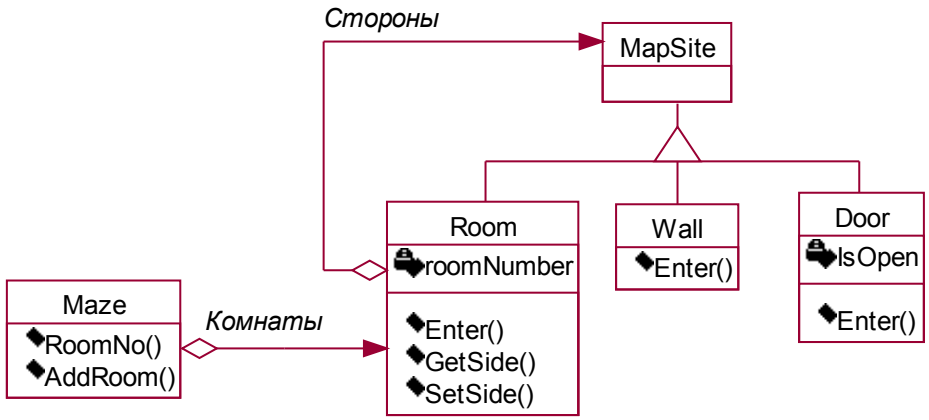

Рисунок 2.1. Структура лабиринта

У каждой комнаты есть четыре стороны. Для задания северной, южной, восточной и западной сторон будем использовать перечисление Direction в терминологии языка C++:

```
enum Direction {North, South, East, West};
```

Класс MapSite - это общий абстрактный класс для всех компонентов лабиринта. Определим в нем только одну операцию Enter. Когда вы входите в комнату, ваше местоположение изменяется. При попытке затем войти в дверь может произойти одно из двух. Если дверь открыта, то вы попадаете в следующую комнату. Если же дверь закрыта, то попасть в закрытую комнату без ключа невозможно и т.п.:

```
class MapSite {
 public:
virtual void Enter() = 0;
};
```

Операция Enter составляет основу для более сложных игровых операций. Например, если вы находитесь в комнате и говорите «Иду на восток», то игрой определяется, какой объект класса MapSite находится к востоку от вас, и для него вызывается операция Enter.

Определенные в подклассах операции Enter «выяснят», изменили вы направление или нет. В реальной игре Enter могла бы принимать в качестве аргумента объект, представляющий блуждающего игрока. Room - это конкретный подкласс класса MapSite, который определяет ключевые отношения между компонентами лабиринта. Он содержит ссылки на другие объекты MapSite, а также хранит номер комнаты. Номерами идентифицируются все

комнаты в лабиринте:

```
class Room : public MapSite {
public:
      Room(int roomNo);
      MapSite* GetSide(Direction) const;
      void SetSide(Direction, MapSite*);
      virtual void Enter();
private:
      MapSite* _sides[4];
      int _roomNumber;
};
```

Следующие классы представляют стены и двери, находящиеся с каждой стороны комнаты:

```
class Wall : public MapSite {
public:
      Wall();
      virtual void Enter();
};
class Door : public MapSite {
public:
      Door(Room* = 0, Room* = 0);
      virtual void Enter();
      Room* OtherSideFrom(Room*);
private:
      Room* _rooml ;
      Room* _room2;
      bool _isOpen;
};
```

Необходимо знать не только об отдельных частях лабиринта. Определим еще класс Maze для представления набора комнат. В этом классе есть операция RoomNo для нахождения комнаты по ее номеру:

```
class Maze {
public:
      Maze();

      void AddRoom(Room*);
      Room* RoomNo(int) const; private:
      // . . .
};
```

RoomNo могла бы реализовывать свою задачу с помощью линейного списка, хэш-таблицы или даже простого массива. Но пока нас не интересуют такие детали. Займемся тем, как описать компоненты объекта, представляющего лабиринт.

Определим также класс MazeGame, который создает лабиринт. Самый простой способ сделать это - строить лабиринт с помощью последовательности операций, добавляющих к нему компоненты, которые потом соединяются. Например, следующая функция-член создаст лабиринт из двух комнат с одной дверью между ними:

```
Maze*  MazeGame: : CreateMaze   ()
{
      Maze*  aMaze  = new Maze;
      Room*  rl  =  new Room(l);
      Room*  r2   = new Room(2);
      Door*  theDoor  = new Door(rl, r2);
      aMaze->AddRoom(rl);
      aMaze->AddRoom(r2);
      rl->SetSide (North, new Wall) ;
      rl->SetSide(East,   theDoor);
      rl->SetSide (South, new Wall);
      rl->SetSide(West,   new Wall);
      r2->SetSide (North, new Wall);
      r2->SetSide(East,   new Wall);
      r2->SetSide (South, new Wall);
      r2->SetSide(West,   theDoor);

      return aMaze;
}
```

Довольно сложная функция, если принять во внимание, что она всего лишь создает лабиринт из двух комнат. Есть очевидные способы упростить ее. Например, конструктор класса Room мог бы инициализировать стороны без дверей заранее. Но это означает лишь перемещение кода в другое место. Суть проблемы не в размере этой функции-члена, а в ее негибкости. В функции жестко «зашита» структура лабиринта. Чтобы изменить структуру, придется изменить саму функцию, либо заместив ее (то есть полностью переписав заново), либо

непосредственно модифицировав ее фрагменты. Оба пути чреваты ошибками и не способствуют повторному использованию.

Порождающие паттерны показывают, как сделать дизайн более гибким, хотя и необязательно меньшим по размеру. В частности, их применение позволит легко менять классы, определяющие компоненты лабиринта.

Предположим, что мы хотим использовать уже существующую структуру в новой игре с волшебными лабиринтами. В такой игре появляются не существовавшие ранее компоненты, например DoorNeedingSpell - запертая дверь, для открывания которой нужно произнести заклинание, или EnchantedRoom - комната, где есть необычные предметы, скажем, волшебные ключи или магические слова. Как легко изменить операцию CreateMaze, чтобы она создавала лабиринты с новыми классами объектов?

Самое серьезное препятствие лежит в жестко зашитой в код информации о том, какие классы инстанцируются. С помощью порождающих паттернов можно различными способами избавиться от явных ссылок на конкретные классы из кода, выполняющего их инстанцирование:

- если CreateMaze вызывает виртуальные функции вместо конструкторов для создания комнат, стен и дверей, то инстанцируемые классы можно подменить, создав подкласс MazeGame и переопределив в нем виртуальные функции. Такой подход применяется в паттерне фабричный метод;
- когда функции CreateMaze в качестве параметра передается объект, используемый для создания комнат, стен и дверей, то их классы можно изменить, передав другой параметр. Это пример паттерна абстрактная фабрика;
- если функции CreateMaze передается объект, способный целиком создать новый лабиринт с помощью своих операций для добавления комнат, дверей и стен, можно воспользоваться наследованием для изменения частей лабиринта или способа его построения. Такой подход применяется в паттерне строитель;
- если CreateMaze параметризована прототипами комнаты, двери и стены, которые она затем копирует и добавляет к лабиринту, то состав лабиринта можно варьировать, заменяя одни объекты-прототипы другими. Это паттерн прототип.

Последний из порождающих паттернов, одиночка, может гарантировать наличие единственного лабиринта в игре и свободный доступ к нему со стороны всех игровых объектов, не прибегая к глобальным переменным или функциям. Одиночка также позволяет легко расширить или заменить лабиринт, не трогая существующий код.

3. Паттерн Abstract Factory

3.1. Название и классификация паттерна

Абстрактная фабрика - паттерн, порождающий объекты.

3.2. Назначение

Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

3.3. Известен также под именем

Kit (инструментарий).

3.4. Мотивация

Рассмотрим инструментальную программу для создания пользовательского интерфейса, поддерживающего разные стандарты внешнего облика, например Motif и Presentation Manager. Внешний облик определяет визуальное представление и поведение элементов пользовательского интерфейса («виджетов») - полос прокрутки, окон и кнопок. Чтобы приложение можно было перенести на другой стандарт, в нем не должен быть жестко закодирован внешний облик виджетов. Если инстанцирование классов для конкретного внешнего облика разбросано по всему приложению, то изменить облик впоследствии будет нелегко.

Можно решить эту проблему, определив абстрактный класс WidgetFactory, в котором объявлен интерфейс для создания всех основных видов виджетов. Есть также абстрактные классы для каждого отдельного вида и конкретные подклассы, реализующие виджеты с определенным внешним обликом. В интерфейсе WidgetFactory имеется операция, возвращающая новый объект-виджет для каждого абстрактного класса виджетов. Клиенты вызывают эти операции для получения экземпляров виджетов, но при этом ничего не знают о том, какие именно классы используют. Стало быть, клиенты остаются независимыми от выбранного стандарта внешнего облика.

Для каждого стандарта внешнего облика существует определенный подкласс WidgetFactory. Каждый такой подкласс реализует операции, необходимые для создания соответствующего стандарту виджета. Например, операция CreateScrollBar в классе Motif WidgetFactory инстанцирует и возвращает полосу прокрутки в стандарте Motif, тогда как соответствующая операция в классе PMWidgetFactory возвращает полосу прокрутки в стандарте Presentation Manager. Клиенты создают виджеты, пользуясь исключительно интерфейсом WidgetFactory, и им ничего не известно о классах, реализующих виджеты для конкретного стандарта. Другими словами, клиенты должны лишь придерживаться интерфейса, определенного абстрактным, а не конкретным классом.

Класс WidgetFactory также устанавливает зависимости между конкретными классами виджетов. Полоса прокрутки для Motif должна использоваться с кнопкой и текстовым полем Motif, и это ограничение поддерживается автоматически, как следствие использования класса MotifWidgetFactory.

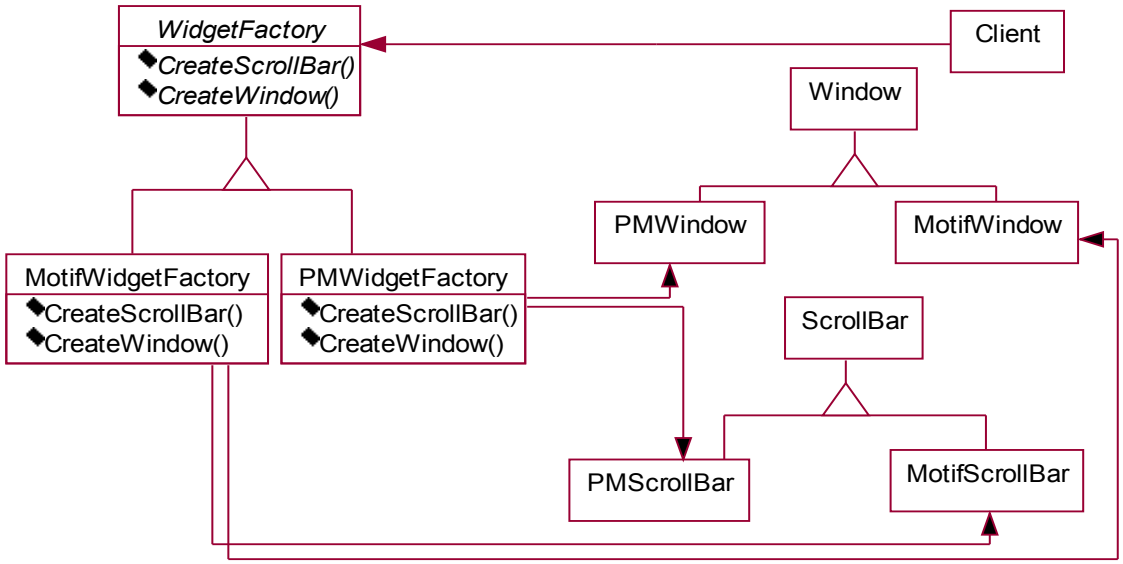

Рисунок 3.1. Пример применения паттерна «абстрактная фабрика»

3.5. Применимость

Используйте паттерн абстрактная фабрика, когда:

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

3.6. Структура

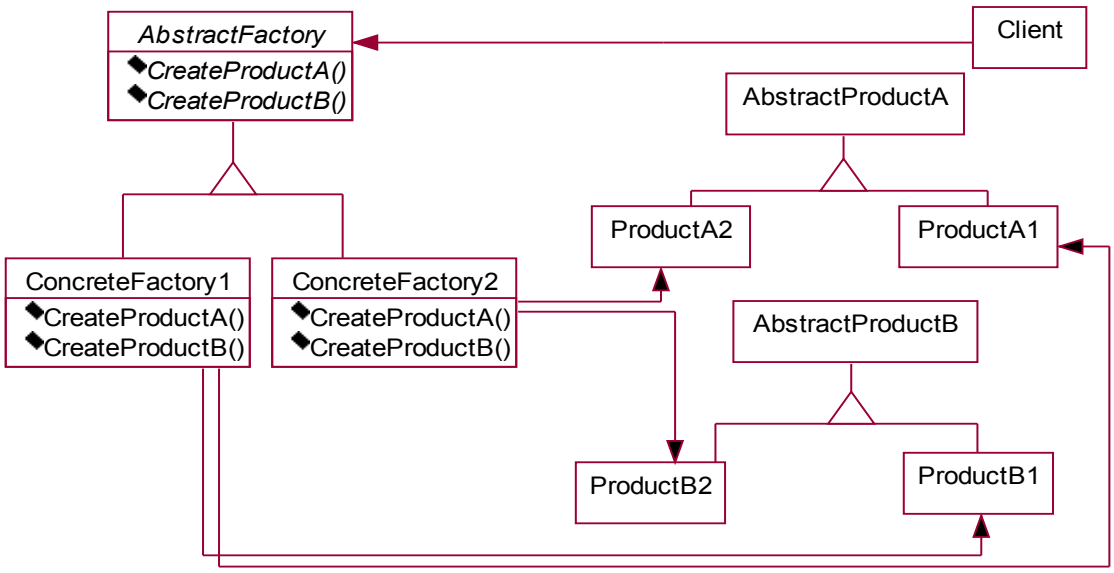

Рисунок 3.2. Структура паттерна «абстрактная фабрика»

3.7. Участники

- **AbstractFactory** (WidgetFactory) - абстрактная фабрика:
  - объявляет интерфейс для операций, создающих абстрактные объекты-продукты;
- **ConcreteFactory** (Motif Widget Factory, PMWidgetFactory) - конкретная фабрика:
  - реализует операции, создающие конкретные объекты-продукты;
- **AbstractProduct** (Window, ScrollBar) - абстрактный продукт:
  - объявляет интерфейс для типа объекта-продукта;
- **ConcreteProduct** (Mot if Window, Motif ScrollBar) - конкретный продукт:

- определяет объект-продукт, создаваемый соответствующей конкретной фабрикой;
- реализует интерфейс Abstract Product; а Client - клиент:
- пользуется исключительно интерфейсами, которые объявлены в классах AbstractFactory и AbstractProduct.

3.8. Отношения

- Обычно во время выполнения создается единственный экземпляр класса ConcreteFactory. Эта конкретная фабрика создает объекты-продукты, имеющие вполне определенную реализацию. Для создания других видов объектов клиент должен воспользоваться другой конкретной фабрикой;
- AbstractFactory передоверяет создание объектов-продуктов своему подклассу ConcreteFactory.

3.9. Результаты

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 виджетов Motif к виджетам Presentation Manager можно, просто переключившись на продукты соответствующей фабрики и заново создав 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. Решение этой проблемы рассматривается в разделе «Реализация».

3.10. Реализация

Вот некоторые полезные приемы реализации паттерна абстрактная фабрика:

- *фабрики как объекты, существующие в единственном экземпляре.* Как правило, приложению нужен только один экземпляр класса ConcreteFactory на каждое семейство продуктов. Поэтому для реализации лучше всего применить паттерн одиночка;
- *создание продуктов.* Класс AbstractFactory объявляет только интерфейс для создания продуктов. Фактическое их создание - дело подклассов ConcreteProduct. Чаще всего для этой цели определяется фабричный метод для каждого продукта (паттерн фабричный метод). Конкретная фабрика специфицирует свои продукты путем замещения фабричного метода для каждого из них. Хотя такая реализация проста, она требует создавать новый подкласс конкретной фабрики для каждого семейства продуктов, даже если они почти ничем не отличаются.

Если семейств продуктов может быть много, то конкретную фабрику удастся реализовать с помощью паттерна прототип. В этом случае она инициализируется экземпляром-прототипом каждого продукта в семействе и создает новый продукт путем клонирования этого прототипа. Подход на основе прототипов устраняет необходимость создавать новый класс конкретной фабрики для каждого нового семейства продуктов.

В языках, где сами классы являются настоящими объектами (например, Smalltalk и Objective C), возможны некие вариации подхода на базе прототипов. В таких языках класс можно представлять себе как вырожденный случай фабрики, умеющей создавать только один вид продуктов. Можно хранить *классы* внутри конкретной фабрики, которая создает разные конкретные продукты в переменных. Это очень похоже на прототипы. Такие классы создают новые экземпляры от имени конкретной фабрики. Новая фабрика инициализируется экземпляром конкретной фабрики с *классами* продуктов, а не путем порождения подкласса. Подобный подход задействует некоторые специфические свойства языка, тогда как подход, основанный на прототипах, от языка не зависит.

- *определение расширяемых фабрик.* Класс AbstractFactory обычно определяет разные операции для каждого вида изготавливаемых продуктов. Виды продуктов кодируются в сигнатуре операции. Для добавления нового вида продуктов нужно изменить интерфейс класса AbstractFactory и всех зависящих от него классов.

Более гибкий, но не такой безопасный способ - добавить параметр к операциям, быть идентификатор класса, целое число, строка или что-то еще, однозначно описывающее вид продукта. При таком подходе классу AbstractFactory нужна только одна операция Make с параметром, указывающим тип создаваемого объекта. Данный прием применялся в обсуждавшихся выше абстрактных фабриках на основе прототипов и классов. Такой вариант проще использовать в динамически типизированных языках вроде Smalltalk, нежели в статически типизированных, каким является C++. Воспользоваться им в C++ можно только, если у всех

объектов имеется общий абстрактный базовый класс или если объекты-продукты могут быть безопасно приведены к корректному типу клиентом, который их запросил.Но даже если приведение типов не нужно, остается принципиальная проблема: все продукты возвращаются клиенту одним и тем же абстрактным интерфейсом с уже определенным типом возвращаемого значения. Клиент не может ни различить классы продуктов, ни сделать какие-нибудь предположения о них. Если клиенту нужно выполнить операцию, зависящую от подкласса, то она будет недоступна через абстрактный интерфейс. Хотя клиент мог бы выполнить динамическое приведение типа (например, с помощью оператора dynamic__cast в C++), это небезопасно и необязательно заканчивается успешно. Здесь мы имеем классический пример компромисса между высокой степенью гибкости и расширяемостью интерфейса.

3.11. Пример кода

Паттерн абстрактная фабрика мы применим к построению обсуждавшихся в начале этой главы лабиринтов.

Класс MazeFactory может создавать компоненты лабиринтов. Он строит комнаты, стены и двери между комнатами. Им разумно воспользоваться из программы, которая считывает план лабиринта из файла, а затем создает его, или из приложения, строящего случайный лабиринт. Программы построения лабиринта принимают MazeFactory в качестве аргумента, так что программист может сам указать классы комнат, стен и дверей:

```
class MazeFactory  {
 public:
      MazeFactory();
      virtual Maze* MakeMaze()   const
      {  return new Maze;   }
      virtual Wall* MakeWall()   const
      {   return new Wall;   }
      virtual Room* MakeRoom(int n)   const
      {  return new Room(n);   }
      virtual Door* MakeDoor(Room*  rl,   Room* ,r2)
      {  return new Door(rl,   r2);   }
```

Напомним, что функция-член CreateMaze строит небольшой лабиринт, состоящий всего из двух комнат, соединенных одной дверью. В ней жестко «зашиты» имена классов, поэтому воспользоваться функцией для создания лабиринтов с другими компонентами проблематично.

Вот версия CreateMaze, в которой нет подобного недостатка, поскольку она принимает MazeFactory в качестве параметра:

```
Maze* MazeGame::CreateMaze   (MazeFactoryk  factory)
{
      Maze*  aMaze  =  factory.MakeMaze();
      Room*  rl  =   factory.MakeRoom(l);
      Room*  r2  =   factory.MakeRoom(2);
      Door*  aDoor  =  factory.MakeDoor(rl,   r2);

      aMaze->AddRoom(rl); aMaze->AddRoom(r2);
      rl->SetSide(North,   factory.MakeWall());
      rl->SetSide(East,   aDoor);
      rl->SetSide(South,   factory.MakeWall());
      rl->SetSide(West,   factory.MakeWall());
      r2->SetSide(North,   factory.MakeWall());
      r2->SetSide(East,   factory.MakeWall());
      r2->SetSide (South,   factory .MakeWall ());
      r2->SetSide(West,   aDoor);

      return aMaze;
}
```

Можно создать фабрику EnchantedMazeFactory для производства волшебных лабиринтов, породив подкласс от MazeFactory. В этом подклассе заме-Щены различные функции-члены, так что он возвращает другие подклассы классов Room, Wall и т.д.:

```
class EnchantedMazeFactory : public MazeFactory {
 public:
      EnchantedMazeFactory();

      virtual Room* MakeRoom(int n) const
      { return new EnchantedRoom(n, CastSpell()); }

      virtual Door* MakeDoor(Room* rl, Room* r2) const
      { return new DoorNeedingSpell(rl, r2); }
 protected:
      Spell* CastSpell() const;
```

```
};
```

А теперь предположим, что необходимо построить для некоторой игры лабиринт, в одной из комнат которого заложена бомба. Если бомба взрывается, то она как минимум обрушивает стены. Тогда можно породить от класса Room подкласс, отслеживающий, есть ли в комнате бомба и взорвалась ли она. Также нам понадобится подкласс класса Wall, который хранит информацию о том, был ли нанесен ущерб стенам. Назовем эти классы соответственно RoomWithABomb и BombedWall.

И наконец, мы определим класс BombedMazeFactory, являющийся подклассом BombedMazeFactory, который создает стены класса BombedWall и комнаты класса RoomWithABomb. В этом классе надо переопределить всего две функции:

```
Wall* BombedMazeFactory::MakeWall () const
{
 return new BombedWall;
}
Room* BombedMazeFactory::MakeRoom(int n) const
{
 return new RoomWithABomb(n);
}
```

Для того чтобы построить простой лабиринт, в котором могут быть спрятаны бомбы, просто надо осуществить вызов функции CreateMaze, передав ей в качестве параметра BombedMazeFactory:

```
MazeGame game;
BombedMazeFactory  factory;
game.CreateMaze(factory);
```

Для построения волшебных лабиринтов CreateMaze может принимать в качестве параметра и EnchantedMazeFactory.

Отметим, что MazeFactory - всего лишь набор фабричных методов. Это самый распространенный способ реализации паттерна абстрактная фабрика. Еще заметим, что MazeFactory - не абстрактный класс, то есть он работает и как AbstractFactory, и как ConcreteFactory. Это еще одна типичная реализация для простых применений паттерна абстрактная фабрика. Поскольку MazeFactory - конкретный класс, состоящий только из фабричных методов, легко получить новую фабрику MazeFactory, породив подкласс и заместив в нем необходимые операции.

В функции CreateMaze используется операция SetSide для описания сто-пон комнат. Если она создает комнаты с помощью фабрики BombedMazeFactory, то лабиринт будет составлен из объектов класса RoomWithABomb, стороны которых описываются объектами класса BombedWall. Если классу RoomWithABomb потребуется доступ к членам BombedWall, не имеющим аналога в его предках, то придется привести ссылку на объекты-стены от типа Wall* к типу BombedWall*. Такое приведение к типу подкласса безопасно при условии, что аргумент действительно принадлежит классу BombedWall*, а это обязательно так, если стены создаются исключительно фабрикой BombedMazeFactory.

3.12. Известные применения

В библиотеке Interviews для обозначения классов абстрактных фабрик используется суффикс «Kit». Так, для изготовления объектов пользовательского интерфейса с заданным внешним обликом определены абстрактные фабрики WidgetKit HDialogKit. В Interviews есть также класс Lay out Kit, который генерирует разные объекты композиции в зависимости от того, какая требуется стратегия размещения. Например, размещение, которое концептуально можно было бы назвать «в строку», может потребовать разных объектов в зависимости от ориентации документа (книжной или альбомной).

В библиотеке ET++ паттерн абстрактная фабрика применяется для достижения переносимости между разными оконными системами (например, X Windows и SunView). Абстрактный базовый класс WindowSystem определяет интерфейс для создания объектов, которое представляют ресурсы оконной системы (MakeWindow, MakeFont, MakeColor и т.п.). Его конкретные подклассы реализуют эти интерфейсы для той или иной оконной системы. Во время выполнения ET++ создает экземпляр конкретного подкласса WindowSystem, который уже и порождает объекты, соответствующие ресурсам данной оконной системы.

3.13. Родственные паттерны

Классы AbstractFactory часто реализуются фабричными методами (см. паттерн фабричный метод), но могут быть реализованы и с помощью паттерна прототип.

Конкретная фабрика часто описывается паттерном одиночка.

1. Паттерн Prototype

1.1. Название и классификация паттерна

Прототип - паттерн, порождающий объекты.

1.2. Назначение

Задает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем

копирования этого прототипа.

1.3. Мотивация

Построить музыкальный редактор удалось бы путем адаптации общего каркаса графических редакторов и добавления новых объектов, представляющих ноты, паузы и нотный стан. В каркасе 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, двигал бы ноту на стане вверх или вниз, чтобы изменить ее высоту.

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

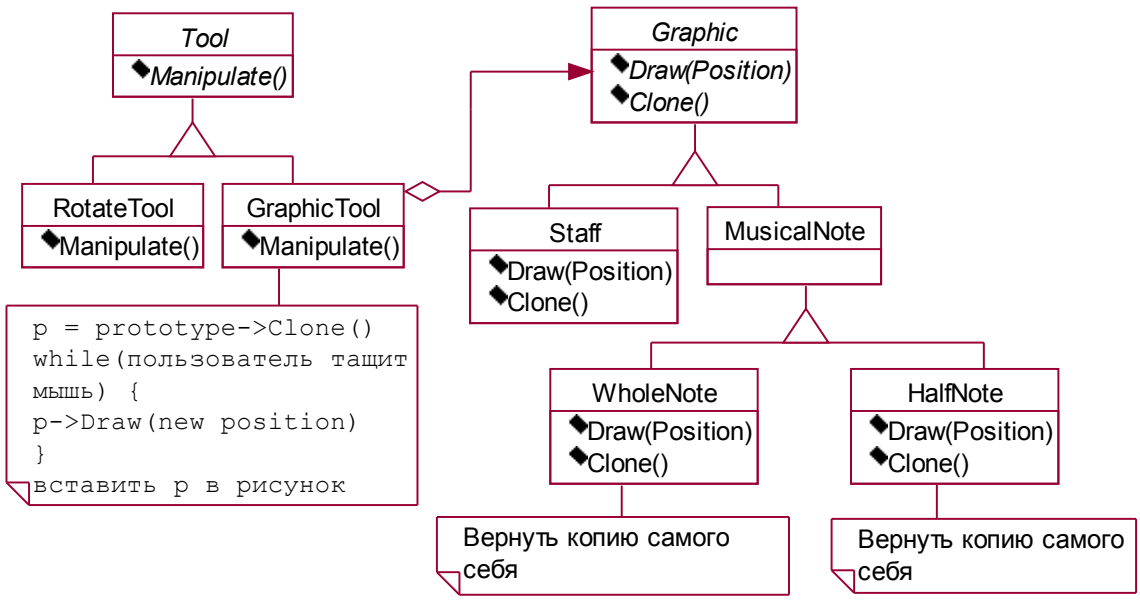

Рисунок 1.1. Пример применения паттерна «прототип»

Однако класс GraphicTool создает некую проблему для проектировщика каркаса. Классы нот и нотных станов специфичны для нашего приложения, а класс GraphicTool принадлежит каркасу. 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. Гибкой альтернативой порождению подклассов является композиция. Вопрос в том, как каркас 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.

Можно воспользоваться паттерном прототип, чтобы еще больше сократить число классов. Для целых и половинных нот у нас есть отдельные классы, но, быть может, это излишне. Вместо этого они могли бы быть экземплярами одного и того же класса, инициализированного разными растровыми изображениями и длительностями звучания. Инструмент для создания целых нот становится просто объектом класса GraphicTool, в котором прототип MusicalNote инициализирован целой нотой. Это может значительно уменьшить число классов в системе. Заодно упрощается добавление нового вида нот в музыкальный редактор.

1.4. Применимость

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.

1.5. Структура

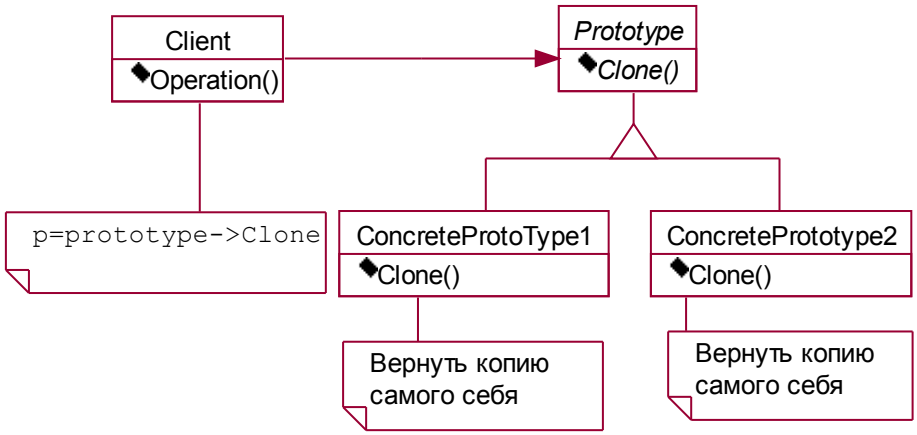

Рисунок 1.1. Структура паттерна «прототип»

1.6. Участники

- **Prototype** (Graphic) - прототип:
  - объявляет интерфейс для клонирования самого себя;
- **ConcretePrototype** (Staff- нотный стан, WholeNote - целая нота, Half Note - половинная нота) - конкретный прототип:
  - реализует операцию клонирования себя; а **Client** (GraphicTool) - клиент:
  - создает новый объект, обращаясь к прототипу с запросом клонировать себя.

1.7. Отношения

Клиент обращается к прототипу, чтобы тот создал свою копию.

1.8. Результаты

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.

Ниже перечислены дополнительные преимущества паттерна прототип:

- *добавление и удаление продуктов во время выполнения.* 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;
- *спецификация новых объектов путем изменения значений.* Динамичные системы позволяют определять поведение за счет композиции объектов - например, путем задания значений переменных объекта, - а не с помощью определения новых классов. По сути дела, вы определяете новые виды объектов, инстанцируя уже существующие классы и регистрируя их экземпляры как прототипы клиентских объектов. Клиент может изменить поведение, делегируя свои обязанности прототипу. Такой дизайн позволяет пользователям определять новые классы без программирования. Фактически клонирование объекта аналогично инстанцированию класса. Паттерн прототип может резко уменьшить число необходимых системе классов. В нашем музыкальном редакторе с помощью одного только класса GraphicTool удастся создать бесконечное разнообразие музыкальных объектов;
- *специфицирование новых объектов путем изменения структуры.* Многие приложения строят объекты из крупных и мелких составляющих. Например, редакторы для проектирования печатных плат создают электрические схемы из подсхем (*Для таких приложений характерны паттерны компоновщик и декоратор*). Такие приложения часто позволяют инстанцировать сложные, определенные пользователем структуры, скажем, для многократного использования некоторой подсхемы. Паттерн прототип поддерживает и такую возможность. Мы просто добавляем подсхему как прототип в палитру доступных элементов схемы. При условии, что объект, представляющий составную схему, реализует операцию Clone как глубокое копирование, схемы с разными структурами могут выступать в качестве прототипов;
- *уменьшение числа подклассов.* Паттерн фабричный метод часто порождает иерархию классов Creator, параллельную иерархии классов продуктов. Прототип позволяет клонировать прототип, а не запрашивать фабричный метод создать новый объект. Поэтому иерархия класса Creator становится вообще ненужной. Это преимущество касается главным образом языков типа C++, где классы не рассматриваются как настоящие объекты. В языках же типа Smalltalk и Objective C это не так существенно, поскольку всегда можно использовать объект-класс в качестве создателя. В таких языках объекты-классы уже выступают как прототипы;
- *динамическое конфигурирование приложения классами.* Некоторые среды позволяют динамически

загружать классы в приложение во время его выполнения. Паттерн прототип - это ключ к применению таких возможностей в языке типа C++.

Приложение, которое создает экземпляры динамически загружаемого класса, не может обращаться к его конструктору статически. Вместо этого исполняющая среда автоматически создает экземпляр каждого класса в момент его загрузки и регистрирует экземпляр в диспетчере прототипов (см. раздел «Реализация»). Затем приложение может запросить у диспетчера прототипов экземпляры вновь загруженных классов, которые изначально не были связаны с программой. Каркас приложений ET++ в своей исполняющей среде использует именно такую схему.

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.

1.9. Реализация

Прототип особенно полезен в статически типизированных языках вроде C++, где классы не являются объектами, а во время выполнения информации о типе недостаточно или нет вовсе. Меньший интерес данный паттерн представляет для таких языков, как Smalltalk или Objective C, в которых и так уже есть нечто эквивалентное прототипу (именно - объект-класс) для создания экземпляров каждого класса. В языки, основанные на прототипах, например Self, где создание любого объекта выполняется путем клонирования прототипа, этот паттерн просто встроен.

- Рассмотрим основные вопросы, возникающие при реализации прототипов: а *использование диспетчера прототипов.* Если число прототипов в системе не фиксировано (то есть они могут создаваться и уничтожаться динамически), ведите реестр доступных прототипов. Клиенты должны не управлять прототипами самостоятельно, а сохранять и извлекать их из реестра. Клиент запрашивает прототип из реестра перед его клонированием. Такой реестр мы будем называть *диспетчером прототипов.* Диспетчер прототипов - это ассоциативное хранилище, которое возвращает прототип, соответствующий заданному ключу. В нем есть операции для регистрации прототипа с указанным ключом и отмены регистрации. Клиенты могут изменять и даже «просматривать» реестр во время выполнения, а значит, расширять систему и вести контроль над ее состоянием без написания кода;
- *реализация операции Clone.* Самая трудная часть паттерна прототип - правильная реализация операции Clone. Особенно сложно это в случае, когда в структуре объекта есть круговые ссылки. В большинстве языков имеется некоторая поддержка для клонирования объектов. Например, Smalltalk предоставляет реализацию копирования, которую все подклассы наследуют от класса Object. В C++ есть копирующий конструктор. Но эти средства не решают проблему «глубокого и поверхностного копирования». Суть ее в следующем: должны ли при клонировании объекта клонироваться также и его переменные экземпляра или клон просто разделяет с оригиналом эти переменные? Поверхностное копирование просто, и часто его бывает достаточно. Именно такую возможность и предоставляет по умолчанию Smalltalk. В C++ копирующий конструктор по умолчанию выполняет почленное копирование то есть указатели разделяются копией и оригиналом. Но для клонирования прототипов со сложной структурой обычно необходимо глубокое копирование, поскольку клон должен быть независим от оригинала. Поэтому нужно гарантировать, что компоненты клона являются клонами компонентов прототипа. При клонировании вам приходится решать, что именно может разделяться и может ли вообще.
  Если объекты в системе предоставляют операции Save (сохранить) и Load (загрузить), то разрешается воспользоваться ими для реализации операции Clone по умолчанию, просто сохранив и сразу же загрузив объект. Операция Save сохраняет объект в буфере памяти, а Load создает дубликат, реконструируя объект из буфера;
- *инициализация клонов.* Хотя некоторым клиентам вполне достаточно клона как такового, другим нужно инициализировать его внутреннее состояние полностью или частично. Обычно передать начальные значения операции Clone невозможно, поскольку их число различно для разных классов прототипов. Для некоторых прототипов нужно много параметров инициализации, другие вообще ничего не требуют. Передача Clone параметров мешает построению единообразного интерфейса клонирования. Может оказаться, что в ваших классах прототипов уже определяются операции для установки и очистки некоторых важных элементов состояния. Если так, то этими операциями можно воспользоваться сразу после клонирования. В противном случае, возможно, понадобится ввести операцию Initialize (см. раздел «Пример кода»), которая принимает начальные значения в качестве аргументов и соответственно устанавливает внутреннее состояние клона. Будьте осторожны, если операция Clone реализует глубокое копирование: копии может понадобиться удалять (явно или внутри Initialize) перед повторной инициализацией.

1.10. Пример кода

Определим подкласс MazePrototypeFactory класса MazeFactory. Этот подкласс будет инициализироваться прототипами объектов, которые ему предстоит создавать, поэтому нам не придется порождать подклассы только ради изменения классов создаваемых стен или комнат.

MazePrototypeFactory дополняет интерфейс MazeFactory конструктором, принимающим в качестве аргументов прототипы:

```
class MazePrototypeFactory   :   public MazeFactory
```

```
{
public:
      MazePrototypeFactory(Maze*,   Wall*,   Room*,   Door*);
      virtual Maze* MakeMaze()   const;
      virtual  Room*  MakeRoom(int)   const;
      virtual Wall* MakeWall() const;
      virtual Door* MakeDoor(Room*, Room*) const;
private:
      Maze* _prototypeMaze;
      Room* _prototypeRoom;
      Wall* _prototypeWall;
      Door* _prototypeDoor;
};
```

Новый конструктор просто инициализирует свои прототипы:

```
MazePrototypeFactory::MazePrototypeFactory   ( Maze*  m,   Wall*  w,   Room*  r,   Door*
d)
{
      _prototypeMaze = m;
      _prototypeWall = w;
      _prototypeRoom = r;
      _prototypeDoor = d;
}
```

Функции-члены для создания стен, комнат и дверей похожи друг на друга: каждая клонирует, а затем инициализирует прототип. Вот определения функций MakeWall и MakeDoor:

```
Wall*  MazePrototypeFactory::MakeWall   ()   const
{
      return _prototypeWall->Clone();
}
Door*  MazePrototypeFactory::MakeDoor   (Room*  rl,   Room *r2)   const
{
      Door*  door  = _prototypeDoor->Clone();
      door->Initialize(rl,   r2) ;
      return door;
}
```

Можно применить MazePrototypeFactory для создания прототипичного или принимаемого по умолчанию лабиринта, просто инициализируя его прототипами базовых компонентов:

```
MazeGame game;
MazePrototypeFactory simpleMazeFactory(
new Maze,   new Wall,   new Room,   new Door );
Maze*  maze  = game.CreateMaze(simpleMazeFactory);
```

Для изменения типа лабиринта инициализируем MazePrototypeFactory Другим набором прототипов. Следующий вызов создает лабиринт с дверью типа BombedDoor и комнатой типа RoomWithABomb:

```
MazePrototypeFactory bombedMazeFactory( new Maze,    new BombedWall, new RoomWithABomb,
new Door );
```

Объект, который предполагается использовать в качестве прототипа, например экземпляр класса Wall, должен поддерживать операцию Clone. Кроме того, у него должен быть копирующий конструктор для клонирования. Также может потребоваться операция для повторной инициализации внутреннего состояния. Мы добавим в класс Door операцию Initialize, чтобы дать клиентам возможность инициализировать комнаты клона.

```
class  Door   :   public MapSite
{
public:
      Door();
      Door(const Door&);
      virtual void Initialize(Room*, Room*);
      virtual Door* Clone() const;
      virtual void Enter();
      Room* OtherSideFrom(Room*);
private:
      Room* _rooml;
      Room* _room2;
};
Door::Door (const Door& other)
{
      _room1 = other._rooml;
      _room2 = other._room2;
}
void Door::Initialize (Room* rl, Room* r2)
```

```
{
      _room1 = r1;
      _room2 = r2;
}
Door* Door::Clone () const
{
      return new Door(*this);
}
```

Подкласс BombedWall должен заместить операцию Clone и реализовать соответствующий копирующий конструктор:

```
class BombedWall : public Wall
{
public:
      BombedWall();
      BombedWall(const BombedWall&);
      virtual Wall* Clone() const;
      bool HasBomb();
private:
      bool _bomb;
};
BombedWall::BombedWall (const BombedWall& other) : Wall(other)
{
      _bomb = other._bomb;
}
Wall* BombedWall::Clone () const
{
      return new BombedWall(*this);
}
```

Операция BombedWall:: Clone возвращает Wall*, а ее реализация - указатель на новый экземпляр подкласса, то есть BombedWal1 *. Мы определяем Clone в базовом классе именно таким образом, чтобы клиентам, клонирующим прототип, не надо было знать о его конкретных подклассах. Клиентам никогда не придется приводить значение, возвращаемое Clone, к нужному типу.

1.11. Известные применения

Быть может, впервые паттерн прототип был использован в системе Sketchpad Ивана Сазерленда (Ivan Sutherland). Первым широко известным применением этого паттерна в ОО-языке была система Thing-Lab, в которой пользователи могли сформировать составной объект, а затем превратить его в прототип, поместив в библиотеку повторно используемых объектов. Адель Голдберг и Давид Робсон упоминают прототипы в качестве паттернов в работе, но Джеймс Коплиен рассматривает этот вопрос гораздо шире. Он описывает связанные с прототипом идиомы языка C++ и приводит много примеров и вариантов.

Etgdb - это оболочка отладчиков на базе ET++, где имеется интерфейс вида point-and-click (укажи и щелкни) для различных командных отладчиков. Для каждого из них есть свой подкласс Debugger Adaptor. Например, GdbAdaptor настраивает etgdb на синтаксис команд GNU gdb, а SunDbxAdaptor - на отладчик dbx компании Sun. Набор подклассов DebuggerAdaptor не «зашит» в etgdb. Вместо этого он получает имя адаптера из переменной среды, ищет в глобальной таблице прототип с указанным именем, а затем его клонирует. Добавить к etgdb новые отладчики можно, связав ядро с подклассом DebuggerAdaptor, разработанным для этого отладчика.

Библиотека приемов взаимодействия в программе Mode Composer хранит прототипы объектов, поддерживающих различные способы интерактивных отношений. Любой созданный с помощью Mode Composer способ взаимодействия можно применить в качестве прототипа, если поместить его в библиотеку. Паттерн прототип позволяет программе поддерживать неограниченное число вариантов отношений.

Пример музыкального редактора, обсуждавшийся в начале этого раздела, основан на каркасе графических редакторов Unidraw.

1.12. Родственные паттерны

В некоторых отношениях прототип и абстрактная фабрика являются конкурентами. Но их используют и совместно. Абстрактная фабрика может хранить набор прототипов, которые клонируются и возвращают изготовленные объекты.

В тех проектах, где активно применяются паттерны компоновщик и декоратор, тоже можно извлечь пользу из прототипа.

4. Порядок выполнения работы

Варианты:

1. Применить паттерн абстрактная фабрика при построении схемы из простых графических объектов.
   Продукты фабрики: прямоугольник, линия, овал, текст.

2. Применить паттерн абстрактная фабрика при построении интерфейса пользователя Продукты фабрики: список, поле ввода, кнопка, язык отображения.
3. Применить паттерн абстрактная фабрика при построении логической структуры лабиринта. Продукты фабрики: комната, дверь.

Применить паттерн проектирования “Prototype” совместно с абстрактной фабрикой, то есть внести изменения в проект “Порождающие паттерны. Абстрактная фабрика”. То есть теперь в проекте абстрактная фабрика должна параметризоваться прототипами.

6. Контрольные вопросы

1. Что такое паттерны проектирования? Их назначение и общие черты.
2. Как описываются паттерны проектирования?
3. Для чего предназначены порождающие паттерны?
4. Назначение и случаи применения паттерна абстрактная фабрика?
5. Какие преимущества и ограничения у паттерна абстарктная фабрика?
6. Каким образом можно реализовать данный паттерн?
7. Для чего предназначен паттерн “прототип”?
8. В каких случаях применяется прототип?
9. Назовите основные отличия данного паттерна от других порождающих паттернов?
10. Каковы особенности реализации прототипа?
11. Какие достоинства и прототипа?

ЧАСТЬ 2 Лабораторная работа №. Порождающие паттерны

1. Паттерн Singleton

1.1. Название и классификация паттерна

Одиночка - паттерн, порождающий объекты.

1.2. Назначение

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

1.3. Мотивация

Для некоторых классов важно, чтобы существовал только один экземпляр. Хотя в системе может быть много принтеров, но возможен лишь один спулер. Должны быть только одна файловая система и единственный оконный менеджер. В цифровом фильтре может находиться только один аналого-цифровой преобразователь (АЦП). Бухгалтерская система обслуживает только одну компанию.
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.

1.4. Применимость

Используйте паттерн одиночка, когда:

- должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам;
- единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

1.5. Структура

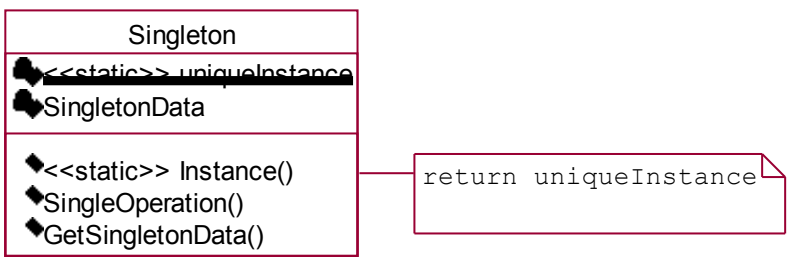

Рисунок 2.1. Структура паттерна «одиночка»

1.6. Участники

Singleton - одиночка:

- определяет операцию Instance, которая позволяет клиентам получать доступ к единственному экземпляру. Instance - это операция класса, то есть метод класса в терминологии Smalltalk и статическая функция-член в C++;
- может нести ответственность за создание собственного уникального экземпляра.

1.7. Отношения

Клиенты получают доступ к экземпляру класса Singleton операцию Instance.
только через его

1.8. Результаты

У паттерна одиночка есть определенные достоинства:

- *контролируемый доступ к единственному экземпляру.* Поскольку класс Singleton инкапсулирует свой единственный экземпляр, он полностью контролирует то, как и когда клиенты получают доступ к нему;
- *уменьшение числа имен.* Паттерн одиночка - шаг вперед по сравнению с глобальными переменными. Он позволяет избежать засорения пространства имен глобальными переменными, в которых хранятся уникальные экземпляры;
- *допускает уточнение операций и представления.* От класса Singleton можно порождать подклассы, а приложение легко сконфигурировать экземпляром расширенного класса. Можно конкретизировать приложение экземпляром того класса, который необходим во время выполнения;
- *допускает переменное число экземпляров.* Паттерн позволяет вам легко изменить свое решение и разрешить появление более одного экземпляра класса Singleton. Вы можете применять один и тот же подход для управления числом экземпляров, используемых в приложении. Изменить нужно будет лишь операцию, дающую доступ к экземпляру класса Singleton;
- *большая гибкость, чем у операций класса.* Еще один способ реализовать функциональность одиночки - использовать операции класса, то есть статические функции-члены в C++ и методы класса в Smalltalk. Но оба этих приема препятствуют изменению дизайна, если потребуется разрешить наличие нескольких экземпляров класса. Кроме того, статические функции-члены в C++ не могут быть виртуальными, так что их нельзя полиморфно заместить в подклассах.

1.9. Реализация

При использовании паттерна одиночка надо рассмотреть следующие вопросы:

- *гарантирование единственного экземпляра.* Паттерн одиночка устроен так, что тот единственный экземпляр, который имеется у класса, - самый обычный, но больше одного экземпляра создать не удастся. Чаще всего для этого прячут операцию, создающую экземпляры, за операцией класса (то есть за статической функцией-членом или методом класса), которая гарантирует создание не более одного экземпляра. Данная операция имеет доступ к переменной, где хранится уникальный экземпляр, и гарантирует инициализацию переменной этим экземпляром перед возвратом ее клиенту. При таком подходе можно не сомневаться, что одиночка будет создан и инициализирован перед первым использованием.

  В C++ операция класса определяется с помощью статической функции-члена Instance класса Singleton. В этом классе есть также статическая переменная-член _instance, которая содержит указатель на уникальный экземпляр.

  Класс Singleton объявлен следующим образом:

```
class Singleton {
public:
      static Singleton* Instance();
protected:
      Singleton();
private:
      static Singleton* _instance;
};
```

  А реализация такова:

```
Singleton* Singleton::_instance = 0;
Singleton* Singleton::Instance()
{
      if (_instance == 0)
      {
            _instance = new Singleton;
      }
      return _instance;
}
```

Клиенты осуществляют доступ к одиночке исключительно через функцию-член Instance. Переменная _instance инициализируется нулем, а статическая функция-член Instance возвращает ее значение, инициализируя ее уникальным экземпляром, если в текущий момент оно равно 0. Функция Instance использует отложенную инициализацию: возвращаемое ей значение не создается и не хранится вплоть до момента первого обращения. Обратите внимание, что конструктор защищенный. Клиент, который попытается инстанцировать класс Singleton непосредственно, получит ошибку на этапе компиляции. Это дает гарантию, что будет создан только один экземпляр.

Далее, поскольку _instance - указатель на объект класса Singleton, то функция-член Instance может присвоить этой переменной указатель на любой подкласс данного класса. Применение возможности мы увидим в разделе «Пример кода».

О реализации в C++ скажем особо. Недостаточно определить рассматриваемый паттерн как глобальный или статический объект, а затем полагаться на автоматическую инициализацию. Тому есть три причины:

- нет возможности гарантировать, что будет объявлен только один экземпляр статического объекта;
- может не быть достаточно информации для инстанцирования любого одиночки во время статической инициализации. Одиночке могут быть необходимы данные, вычисляемые позже, во время выполнения программы;
- в C++ не определяется порядок вызова конструкторов для глобальных объектов через границы единиц трансляции. Это означает, что между одиночками не может существовать никаких зависимостей. Если они есть, то ошибок не избежать.

Еще один (хотя и не слишком серьезный) недостаток глобальных/статических объектов в том, что приходится создавать всех одиночек, даже если они не используются. Применение статической функции-члена решает эту проблему.

- *порождение подклассов Singleton.* Основной вопрос не столько в том, как определить подкласс, а в том, как сделать, чтобы клиенты могли использовать его единственный экземпляр. По существу, переменная, ссылающаяся на экземпляр одиночки, должна инициализироваться вместе с экземпляром подкласса. Простейший способ добиться этого - определить одиночку, которого нужно применять в операции Instance класса Singleton. В разделе «Пример кода» показывается, как можно реализовать эту технику с помощью переменных среды.

Другой способ выбора подкласса Singleton - вынести реализацию операции Instance из родительского класса (например, MazeFactory) и поместить ее в подкласс. Это позволит программисту на C++ задать класс одиночки на этапе компоновки (скомпоновав программу с объектным файлом, содержащим другую реализацию), но от клиента одиночка будет по-прежнему скрыт.

Такой подход фиксирует выбор класса одиночки на этапе компоновки, затрудняя тем самым его подмену во время выполнения. Применение условных операторов для выбора подкласса увеличивает гибкость решения, но все равно множество возможных классов Singleton остается жестко «зашитым» в код. В общем случае ни тот, ни другой подход не обеспечивают достаточной гибкости.

Ее можно добиться за счет использования *реестра одиночек.* Вместо того чтобы задавать множество возможных классов Singleton в операции Instance, одиночки могут регистрировать себя по имени в некотором всем известном реестре.

Реестр сопоставляет одиночкам строковые имена. Когда операции Instance нужен некоторый одиночка, она запрашивает его у реестра по имени. Начинается поиск указанного одиночки, и, если он существует, реестр возвращает его. Такой подход освобождает Instance от необходимости «знать» все возможные классы или экземпляры Singleton. Нужен лишь единый для всех классов Singleton интерфейс, включающий операции с реестром:

```
class Singleton {
public:
      static void Register(const char* name, Singleton*);
      static Singleton* Instance(); protected:
      static Singleton* Lookup(const char* name);
private:
static Singleton* _instance;
static List<NameSingletonPair>* _registry;
};
```

Операция Register регистрирует экземпляр класса Singleton под указанным именем. Чтобы не усложнять реестр, мы будем хранить в нем список объектов NamesingletonPair. Каждый такой объект отображает имя на одиночку. Операция Lookup ищет одиночку по имени. Предположим, что имя нужного одиночки передается в переменной среды:

```
Singleton*  Singleton::Instance ()
{
      if   (_instance ==  0)
      {
      const  char*  singletonName  = getenv("SINGLETON");
      // пользователь или среда предоставляют это имя на стадии
      //  запуска программы
      _instance = Lookup(singletonName);
      // Lookup возвращает 0,   если такой одиночка не найден
```

```
        return _instance;
    }
```

В какой момент классы Singleton регистрируют себя? Одна из возможностей - конструктор. Например, подкласс MySingleton мог бы работать так:

```
MySingleton::MySingleton()
{
    ...
        Singleton::Register("MySingleton", this);
}
```

Разумеется, конструктор не будет вызван, пока кто-то не инстанцирует класс, но ведь это та самая проблема, которую паттерн одиночка и пытается разрешить! В C++ ее можно попытаться обойти, определив статический экземпляр класса MySingleton. Например, можно вставить строку
static MySingleton theSingleton;
в файл, где находится реализация MySingleton.

Теперь класс Singleton не отвечает за создание одиночки. Его основной обязанностью становится обеспечение доступа к объекту-одиночке из любой части системы. Подход, сводящийся к применению статического объекта, по-прежнему имеет потенциальный недостаток: необходимо создавать экземпляры всех возможных подклассов Singleton, иначе они не будут зарегистрированы.

1.10. Пример кода

Предположим, нам надо определить класс MazeFactory для создания лабиринтов, описанный на стр. 99. MazeFactory определяет интерфейс для построения различных частей лабиринта. В подклассах эти операции могут переопределяться, чтобы возвращать экземпляры специализированных классов продуктов, например объекты BombedWall, а не просто Wall.

Существенно здесь то, что приложению Maze нужен лишь один экземпляр фабрики лабиринтов и он должен быть доступен в коде, строящем любую часть лабиринта. Тут-то паттерн одиночка и приходит на помощь. Сделав фабрику MazeFactory одиночкой, мы сможем обеспечить глобальную доступность объекта, представляющего лабиринт, не прибегая к глобальным переменным.

Для простоты предположим, что мы никогда не порождаем подклассов от MazeFactory. (Чуть ниже будет рассмотрен альтернативный подход.) В C++ для того, чтобы превратить фабрику в одиночку, мы добавляем в класс MazeFactory статическую операцию Instance и статический член _instance, в котором будет храниться единственный экземпляр. Нужно также сделать конструктор защищенным, чтобы предотвратить случайное инстанцирование, в результате которого будет создан лишний экземпляр:

```
class MazeFactory
{
public:
      static MazeFactory* Instance();
      // здесь находится существующий интерфейс
      protected:
      MazeFactory();
private:
static MazeFactory* _instance;
};
```

Реализация класса такова:

```
MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::_instance ()
{
      if (_instance ==0)
      {
            _instance = new MazeFactory;
      }
      return _instance;
}
```

Теперь посмотрим, что случится, когда у MazeFactory есть подклассы и определяется, какой из них использовать. Вид лабиринта мы будем выбирать с помощью переменной среды, поэтому добавим код, который инстанцирует нужный подкласс MazeFactory в зависимости от значения данной переменной. Лучше всего поместить код в операцию Instance, поскольку она уже и так инстанцирует MazeFactory:

```
MazeFactory* MazeFactory::Instance ()
{
      if (_instance ==0)
      {
            const char* mazeStyle = getenv("MAZESTYLE"};
            if (strcmp(mazeStyle, "bombed") == 0)
            {
```

```
                _instance = new BombedMazeFactory;
            }
            else
            if (strcmp(mazeStyle, "enchanted") == 0)
            {
                _instance = new EnchantedMazeFactory;
                // ... другие возможные подклассы
            }
            else
            {
                 // по умолчанию
                _instance = new MazeFactory;
            }
        }
         return „instance;
}
```

Отметим, что операцию Instance нужно модифицировать при определении каждого нового подкласса MazeFactory. В данном приложении это, может быть, и не проблема, но для абстрактных фабрик, определенных в каркасе, такой подход трудно назвать приемлемым.

Одно из решений - воспользоваться принципом реестра, описанным в разделе «Реализация», Может помочь и динамическое связывание, тогда приложению не нужно будет загружать все неиспользуемые подклассы.

1.11. Известные применения

Примером паттерна одиночка в Smalltalk-80 является множество изменений кода, представленное классом ChangeSet, Более тонкий пример - это отношение между классами и их *метаклассами.* Метаклассом называется класс класса, каждый метакласс существует в единственном экземпляре. У метакласса нет имени (разве что косвенное, определяемое экземпляром), но он контролирует свой уникальный экземпляр, и создать второй обычно не разрешается.

В библиотеке Interviews для создания пользовательских интерфейсов - паттерн одиночка применяется для доступа к единственным экземплярам классов Session (сессия) и WidgetKit (набор виджетов). Классом Session определяется главный цикл распределения событий в приложении. Он хранит пользовательские настройки стиля и управляет подключением к одному или нескольким физическим дисплеям. WidgetKit - это абстрактная фабрика для определения внешнего облика интерфейсных виджетов. Операция Widget-Kit::instance() определяет конкретный инстанцируемый подкласс WidgetKit. На основе переменной среды, которую устанавливает Session. Аналогичная операция в классе Session «выясняет», поддерживаются ли монохромные или цветные дисплеи, и соответственно конфигурирует одиночку Session.

1.12. Родственные паттерны

С помощью паттерна одиночка могут быть реализованы многие паттерны. См описание абстрактной фабрики, строителя и прототипа.

2. Паттерн Builder

2.1. Название и классификация паттерна

Строитель - паттерн, порождающий объекты.

2.2. Назначение

Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться различные представления.

2.3. Мотивация

Программа, в которую заложена возможность распознавания и чтения документа в формате RTF, должна также «уметь» преобразовывать его во многие другие форматы, например в простой ASCII-текст или в представление, которое можно отобразить в виджете для ввода текста. Однако число вероятных преобразований заранее неизвестно. Поэтому должна быть обеспечена возможность без труда добавлять новый конвертор.

Таким образом, нужно сконфигурировать класс RTFReader с помощью объекта TextConverter, который мог бы преобразовывать 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.

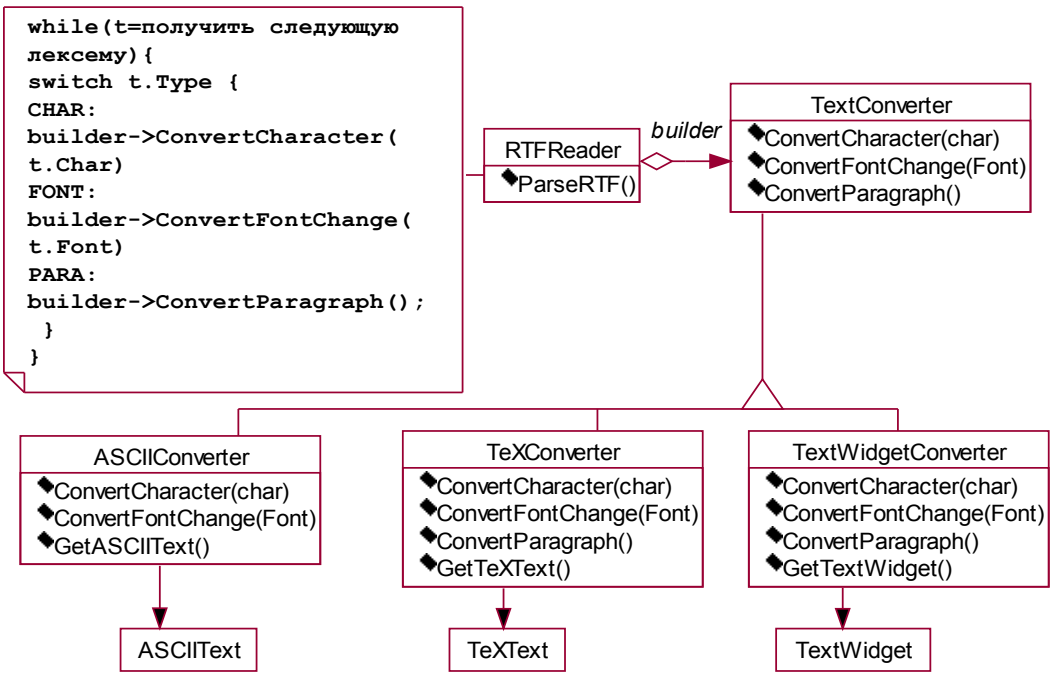

Рисунок 1.1. Пример применения паттерна «строитель»

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 конвертора принимает механизм создания и сборки сложного объекта и скрывает его за абстрактным интерфейсом. Конвертор отделен от загрузчика, который отвечает за синтаксический разбор RTF-документа.

В паттерне строитель абстрагированы все эти отношения. В нем любой класс конвертора 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.

2.4. Применимость

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.

2.5. Структура

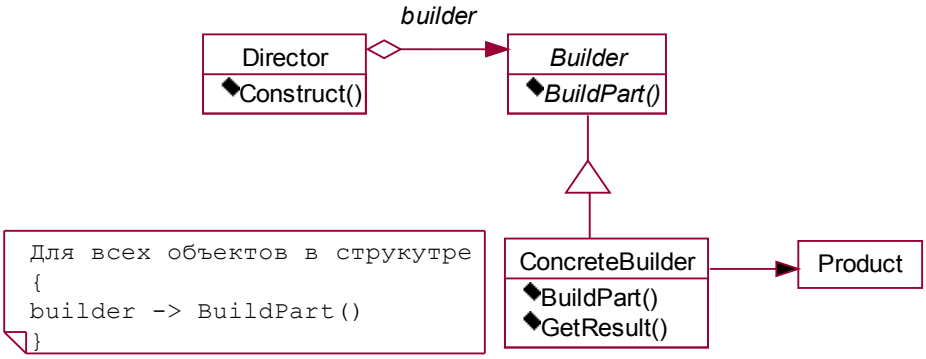

Рисунок 1.2. Структура паттерна «строитель»

2.6. Участники

- **Builder** (TextConverter) - строитель:

задает абстрактный интерфейс для создания частей объекта Product;

- **ConcreteBuilder** (ASCIIConverter, TeXConverter, TextWidgetConverter) -конкретный строитель:

конструирует и собирает вместе части продукта посредством реализации интерфейса Builder;
определяет создаваемое представление и следит за ним;
предоставляет интерфейс для доступа к продукту (например, GetASCIIText, GetTextWidget);

- **Director** (RTFReader) - распорядитель:

конструирует объект, пользуясь интерфейсом Builder;

- **Product** (ASCIIText, TeXText, TextWidget) - продукт:

представляет сложный конструируемый объект. ConcreteBuilder строит внутреннее представление продукта и определяет процесс его сборки;
включает классы, которые определяют составные части, в том числе интерфейсы для сборки конечного результата из частей.

2.7. Отношения

- **клиент** создает объект-распорядитель Director и конфигурирует его нужным объектом-строителем Builder;
- **распорядитель** уведомляет **строителя** о том, что нужно построить очередную часть **продукта**;
- **строитель** обрабатывает запросы распорядителя и добавляет новые части **к продукту**;
- **клиент** забирает продукт у **строителя**.

Следующая диаграмма взаимодействий иллюстрирует взаимоотношения **строителя** и **распорядителя** с **клиентом**.

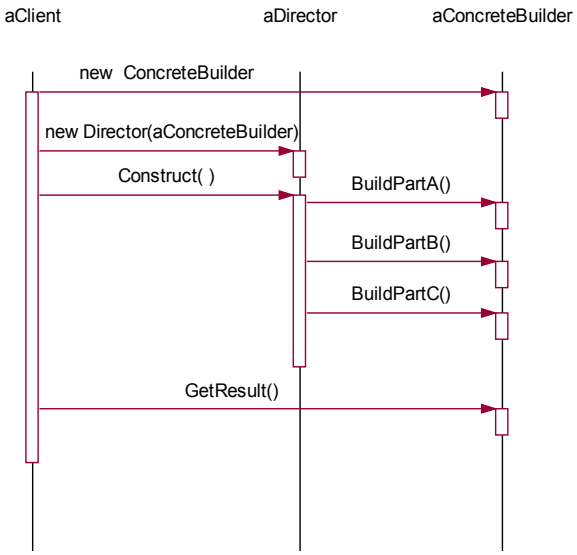

Рисунок 1.3. Отношения между участниками паттерна «строитель»

2.8. Результаты

Плюсы и минусы паттерна строитель и его применения:

- *позволяет изменять внутреннее представление продукта.* Объект Builder предоставляет распорядителю абстрактный интерфейс для конструирования продукта, за которым он может скрыть представление и

внутреннюю структуру продукта, а также процесс его сборки. Поскольку продукт конструируется через абстрактный интерфейс, то для изменения внутреннего представления достаточно всего лишь определить новый вид строителя;

- *изолирует код, реализующий конструирование и представление.* Паттерн строитель улучшает модульность, инкапсулируя способ конструирования и представления сложного объекта. Клиентам ничего не надо знать о классах, определяющих внутреннюю структуру продукта, они отсутствуют в интерфейсе строителя.

Каждый конкретный строитель ConcreteBuilder содержит весь код, необходимый для создания и сборки конкретного вида продукта. Код пишется только один раз, после чего разные распорядители могут использовать его повторно для построения вариантов продукта из одних и тех же частей В примере с RTF-документом мы могли бы определить загрузчик для формата, отличного от RTF, скажем, SGMLReader, и воспользоваться теми же самыми классами TextConverters для генерирования представлений SGML-документов в виде ASCII-текста, TeX-текста или текстового виджета.

- *дает более тонкий контроль над процессом конструирования.* В отличие от порождающих паттернов, которые сразу конструируют весь объект целиком, строитель делает это шаг за шагом под управлением распорядителя. И когда продукт завершен, распорядитель забирает его у строителя. Поэтому интерфейс строителя в большей степени отражает процесс конструирования продукта, чем другие порождающие паттерны. Это позволяет обеспечить тонкий контроль над процессом конструирования, а значит, и над внутренней структурой готового продукта.

2.9. Реализация

Обычно существует абстрактный класс Builder, в котором определены операции для каждого компонента, который распорядитель может «попросить» создать. По умолчанию эти операции ничего не делают. Но в классе конкретного строителя ConcreteBuilder они замещены для тех компонентов, в создании которых он принимает участие.

Вот еще некоторые достойные внимания вопросы реализации:

- *интерфейс сборки и конструирования.* Строители конструируют свои продукты шаг за шагом. Поэтому интерфейс класса Builder должен быть достаточно общим, чтобы обеспечить конструирование при любом виде конкретного строителя.

  Ключевой вопрос проектирования связан с выбором модели процесса конструирования и сборки. Обычно бывает достаточно модели, в которой результаты выполнения запросов на конструирование просто добавляются к продукту. В примере с RTF-документами строитель преобразует и добавляет очередную лексему к уже конвертированному тексту. Но иногда может потребоваться доступ к частям сконструированного к данному моменту продукта. В примере с лабиринтом, который будет описан в разделе «Пример кода», интерфейс класса Maze Builder позволяет добавлять дверь между уже существующими комнатами. Другим примером являются древовидные структуры, скажем, деревья синтаксического разбора, которые строятся снизу вверх. В этом случае строитель должен был бы вернуть узлы-потомки распорядителю, который затем передал бы их назад строителю, чтобы тот мог построить родительские узлы.
- *почему нет абстрактного класса для продуктов.* В типичном случае продукты, изготавливаемые различными строителями, имеют настолько разные представления, что изобретение для них общего родительского класса ничего не дает. В примере с RTF-документами трудно представить себе общий интерфейс у объектов ASCIIText и TextWidget, да он и не нужен. Поскольку клиент обычно конфигурирует распорядителя подходящим конкретным строителем, то, надо полагать, ему известно, какой именно подкласс класса Builder используется и как нужно обращаться с произведенными продуктами;
- *пустые методы класса Builder по умолчанию.* В C++ методы строителя намеренно не объявлены чисто виртуальными функциями-членами. Вместо этого они определены как пустые функции, что позволяет подклассу замещать только те операции, в которых он заинтересован.

2.10. Пример кода

Определим вариант функции-члена CreateMaze, которая принимает в качестве аргумента строитель, принадлежащий классу MazeBuilder.

Класс MazeBuilder определяет следующий интерфейс для построения лабиринтов:

```
class MazeBuilder {
public:
      virtual void BuildMaze() { }
      virtual void BuildRoom(int room) { }
      virtual void BuildDoor(int roomFrom, int roomTo) { }
      virtual Maze* GetMaze() { return 0; }
protected:
      MazeBuilder();
};
```

Этот интерфейс позволяет создавать три вещи: лабиринт, комнату с конкретным номером, двери между пронумерованными комнатами. Операция GetMaze возвращает лабиринт клиенту. В подклассах MazeBuilder данная операция переопределяется для возврата реально созданного лабиринта. Все операции построения

лабиринта в классе MazeBuilder по умолчанию ничего не делают. Но они не объявлены исключительно виртуальными, чтобы в производных классах можно было замещать лишь часть методов.
Имея интерфейс MazeBuilder, можно изменить функцию-член CreateMaze, чтобы она принимала строитель в качестве параметра:

```
Maze* MazeGame::CreateMaze   {MazeBuilder& builder)
{
      builder.BuildMaze();
      builder.BuiIdRoom(l);
      builder.BuiIdRoom(2);
      builder.BuildDoor(1, 2);
      return builder.GetMaze();
}
```

Обратите внимание, как строитель скрывает внутреннее представление лабиринта, то есть классы комнат дверей и стен, и как эти части собираются вместе для завершения построения лабиринта. Кто-то, может, и догадается, что для представления комнат и дверей есть особые классы, но относительно стен нет даже намека. За счет этого становится проще модифицировать способ представления лабиринта, поскольку ни одного из клиентов Maze Builder изменять не надо.

Как и другие порождающие паттерны, строитель инкапсулирует способ создания объектов; в данном случае с помощью интерфейса, определенного классом MazeBuilder. Это означает, что MazeBuilder можно повторно использовать для построения лабиринтов разных видов. В качестве примера приведем функцию CreateComplexMaze:

```
Maze* MazeGame::CreateComplexMaze   (MazeBuilder& builder)
{
      builder.BuiIdRoom(1);
       //   ...
      builder.BuildRoom(1001) ;
      return builder.GetMaze();
}
```

Обратите внимание, что MazeBuilder не создает лабиринты самостоятельно, его основная цель - просто определить интерфейс для создания лабиринтов. Пустые реализации в этом интерфейсе определены только для удобства. Реальную работу выполняют подклассы MazeBuilder.

Подкласс StandardMazeBuilder содержит реализацию построения простых лабиринтов. Чтобы следить за процессом создания, используется переменная _currentMaze:

```
class StandardMazeBuilder   :  public MazeBuilder
{
public:
      StandardMazeBuilder();
      virtual void BuildMaze();
      virtual  void BuildRoom(int);
      virtual  void BuildDoor(int,   int);
      virtual  Maze*  GetMaze();
private:
      Direction CommonWall(Room*,   Room*);
      Maze* _currentMaze;
};
```

CommonWall (общая стена) - это вспомогательная операция, которая определяет направление общей для двух комнат стены.
Конструктор StandardMazeBuilder просто инициализирует_currentMaze:

```
StandardMazeBuilder::StandardMazeBuilder   ()
{
      _currentMaze   =   0;
 }
```

BuildMaze инстанцирует объект класса Maze, который будет собираться другими операциями и, в конце концов, возвратится клиенту (с помощью GetMaze):

```
void StandardMazeBuilder::BuildMaze ()
{
      _currentMaze  =  new Maze;
}
Maze*  StandardMazeBuilder::GetMaze ()
{
      return _currentMaze;
}
```

Операция BuildRoom создает комнату и строит вокруг нее стены:

```
void StandardMazeBuilder::BuildRoom  (int n)
{
      if(!_currentMaze->RoomNo(n))
      {
```

```
            Room*  room = new Room(n);
            _currentMaze->AddRoom(room);
            room->SetSide(North,   new Wall);
            room->SetSide(South,  new Wall);
            room->SetSide(East,   new Wall);
            room->SetSide(West,  new Wall);
        }
}
```

Чтобы построить дверь между двумя комнатами, StandardMazeBuilder находит обе комнаты в лабиринте и их общую стену:

```
void StandardMazeBuilder::BuildDoor (int nl,   int n2)
{
      Room*  rl  = _currentMaze->RoomNo(nl);
      Room*  r2  = _currentMaze->RoomNo(n2);
      Door*  d = new Door(rl,   r2);
      rl->SetSide(CommonWall(rl,r2),   d);
      r2->SetSide(CommonWall(r2,rl),   d);
}
```

Теперь для создания лабиринта клиенты могут использовать CreateMaze в сочетании с StandardMazeBuilder:

```
      Maze* maze;
      MazeGame game;
      StandardMazeBuilder builder;
      game.CreateMaze(builder);
      maze = builder.GetMaze();
```

Мы могли бы поместить все операции класса StandardMazeBuilder в класс Maze и позволить каждому лабиринту строить самого себя. Но чем меньше класс Maze, тем проще он для понимания и модификации, а StandardMazeBuilder легко отделяется от Maze. Еще важнее то, что разделение этих двух классов позволяет иметь множество разновидностей класса MazeBuilder, в каждом из которых есть собственные классы для комнат, дверей и стен.

Необычным вариантом MazeBuilder является класс CountingMazeBuilder. Этот строитель вообще не создает никакого лабиринта, он лишь подсчитывает число компонентов разного вида, которые могли бы быть созданы:

```
class CountingMazeBuilder   :  public MazeBuilder
{
public:
      CountingMazeBuilder();
      virtual void BuildMaze();
      virtual  void BuildRoom(int); virtual void BuildDoor(int,   int); virtual  void
      AddWall(int,   Direction);
      void GetCounts(int&,   int&)   const;
private:
      int _doors;
      int _rooms;
};
```

Конструктор инициализирует счетчики, а замещенные операции класса MazeBuilder увеличивают их:

```
CountingMazeBuilder::CountingMazeBuilder   ()
{
      _rooms  = _doors  =  0;
}
void CountingMazeBuilder::BuildRoom (int)
{
      _rooms++;
}
void CountingMazeBuilder::BuildDoor (int, int)
{
      _doors++;
}
void CountingMazeBuilder::GetCounts (int& rooms, int& doors ) const
{
      rooms = _rooms;
      doors = _doors;
}
```

Вот как клиент мог бы использовать класс CountingMazeBuilder:

```
int rooms, doors;
MazeGame game;
CountingMazeBuilder builder;
game.CreateMaze(builder);
```

```
builder.GetCounts(rooms, doors);
cout « "В лабиринте есть "
« rooms « " комнат и "
« doors « " дверей" « endl;
```

2.11. Известные применения

Приложение для конвертирования из формата RTF взято из библиотеки ET+. В ней используется строитель для обработки текста, хранящегося в таком формате.

Паттерн строитель широко применяется в языке Smalltalk-80:

4. класс Parser в подсистеме компиляции - это распорядитель, которому в качестве аргумента передается объект ProgramNodeBuilder. Объект класса Parser извещает объект ProgramNodeBuilder после распознавания каждой синтаксической конструкции. После завершения синтаксического разбора Parser обращается к строителю за созданным деревом разбора и возвращает его клиенту;
5. Class Builder- это строитель, которым пользуются все классы для создания своих подклассов. В данном случае этот класс выступает одновременно в качестве распорядителя и продукта;
6. ByteCodeStream- это строитель, который создает откомпилированный метод в виде массива байтов. ByteCodeStream является примером нестандартного применения паттерна строитель, поскольку сложный объект представляется как массив байтов, а не как обычный объект Smalltalk. Но интерфейс к ByteCodeStream типичен для строителя, и этот класс легко можно было бы заменить другим, который представляет программу в виде составного объекта.

2.12. Родственные паттерны

Абстрактная фабрика похожа на строитель, в том смысле, что может конструировать сложные объекты. Основное различие между ними в том, что строитель делает акцент на пошаговом конструировании объекта, а абстрактная фабрика - на создании семейств объектов (простых или сложных). Строитель возвращает продукт на последнем шаге, тогда как с точки зрения абстрактной фабрики продукт возвращается немедленно. Паттерн компоновщик - это то, что часто создает строитель.

3. Паттерн Factory Method

3.1. Название и классификация паттерна

Фабричный метод - паттерн, порождающий классы.

3.2. Назначение

Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать инстанцирование подклассам**.** Известен также под именем Virtual Constructor (виртуальный конструктор).

3.3. Мотивация

Каркасы пользуются абстрактными классами для определения и поддержания отношений между объектами. Кроме того, каркас часто отвечает за создание самих объектов.

Рассмотрим каркас для приложений, способных представлять пользователю сразу несколько документов. Две основных абстракции в таком каркасе - 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** (открыть) или **New** (создать).

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: каркас должен инстанцировать классы, но «знает» он лишь об абстрактных классах, которые инстанцировать нельзя.

Решение предлагает паттерн фабричный метод. В нем инкапсулируется информация о том, какой подкласс класса Document создать, и это знание выводится за пределы каркаса.

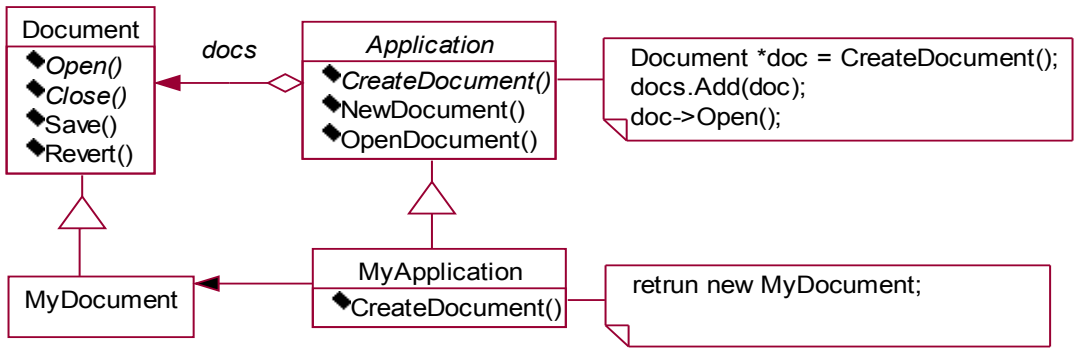

Рисунок 2.1. Пример использования паттерна «фабричный метод»

Подклассы класса Application переопределяют абстрактную операцию CreateDocument таким образом, чтобы она возвращала подходящий подкласс класса Document. Как только подкласс Application инстанцирован, он может инстанцировать специфические для приложения документы, ничего не зная об их классах. Операцию CreateDocument мы называем *фабричным методом,* поскольку она отвечает за «изготовление» объекта.

3.4. Применимость

Используйте паттерн фабричный метод, 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.

3.5. Структура

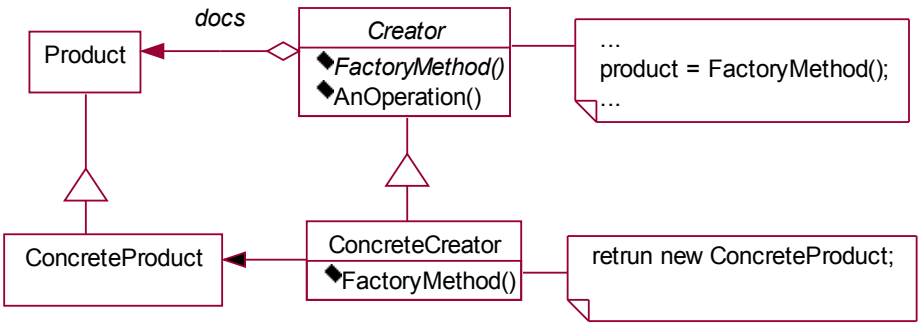

Рисунок 2.2. Структура паттерна «строитель»

3.6. Участники

- **Product** (Document) - продукт:
  определяет интерфейс объектов, создаваемых фабричным методом;
- **ConcreteProduct** (MyDocument) - конкретный продукт:
  реализует интерфейс Product;
- **Creator** (Application) - создатель:
  объявляет фабричный метод, возвращающий объект типа Product. Creator может также определять реализацию по умолчанию фабричного метода, который возвращает объект ConcreteProduct;
  может вызывать фабричный метод для создания объекта Product.
- **ConcreteCreator** (MyApplication) - конкретный создатель:
  замещает фабричный метод, возвращающий объект ConcreteProduct.

3.7. Отношения

Создатель «полагается» на свои подклассы в определении фабричного метода, который будет возвращать экземпляр подходящего конкретного продукта.

3.8. Результаты

Фабричные методы избавляют проектировщика от необходимости встраивать в код зависящие от приложения классы. Код имеет дело только с интерфейсом класса Product, поэтому он может работать с любыми определенными пользователями классами конкретных продуктов. Потенциальный недостаток фабричного метода состоит в том, что клиентам, возможно, придется создавать подкласс класса Creator для создания лишь одного объекта ConcreteProduct. Порождение подклассов оправдано, если клиенту так или иначе приходится создавать подклассы Creator, в противном случае клиенту придется иметь дело с дополнительным уровнем подклассов.

А вот еще два последствия применения паттерна фабричный метод:

- *предоставляет подклассам операции-зацепки (hooks).* Создание объектов внутри класса с помощью фабричного метода всегда оказывается более гибким решением, чем непосредственное создание. Фабричный метод создает в подклассах операции-зацепки для предоставления расширенной версии объекта.

В примере с документом класс Document мог бы определить фабричный метод CreateFileDialog, который создает диалоговое окно для выбора файла существующего документа. Подкласс этого класса мог бы определить специализированное для приложения диалоговое окно, заместив этот фабричный метод. В данном случае фабричный метод не является абстрактным, а содержит разумную реализацию по умолчанию;

- *соединяет параллельные иерархии.* В примерах, которые рассматривались до сих пор, фабричные методы вызывались только создателем. Но это совершенно необязательно: клиенты тоже могут применять фабричные методы, особенно при наличии параллельных иерархий классов. Параллельные иерархии возникают в случае, когда класс делегирует часть своих обязанностей другому классу, не являющемуся производным от него.

Рассмотрим, например, графические фигуры, которыми можно манипулировать интерактивно: растягивать, двигать или вращать с помощью мыши. Реализация таких взаимодействий с пользователем - не всегда простое дело. Часто приходится сохранять и обновлять информацию о текущем состоянии манипуляций. Но это состояние нужно только во время самой манипуляции, поэтому помещать его в объект, представляющий фигуру, не следует. К тому же фигуры ведут себя по-разному, когда пользователь манипулирует ими. Например, растягивание отрезка может сводиться к изменению положения концевой точки, а растягивание текста - к изменению междустрочных интервалов. При таких ограничениях лучше использовать отдельный объект-манипулятор Manipulator, который реализует взаимодействие и контролирует его текущее состояние. У разных фигур будут свои манипуляторы, являющиеся подклассом Manipulator. Получающаяся иерархия класса Manipulator параллельна (по крайней мере, частично) иерархии класса Figure. Класс Figure предоставляет фабричный метод CreateManipulator, который позволяет клиентам создавать соответствующий фигуре манипулятор. Подклассы Figure замещают этот метод так, чтобы он возвращал подходящий для них подкласс Manipulator. Вместо этого класс Figure может реализовать CreateManipulator так, что он будет возвращать экземпляр класса Manipulator по умолчанию, а подклассы Figure могут наследовать это умолчание. Те классы фигур, которые функционируют по описанному принципу, не нуждаются в специальном манипуляторе, поэтому иерархии параллельны только отчасти.

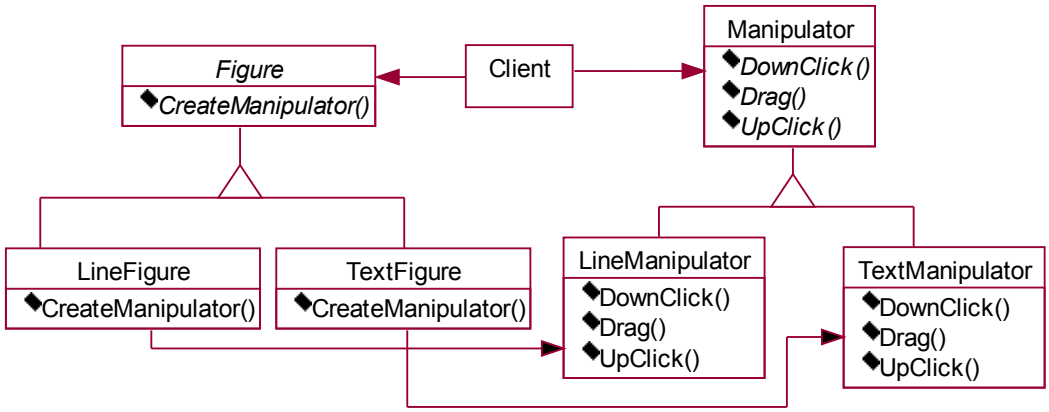

Рисунок 2.3. Пример использования паттерна «фабричный метод» при параллельных иерархиях

Фабричный метод определяет связь между обеими иерархиями классов. В нем локализуется знание о том, какие классы способны работать совместно.

3.9. Реализация

Рассмотрим следующие вопросы использования паттерна фабричный метод:

- *две основных разновидности паттерна.* Во-первых, это случай, когда класс Creator является абстрактным и не содержит реализации объявленного в нем фабричного метода. Вторая возможность: Creator - конкретный класс, в котором по умолчанию есть реализация фабричного метода. Редко, но встречается и абстрактный класс, имеющий реализацию по умолчанию; В первом случае для определения реализации необходимы подклассы, поскольку никакого разумного умолчания не существует. При этом обходится проблема, связанная с необходимостью инстанцировать заранее неизвестные классы. Во втором случае конкретный класс Creator использует фабричный метод, главным образом ради повышения гибкости. Выполняется правило: «Создавай объекты в отдельной операции, чтобы подклассы могли подменить способ их создания». Соблюдение этого правила гарантирует, что авторы подклассов смогут при необходимости изменить класс объектов, инстанцируемых их родителем;
- *параметризованные фабричные методы*. Это еще один вариант паттерна, который позволяет фабричному методу создавать разные виды продуктов. Фабричному методу передается параметр, который идентифицирует вид создаваемого объекта. Все объекты, получающиеся с помощью фабричного метода, разделяют общий интерфейс Product. В примере с документами класс Application может поддерживать разные виды документов. Вы передаете методу CreateDocument лишний параметр, который и определяет, документ какого вида нужно

создать.

В каркасе Unidraw для создания графических редакторов используется именно этот подход для реконструкции объектов, сохраненных на диске. Unidraw определяет класс Creator с фабричным методом Create, которому в качестве аргумента передается идентификатор класса, определяющий, какой класс инстанцировать. Когда Unidraw сохраняет объект на диске, он сначала записывает идентификатор класса, а затем его переменные экземпляра. При реконструкции объекта сначала считывается идентификатор класса. Прочитав идентификатор класса, каркас вызывает операцию Create, передавая ей этот идентификатор как параметр. Create ищет конструктор соответствующего класса и с его помощью производит инстанцирование. И наконец, Create вызывает операцию Read созданного объекта, которая считывает с диска остальную информацию и инициализирует переменные экземпляра.

Параметризованный фабричный метод в общем случае имеет следующий вид (здесь My Product и Your Product - подклассы Product):

```
class Creator
{
public:
      virtual Product* Create(ProductId);
};

Product* Creator::Create (ProductId id)
{
      if (id == MINE) return new MyProduct;
      if (id == YOURS) return new YourProduct;
       // выполнить для всех остальных продуктов...
      return 0;
}
```

Замещение параметризованного фабричного метода позволяет легко и избирательно расширить или заменить продукты, которые изготавливает создатель. Можно завести новые идентификаторы для новых видов продуктов или ассоциировать существующие идентификаторы с другими продуктами. Например, подкласс MyCreator мог бы переставить местами MyProduct и YourProduct для поддержки третьего подкласса TheirProduct:

```
Product*  MyCreator::Create   (ProductId id)
{
      if   (id == YOURS)   return new MyProduct;
      if   (id == MINE)   return new YourProduct;
       //  N.B.:   YOURS и MINE переставлены
      if   (id ==  THEIRS)   return new TheirProduct;

      return Creator::Create(id);   //  вызывается,   если больше ничего
                                    //не осталось
}
```

Обратите внимание, что в самом конце операция вызывает метод Create родительского класса. Так делается постольку, поскольку MyCreator::Create обрабатывает только продукты YOURS, MINE и THEIRS иначе, чем родительский класс. Поэтому MyCreator расширяет некоторые виды создаваемых продуктов, а создание остальных поручает своему родительскому классу;

- *языково-зависимые вариации и проблемы.* В разных языках возникают собственные интересные варианты и некоторые нюансы.

  Так, в программах на Smalltalk часто используется метод, который возвращает класс подлежащего инстанцированию объекта. Фабричный метод Creator может воспользоваться возвращенным значением для создания продукта, а ConcreteCreator может сохранить или даже вычислить это значение. В результате привязка к типу конкретного инстанцируемого продукта ConcreteProduct происходит еще позже.

В C++ фабричные методы всегда являются виртуальными функциями, а часто даже исключительно виртуальными. Нужно быть осторожней и не вызывать фабричные методы в конструкторе класса Creator: в этот момент фабричный метод в производном классе ConcreteCreator еще недоступен. Обойти такую сложность можно, если получать доступ к продуктам только с помощью функций доступа, создающих продукт по запросу. Вместо того чтобы создавать конкретный продукт, конструктор просто инициализирует его нулем. Функция доступа возвращает продукт, но сначала проверяет, что он существует. Если это не так, функция доступа создает продукт. Подобную технику часто называют отложенной инициализацией. В следующем примере показана типичная реализация:

```
class Creator
{
public:
      Product* GetProduct();
protected:
      virtual Product* CreateProduct();
private:
      Product* _product;
};
Product* Creator::GetProduct ()
{
```

```
      if (_product ==0)
      {
            _product - CreateProduct();
      }

return _product;
}
```

- *использование шаблонов, чтобы не порождать подклассы.* К сожалению, допустима ситуация, когда вам придется порождать подклассы только для того, чтобы создать подходящие объекты-продукты. В C++ этого можно избежать, предоставив шаблонный подкласс класса Creator, параметризованный классом Product:

```
class Creator
{
public:
      virtual Product* CreateProduct() = 0;
};
template <class TheProduct> class StandardCreator: public Creator
{
public:
      virtual  Product* CreateProduct();
};
template <class  TheProduct> Product*  StandardCreator<TheProduct>::CreateProduct   ()
{
      return new TheProduct;
}
```

С помощью данного шаблона клиент передает только класс продукта:

```
class MyProduct   :  public  Product
{
public:
      MyProduct();
      //   ...
};
StandardCreator<MyProduct> myCreator;
```

- *соглашения об именовании*. На практике рекомендуется применять такие соглашения об именах, которые дают ясно понять, что вы пользуетесь фабричными методами. Например, каркас MacApp на платформе Macintosh всегда объявляет абстрактную операцию, которая определяет фабричный метод, в виде Class* DoMakeClass (), где Class - это класс продукта.

3.10. Пример кода

Функция CreateMaze строит и возвращает лабиринт. Одна из связанных с ней проблем состоит в том, что классы лабиринта, комнат, дверей и стен жестко «зашиты» в данной функции. Мы введем фабричные методы, которые позволят выбирать эти компоненты подклассам.

Сначала определим фабричные методы в игре MazeGame для создания объектов лабиринта, комнат, дверей и стен:

```
class MazeGame
{
public:
      Maze* CreateMaze();
// фабричные методы:
virtual Maze* MakeMaze() const
{
      return new Maze; }
virtual Room* MakeRoom(int n) const
{
       return new Room(n);
}
 virtual Wall* MakeWall() const
{
       return new Wall;
}
 virtual Door* MakeDoor(Room* rl, Room* r2) const
{
      return new Door(rl, r2);
}
```

Каждый фабричный метод возвращает один из компонентов лабиринта. Класс MazeGame предоставляет реализации по умолчанию, которые возвращают простейшие варианты лабиринта, комнаты, двери и стены.
Теперь мы можем переписать функцию CreateMaze с использованием этих фабричных методов:

```
Maze* MazeGame::CreateMaze ()
{
      Maze* aMaze = MakeMaze();
      Room* rl = MakeRoom(1);
      Room* r2 = MakeRoom(2);
      Door* theDoor = MakeDoor(rl, r2);
      aMaze->AddRoom(rl);
      aMaze->AddRoom(r2);
      rl->SetSide(North, MakeWall());
      rl->SetSide(East, theDoor);
      rl->SetSide(South, MakeWall());
      rl->SetSide(West, MakeWall());
      r2->SetSide (North, MakeWall());
      r2->SetSide(East, MakeWall());
      r2->SetSide(South, MakeWall());
      r2->SetSide(West, theDoor);
      return   aMaze;
}
```

В играх могут порождаться различные подклассы MazeGame для специализации частей лабиринта. В этих подклассах допустимо переопределение некоторые или всех методов, от которых зависят разновидности продуктов. Например, в игре BombedMazeGame продукты Room и Wall могут быть переопределены так, чтобы возвращать комнату и стену с заложенной бомбой:

```
class BombedMazeGame   :  public MazeGame
{
public:
      BombedMazeGame();
      virtual Wall* MakeWall()   const
      {
               return new BombedWall;
      }
      virtual Room* MakeRoom(int n)  const
      {
         return new RoomWithABomb(n);
      }
};
```

А в игре EnchantedMazeGame допустимо определить такие варианты:

```
class  EnchantedMazeGame   :   public MazeGame
{
public:
      EnchantedMazeGame();
      virtual  Room*  MakeRoom(int n)   const
      {
            return new EnchantedRoom(n,   CastSpell());
      }
      virtual  Door* MakeDoor(Room*  rl,   Room*  r2)   const
      {
            return new DoorNeedingSpell(rl,   r2);
      }
protected:
      Spell*  CastSpell()   const;
};
```

3.11. Известные применения

Фабричные методы в изобилии встречаются в инструментальных библиотеках и каркасах. Рассмотренный выше пример с документами - это типичное применение в каркасе MacApp и библиотеке ET++ Пример с манипулятором заимствован из каркаса Unidraw.

Класс View в схеме модель/вид/контроллер из языка Smalltalk-80 имеет метод defaultController, который создает контроллер, и этот метод выглядит как фабричный. Но подклассы View специфицируют класс своего контроллера по умолчанию, определяя метод defaultControllerClass, возвращающий класс, экземпляры которого создает defaultController. Таким образом, реальным фабричным методом является defaultControllerClass, то есть метод, который должен переопределяться в подклассах.

Более необычным является пример фабричного метода parserClass, тоже взятый из Smalltalk-80, который определяется поведением Behavior (суперкласс всех объектов, представляющих классы). Он позволяет классу использовать специализированный анализатор своего исходного кода. Например, клиент может определить класс SQL Parser для анализа исходного кода класса, содержащего встроенные предложения на языке SQL. Класс Behavior реализует parserClass так, что тот возвращает стандартный для Smalltalk класс анализатора Parser. Класс

же, включающий предложения SQL, замещает этот метод (как метод класса) и возвращает класс SQLParser.

Система Orbix ORB от компании IONA Technologies использует фабричный метод для генерирования подходящих заместителей (см. паттерн заместитель) в случае, когда объект запрашивает ссылку на удаленный объект. Фабричный метод позволяет без труда заменить подразумеваемого заместителя, например таким, который применяет кэширование на стороне клиента.

3.12. Родственные паттерны

Абстрактная фабрика часто реализуется с помощью фабричных методов. Пример в разделе «Мотивация» из описания абстрактной фабрики иллюстрирует также и паттерн фабричные методы.

Паттерн фабричные методы часто вызывается внутри шаблонных методов. В примере с документами NewDocument - это шаблонный метод.

Прототипы не нуждаются в порождении подклассов от класса Creator. Однако им часто бывает необходима операция Initialize в классе Product. Creator использует Initialize для инициализации объекта. Фабричному методу такая операция не требуется.

4. Порядок выполнения работы

4.1. Применить паттерн проектирования “Singleton” совместно с абстрактной фабрикой, то есть внести изменения в проект, разработанный на лабораторной работе “Порождающие паттерны. Абстрактная фабрика”.

4.2. Применить паттерн “Строитель” для построения

- Отчета по частям
  - Части: Header – Заголовок, Block – Блок содержащий данные по отчету (результат SQL запроса), Ending – Концевик
  - ConcreteBuilder: HtmlBuilder, TxtBuilder, XlsBuilder, DocBuilder
- Представления робота в игровой программе
  - Части: Head, Body, Engine
  - ConcreteBuilder выбрать самостоятельно
- Представления лица героя в ролевой игровой программе
  - Части: Eyes - параметры (Color, Figure), Nose (Color , Figure), Mouth (Color, Figure), Ears (Color, Figure), Hair (Color, Figure)
  - ConcreteBuilder: UglyFaceBuilder, GoodFaceBuilder, SmileFaceBuilder

Применить паттерн “Фабричный метод” при создании конкретного строителя и передачи его клиенту. Самостоятельно продумать контекст в каркасе для вызова шаблонного метода.

5. Контрольные вопросы

12. Для чего предназначен паттерн “одиночка”?
13. В каких случаях необходим паттерн “одиночка”?
14. Каковы особенности реализации одиночки?
15. Для чего предназначен паттерн “Строитель ”?
16. В каких случаях применяется прототип?
17. Назовите основные отличия данного паттерна от других порождающих паттернов?
18. Для чего предназначен паттерн “Фабричный метод”?
19. В каких случаях необходим этот паттерн?
20. Каковы особенности реализации шаблонного метода?