

Урок №1

Порождающие паттерны проектирования

Содержание

1. Введение в паттерны проектирования	2
1.1. О тенденциях в развитии паттернов в программировании	2
1.2. Причины возникновения паттернов проектирования	4
1.3. Понятие паттерна проектирования.....	4
1.4. Практическое применение паттернов проектирования.....	6
1.5. Классификация паттернов	8
2. Краткое введение в UML	10
2.1. Диаграмма классов.....	10
2.2. Диаграмма объектов	15
3. Порождающие паттерны.....	16
3.1. Abstract Factory	16
3.2. Builder.....	25
3.3. Factory Method	37
3.4. Prototype	46
3.5. Singleton	56
3.6. Анализ и сравнение порождающих паттернов	63
4. Домашнее задание	64
Использованные информационные источники	65

1. Введение в паттерны проектирования

Термин «паттерн» (pattern) следует понимать как «образец». Часто его заменяют термином «шаблон» (template). По словам Кристофера Александра¹, «... любой паттерн описывает задачу, которая снова и снова возникает в нашей работе, а также принцип её решения, причем таким образом, что решение можно использовать миллион раз, ничего не изобретая заново...» [GoF95]. Такое определение паттерна существует в архитектуре (т.е. строительстве), но оно очень подходит и для определения паттерна в программировании.

В настоящем уроке мы рассмотрим основные понятия, связанные с паттернами проектирования в объектно-ориентированном программировании, также сделаем вступление в унифицированный язык визуального моделирования UML (Unified Modeling Language), с использованием которого описывается структура всех паттернов проектирования. Главной задачей урока является изложение важной категории паттернов объектно-ориентированного проектирования – порождающих паттернов (Creation Patterns), изначально описанных в [GoF95]. Все примеры практического использования паттернов представлены на языке C#, их исходные коды прилагаются к уроку.

1.1. О тенденциях в развитии паттернов в программировании

Идея паттернов проектирования первоначально возникла в архитектуре. Архитектор Кристофер Александр написал две революционные книги, содержащие описание шаблонов в строительной архитектуре и городском планировании: «A Pattern Language: Towns, Buildings, Contributions» (1977 г.), «The Timeless Way of Building» (1979 г.). В этих книгах были представлены общие идеи, которые могли использоваться даже в областях, не имеющих отношения к архитектуре, в том числе и в программировании.

¹ Кристофер Александр – архитектор, составивший в 1970-х годах XX ст. каталог паттернов проектирования в области строительной архитектуры. Позже его идея получила широкое развитие в области разработки программного обеспечения.

В 1987 году Кент Бэк (Kent Beck) и Вард Каннингем (Ward Cunningham) на основе идеи Кристофера Александра разработали шаблоны разработки программного обеспечения для графических оболочек на языке Smalltalk. В 1988 году Эрих Гамма (Erich Gamma) начал писать докторскую диссертацию при цюрихском университете об общей переносимости методики паттернов проектирования на разработку программного обеспечения. В 1989-1991 годах Джеймс Коплин (James Coplien) трудился над разработкой идиом для программирования на C++ и опубликовал в 1991 году книгу *Advanced C++ Idioms*. В этом же году Эрих Гамма заканчивает свою докторскую диссертацию и переезжает в США, где в сотрудничестве с Ричардом Хелмом (Richard Helm), Ральфом Джонсоном (Ralph Johnson) и Джоном Влиссидсом (John Vlissides) публикует книгу *Design Patterns – Elements of Reusable Object-Oriented Software* (русскоязычный перевод книги – [GoF95]). В этой книге описаны 23 паттерна проектирования. Также команда авторов этой книги известна под названием «Банда четырёх» (Gang of Four, часто сокращается – GoF). Эта книга и стала причиной роста популярности паттернов проектирования.

Таким образом, широкое использование паттернов в программировании началось из описания базовых 23-х шаблонов проектирования «Банды четырёх».

Следующим шагом стало описание Мартином Фаулером *Enterprise Patterns*, где были раскрыты типичные решения при разработке корпоративных приложений, например, работа с базами данных, транзакциями и т.п.

Джошуа Криевски показал, как можно постоянным рефакторингом², руководствуясь базовыми принципами ООП, обеспечить эволюцию кода, перемещая его от одного паттерна к другому в зависимости от требований.

После начала акцентирования внимания на модульном тестировании (Unit Testing) появилось понятие тестируемости программного кода. Все паттерны при этом были переосмыслены с позиций тестируемости. При этом, на-

² Рефакторинг – изменения внутреннего устройства программного кода без модификации его внешней функциональности.

пример, оказалось, что паттерн Singleton – это антипаттерн (см. пункт 3.5), а Abstract Factory (см. пункт 3.1) вообще заменили IoC (Inversion of Control) контейнеры. После выхода книги xUnit Test Patterns в 2008 году появилось несколько десятков паттернов тестирования.

1.2. Причины возникновения паттернов проектирования

В конце 80-х годов XX века в сфере разработки программного обеспечения, в частности, объектно-ориентированном проектировании, накопилось много различных сходных по своей сути решений. Эти решения требовали систематизации, обобщения на всевозможные ситуации, а также доступного описания, способствующего пониманию их людьми, которые до этого никогда их не использовали. Такое упорядочение знаний в объектно-ориентированном проектировании позволило бы повторно использовать готовые и уже проверенные решения, а не снова и снова «изобретать велосипед».

Решение проблемы систематизации накопленного опыта в объектно-ориентированном проектировании, а также представление его широкому кругу разработчиков и взяли на себя паттерны проектирования.

1.3. Понятие паттерна проектирования

В общем смысле паттерн представляет собой образец решения некоторой задачи так, что это решение можно использовать в различных ситуациях. В объектно-ориентированном проектировании паттерну можно дать следующие определение.

Под паттерном проектирования (design pattern) будем понимать описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте [GoF95].

Алгоритмы процедурного программирования также являются паттернами, но не проектирования, а вычислений, поскольку они решают не архитектурные, а вычислительные задачи.

Также следует подчеркнуть отличие паттернов проектирования от идиом. Идиомы – это паттерны, описывающие типичные решения на конкретном языке

ке программирования, а паттерны проектирования не зависят от выбора языка (хотя их реализации, зачастую, зависимы от языка программирования).

В общем случае каждый паттерн состоит из таких составляющих:

1. Имя является уникальным идентификатором паттерна. Ссылка на него дает возможность описать задачу проектирования и подход к её решению. Имена паттернов проектирования, описанных в [GoF95], являются общепринятыми, поэтому, например, могут использоваться в проектной документации как ссылки на типичные архитектурные решения.

2. Задача описывает ситуацию, в которой можно применять паттерн. При формулировке задачи паттерна важно описать, в каком контексте она возникает.

3. Решения задачи проектирования в виде паттерна определяет общие функции каждого элемента дизайна и отношения между ними. При этом следует подчеркнуть, что рассматривается не конкретная, а общая ситуация, так как паттерн проектирования – это шаблон, который может применяться многократно для решения типичной задачи в различных контекстах.

4. Результаты представляют следствия применения паттерна. В них описываются преимущества и недостатки выбранного решения, его последствия, различного рода компромиссы, вариации паттерна. Также в результатах следует упомянуть об особенностях использования паттерна в контексте конкретного языка программирования.

Паттерн – это общее описание хорошего способа решения задачи. Рядом с понятием «паттерн» часто фигурирует понятие «антипаттерн», или «антишаблон». Антипаттерн – это часто повторяемое плохое решение, которое не рекомендуется использовать. Цель паттерна – распознать возможность применения хорошего решения проблемы. А цель антипаттерна – обнаружить плохую ситуацию и предложить подход к её устранению.

1.4. Практическое применение паттернов проектирования

Паттерны проектирования представляют общие решения типичных задач объектно-ориентированного проектирования. При этом может сложиться впечатление, что для формирования удачного дизайна системы в первую очередь необходимо спроектировать её части, основываясь на том или ином паттерне проектирования, иными словами, необходимо подвести дизайн системы под уже известные образцы. С практической точки зрения, отталкиваться от паттернов проектирования при разработке дизайна системы, не есть самым эффективным и гибким решением. Зачастую, такой подход может привести к правильному, с академической точки зрения, но не отвечающему требованиям гибкости и масштабируемости решению.

При разработке дизайна системы следует руководствоваться такими базовыми принципами:

- Всегда формировать простой дизайн: из двух предложенных решений, как правило, лучшим является то, что проще.
- Слабая зависимость: дизайн модуля должен быть таким, что бы в случае его модификации зависимые фрагменты системы не требовали или почти не требовали изменений.

«А где же тогда паттерны проектирования?» – спросите вы.

При таком подходе к проектированию логической структуры системы можно увидеть типичные задачи, которые уже решены с помощью паттернов проектирования. Тогда целесообразно применить уже готовое шаблонное решение, оценив при этом возможности и перспективы на основании его описания. Но здесь снова не следует забывать о принципах простого дизайна и слабой зависимости, так как излишнее желание воспользоваться паттерном проектирования может способствовать формированию плохого дизайна. Надо также помнить о том, что дизайн системы постоянно эволюционирует, поэтому успешно примененный паттерн со временем может трансформироваться в совсем другой или вовсе исчезнуть.

Нельзя не отметить, что благодаря паттернам проектирования произошла унификация терминологии: ссылки на них, как на комплексные решения задач дизайна, можно добавлять в проектную документацию, а также использовать в дискуссиях.

Практический опыт разработки программного обеспечения говорит, что любой полученный результат всегда подлежит тщательной проверке. Одним из индикаторов плохого дизайна системы могут быть следующие негативные явления в программном коде (code smell):

- Дублирование кода: одинаковый (или почти одинаковый код) в разных местах проекта.
- Большие методы: подпрограммы с большим количеством линий кода.
- Большие классы: на один класс положено слишком много, зачастую, разнородных функций.
- Зависть: класс чрезмерно использует методы другого класса.
- Нарушение приватности: класс чрезмерно зависит от деталей реализации другого класса.
- Нарушение заветности: класс переопределяет метод базового класса и при этом нарушает его контракт (т.е. первоначальное предназначение).
- Ленивый класс: класс, который делает слишком мало, т.е. класс с недостаточной или нецелостной функциональностью.
- Чрезмерная сложность: принудительное использование сложных решений, где простой дизайн был бы достаточным.
- Чрезмерно длинные идентификаторы: в том числе, использование длинных имен, для обозначения зависимостей, которые должны быть неявными.

При практическом применении паттернов проектирования также не следует забывать о том, что ООП постоянно развивается. Подходы, которые раньше считались образцовыми, со временем могут стать антипаттернами, а им на смену могут прийти значительно лучшие решения (как, например, случилось с паттерном Singleton (см. пункт 3.5)).

1.5. Классификация паттернов

В объектно-ориентированном анализе и проектировании (ООАП) на сегодняшний день разработано много различных паттернов, и паттерны проектирования (в частности, порождающие паттерны), рассматриваемые в этом уроке – это только одна из подкатегорий.

Для полноты картины вкратце рассмотрим классификацию всех паттернов ООАП:

1. Архитектурные паттерны. Описывают фундаментальные способы структурирования программных систем. Эти паттерны относятся к уровню систем и подсистем, а не классов. Паттерны этой категории систематизировал и описал К. Ларман.

2. Паттерны проектирования. Описывают структуру программных систем в терминах классов. Наиболее известными в этой области являются 23 паттерна, описанные в [GoF95].

3. Паттерны анализа. Представляют общие схемы организации процесса объектно-ориентированного моделирования.

4. Паттерны тестирования. Определяют общие схемы организации процесса тестирования программных систем.

5. Паттерны реализации. Описывают шаблоны, которые используются при написании программного кода.

Более детально рассмотрим паттерны проектирования, обобщив классификации, представленные в [DPWiki], [DPOverview]:

1. Основные паттерны (Fundamental Patterns). Представляют наиболее важные фундаментальные паттерны, которые активно используются другими паттернами.

2. Порождающие паттерны (Creational Patterns). Определяют способы создания объектов в системе.

3. Структурные паттерны (Structural Patterns). Описывают способы построение сложных структур из классов и объектов.

4. Поведенческие паттерны (Behavioral Patterns). Описывают способы взаимодействия между объектами.

5. Паттерны параллельного программирования (Concurrency Patterns). Предназначены для координирования параллельных операций; решают такие проблемы как борьба за ресурсы и взаимоблокировка.

6. Паттерны MVC. Включает такие паттерны как MVC (Model – View – Controller), MVP (Model – View – Presenter) и др.

7. Паттерны корпоративных систем (Enterprise Patterns). Представляют решения для построения и интеграции больших корпоративных программных систем.

В работе [GoF95] представлено 23 паттерна проектирования. За своим назначением эти паттерны классифицируются так:

1. Порождающие паттерны (Creational Patterns). Абстрагируют процесс инстанцирования; делают систему независимой от того, как в ней создаются, компонуются и представляются объекты.

2. Структурные паттерны (Structural Patterns). Решают вопрос о создании из классов и объектов более крупных структур.

3. Поведенческие паттерны (Behavioral Patterns). Распределяют обязанности между объектами; описывают способы их взаимодействия.

В данном учебном курсе мы, главным образом, будем изучать паттерны из [GoF95], в частности, в текущем уроке детально рассмотрены порождающие паттерны.

2. Краткое введение в UML

Унифицированный язык визуального моделирования (Unified Modeling Language – UML) представляет собой средство для визуального представления элементов программной системы. Так как в процессе изучения паттернов проектирования рассматриваемые объектно-ориентированные модели удобно представлять графически в виде диаграмм классов UML, то нам важно рассмотреть их основные элементы.

2.1. Диаграмма классов

Диаграмма классов предназначена для представления статической структуры системы в терминах классов объектно-ориентированного программирования. В нашем учебном курсе все паттерны проектирования, а также примеры их использования описываются с использованием диаграмм этого типа.

Одним из основных составляющих диаграммы классов есть собственно класс. Графически он изображается в виде прямоугольника, разделенного на три части (рис. 2.1.1):

- 1) имя класса;
- 2) атрибуты (поля, члены данных) класса;
- 3) операции (методы, функции-члены) класса.

Секции 2 и 3 в изображении класса могут отсутствовать.

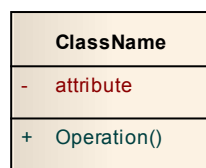


Рис. 2.1.1. Графическое изображение класса

В общем случае атрибут класса обозначается так

[visibility] name [multiplicity] [: type] [= initial value] [{ property }]

где visibility - область видимости, которая может получать следующие значения

- + – Public
- # – Protected

- - Private

name – имя атрибута (единственный обязательный параметр);

multiplicity – кратность, т.е. количество экземпляров атрибута;

type – тип, к которому принадлежит атрибут;

initial value – начальное значение;

property – другие свойства, например, readonly.

Полная форма определения операции класса имеет следующий вид:

[visibility] name ([parameter list]) [: return type] [{ property }]

где visibility – область видимости, которая может получать следующие значения

+ - Public

- Protected

- - Private

name – имя операции;

parameter list – список параметров, каждый элемент которого обозначается так:

[direct] name : type [= default value]

direct – направление передачи значения:

in – входящие

out – выходящие

inout – входящие и выходящие одновременно

name – имя параметра;

type – тип параметра;

default value – значение по умолчанию;

return type – тип возвращаемого значения;

property – другие свойства операции.

Если класс или операция класса определены как абстрактные, то их название обозначается курсивом. Имена статических классов и статических членов класса обозначаются с подчеркиванием.

Интерфейсы на диаграмме классов отображаются двумя способами: полноформатно, как класс со стереотипом Interface (рис. 2.1.2), и в сокращенной круговой нотации (рис. 2.1.3).

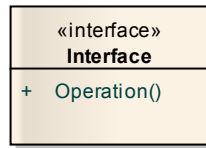


Рис. 2.1.2. Полноформатная нотация интерфейса



Рис. 2.1.3. Круговая нотация интерфейса

Наиболее важным для понимания аспектом в данном случае есть отношения между классами:

1. Отношение наследования (Generalization Relationship). Аналогично наследованию классов в объектно-ориентированном программировании. Обозначается линией со стрелкой в виде треугольника, стрелка направлена от подкласса к суперклассу (рис. 2.1.4).

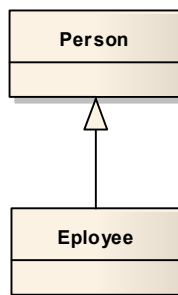


Рис. 2.1.4. Отношение наследования на диаграмме классов

2. Отношение реализации (Realization Relationship). Представляет собой частный случай отношения наследования. Используется для отображения реализации интерфейса классом. Графически изображается в виде пунктирной линии с треугольником на конце; треугольник указывает в сторону интерфейса (рис. 2.1.5).

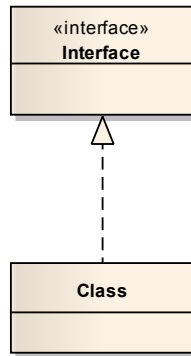


Рис. 2.1.5. Отношение реализации на диаграмме классов

3. Отношение зависимости (Dependency Relationship). Обозначает факт использования заданным классом другого класса. Отношение зависимости может, например, применяться в таких случаях:

- 1) класс содержит локальную переменную, принадлежащую к другому классу;
- 2) класс использует объект другого класса в качестве параметра операции;
- 3) метод класса возвращает объект другого класса;
- 4) класс использует статическую операцию другого класса.

Графически отношение зависимости отображается в виде пунктирной стрелки, направленной от класса который использует к классу, который используется (рис. 2.1.6).

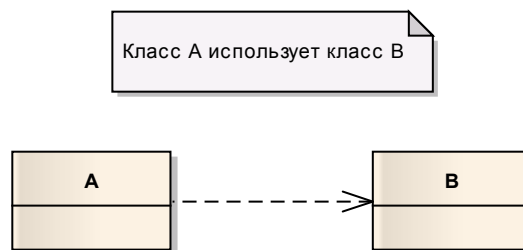


Рис. 2.1.6. Отношение зависимости на диаграмме классов

4. Отношение ассоциации (Association Relationship). Обозначает некое структурное отношение между классами. Возникает между классами, например, когда, в одном классе в качестве поля содержится объект, принадлежащий к другому классу. Графически отображается в виде сплошной линии (рис. 2.1.7).

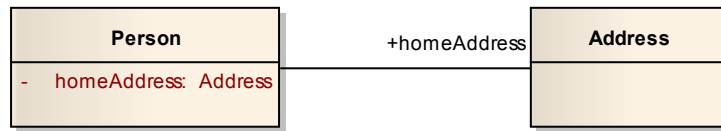


Рис. 2.1.7. Отношение ассоциации на диаграмме классов

5. Отношение агрегации (Aggregation Relationship). Представляет более строгую форму отношения ассоциации, служащую для обозначения логического включения при формировании целого из частей. Графически отображаться в виде сплошной линии с ромбом на конце со стороны класса, который обозначает целое (рис. 2.1.8).

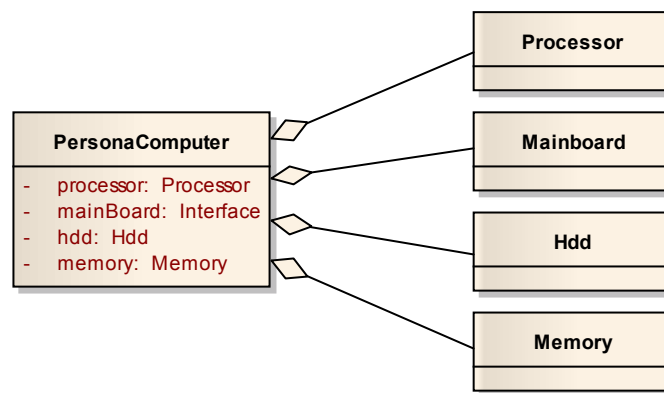


Рис. 2.1.8. Отношение агрегации на диаграмме классов

6. Отношение композиции (Composition Relationship). Представляет более строгий случай отношения агрегации. Предусматривает включение части в целое, при этом часть не может существовать отдельно от целого. Графически отображается как агрегация только с той разницей, что ромб закрашивается в черный цвет (рис. 2.1.9).

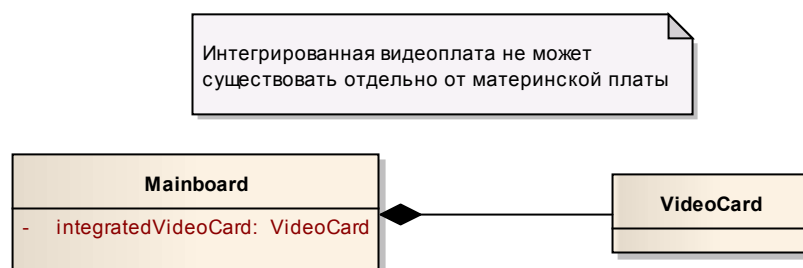


Рис. 2.1.9. Отношение композиции на диаграмме классов

2.2. Диаграмма объектов

В отличие от диаграммы классов, которая отображает статическую структурную модель системы, диаграмма объектов представляет экземпляры классов и отношения между ними в некоторый момент работы программы. Иными словами, диаграмма объектов – это снимок состояния работающей системы в заданный момент времени.

В общем случае имя объекта на диаграмме отображается с подчеркиванием и имеет вид:

Имя объекта: Имя класса

При этом одна из составляющих имени может не указываться.

Простейший пример диаграммы объектов изображен на рис. 2.2.1.

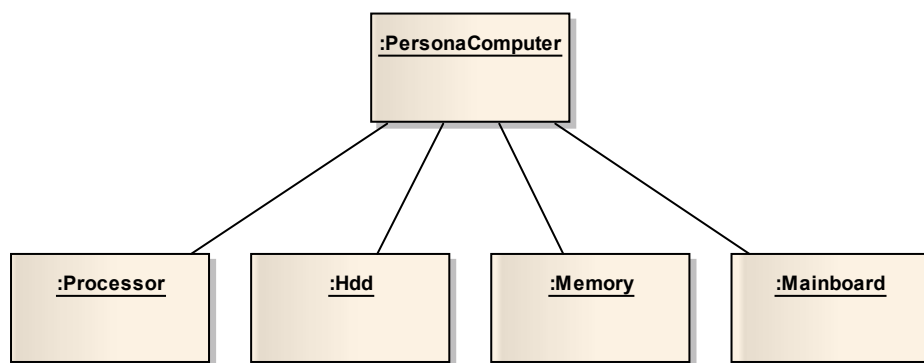


Рис. 2.2.1. Пример диаграммы объектов

Представленные выше диаграммы классов и объектов будут использоваться нами в процессе изучения всех паттернов проектирования.

3. Порождающие паттерны

Порождающие паттерны проектирования абстрагируют процесс инстанцирования. Они помогают сделать систему независимой от способа создания, композиции, и представления объектов. Паттерны этой категории позволяют ответить на вопрос: кто, когда и как создает объекты в системе. [GoF95]

3.1. Abstract Factory

3.1.1. Название паттерна

Abstract Factory / Абстрактная фабрика

Также известный под именем: Toolkit / Инструментарий

Описан в работе [GoF95].

3.1.2. Цель паттерна

Предоставить интерфейс для проектирования и реализации семейства, взаимосвязанных и взаимозависимых объектов, не указывая конкретных классов, объекты которых будут создаваться.

3.1.3. Паттерн следует использовать когда...

- Система не должна зависеть от того, как в ней создаются и компонуются объекты.
- Объекты, входящие в семейство, должны использоваться вместе.
- Система должна конфигурироваться одним из семейств объектов.
- Надо предоставить интерфейс библиотеки, не раскрывая её внутренней реализации.

3.1.4. Причины возникновения паттерна

Перед нами стоит задача разработать игру «Супер Ралли», суть которой, конечно же, заключается в гонках на автомобилях. Одно из основных функциональных требований гласит, что игрок должен иметь возможность выбрать себе автомобиль для участия в гонках. Каждый из предложенных игровых автомобилей состоит из специфического набора составляющих (двигателя, колес, кузова, коробки передач, бензобака), которые в совокупности определяют воз-

возможности автомобиля (скорость, маневренность, устойчивость к повреждениям, длительность непрерывной гонки без дозаправки и др.). Так как разных типов автомобилей в игре может быть много, более того, их количество может изменяться динамически (например, в зависимости от опыта игрока), то, с технической точки зрения, клиентский код, реализующий конфигурирование автомобиля специфичным семейством составляющих, не должен зависеть от типа выбранного автомобиля.

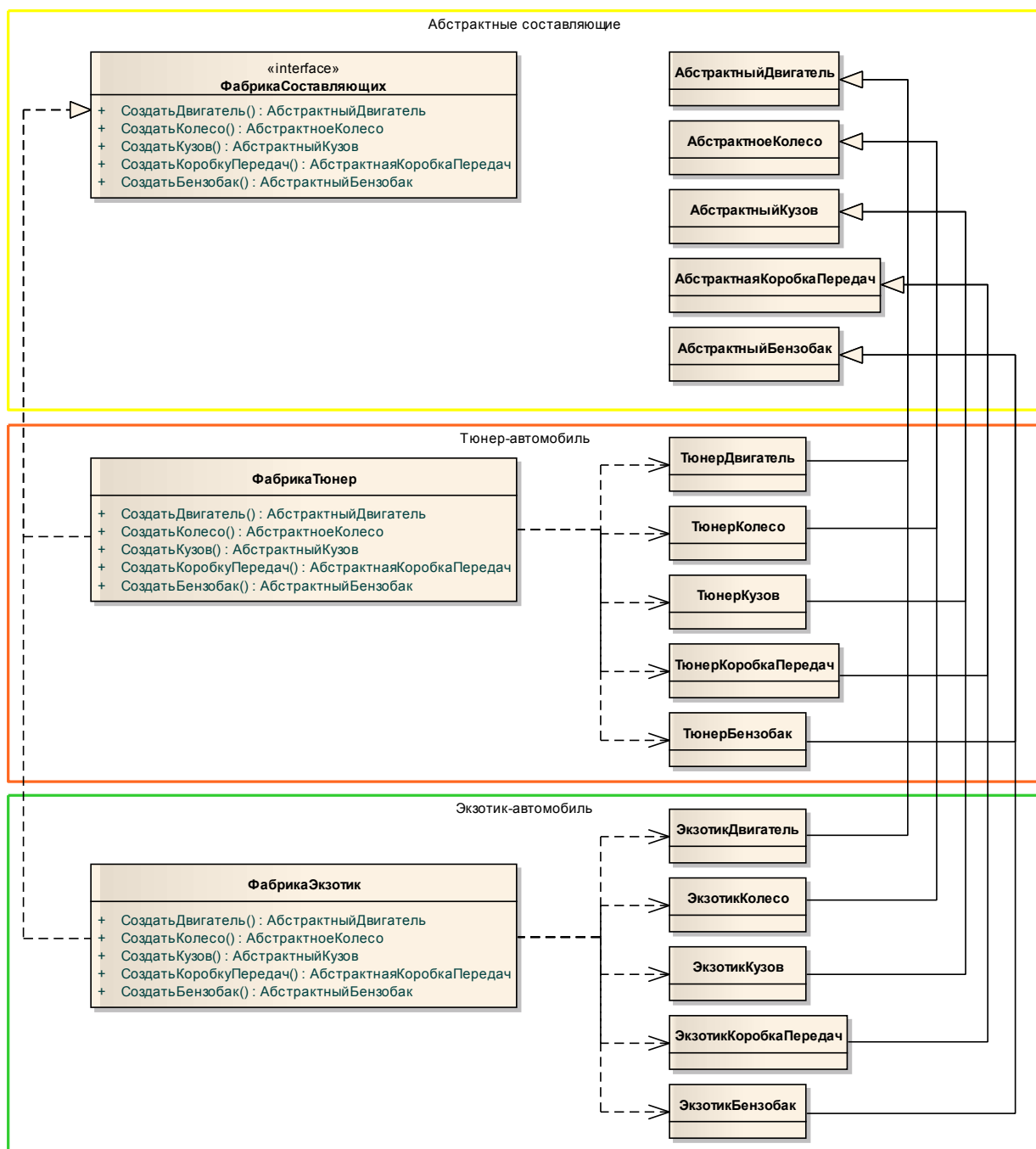


Рис. 3.1.1. Фабрики составляющих игровых автомобилей

Для реализации этого требования предлагается следующие. Рассмотрим интерфейс «ФабрикаСоставляющих», предназначение которого – представить интерфейс для конкретных классов-фабрик, которые будут создавать семейства составляющих для каждого конкретного типа автомобиля. Методы этого класса должны возвращать ссылки на абстрактные составляющие, что позволит в конкретных классах-фабриках, создавать конкретные составляющие (подклассы абстрактных составляющих). Диаграмма классов, демонстрирующая этот подход, изображена на рис. 3.1.1.

Клиентский код, который «собирает» автомобиль с деталей, конфигурируется интерфейсной ссылкой «ФабрикаСоставляющих», методы которой возвращают ссылки на абстрактные составляющие. Таким образом, мы имеем возможность передавать клиенту объект конкретной фабрики, которая создает семейство объектов конкретных составляющих.

Нельзя не отметить существенный недостаток продемонстрированного подхода: определение интерфейса «ФабрикаСоставляющих» на этапе компиляции предусматривает создание фиксированного набора продуктов, из чего следует:

- 1) каждая из конкретных фабрик должна создавать конкретные продукты соответствующие этому набору, т.е. количество и тип продуктов жестко определяется на этапе компиляции и не может меняться на этапе выполнения;
- 2) при необходимости добавления нового типа продуктов нужно изменять абстрактную фабрику и каждую из конкретных фабрик.

Во избежание таких проблем в качестве альтернативы абстрактной фабрике можно рассматривать паттерн прототип (см. пункт 3.4).

Также следует отметить, что при реализации данного подхода каждую из конкретных фабрик можно определить как Singleton (см. пункт 3.5), так как в программе, скорее всего, не будет надобности создавать более одного экземпляра конкретной фабрики.

3.1.5. Структура паттерна

Общая структура паттерна Abstract Factory представлена на рис. 3.1.2.

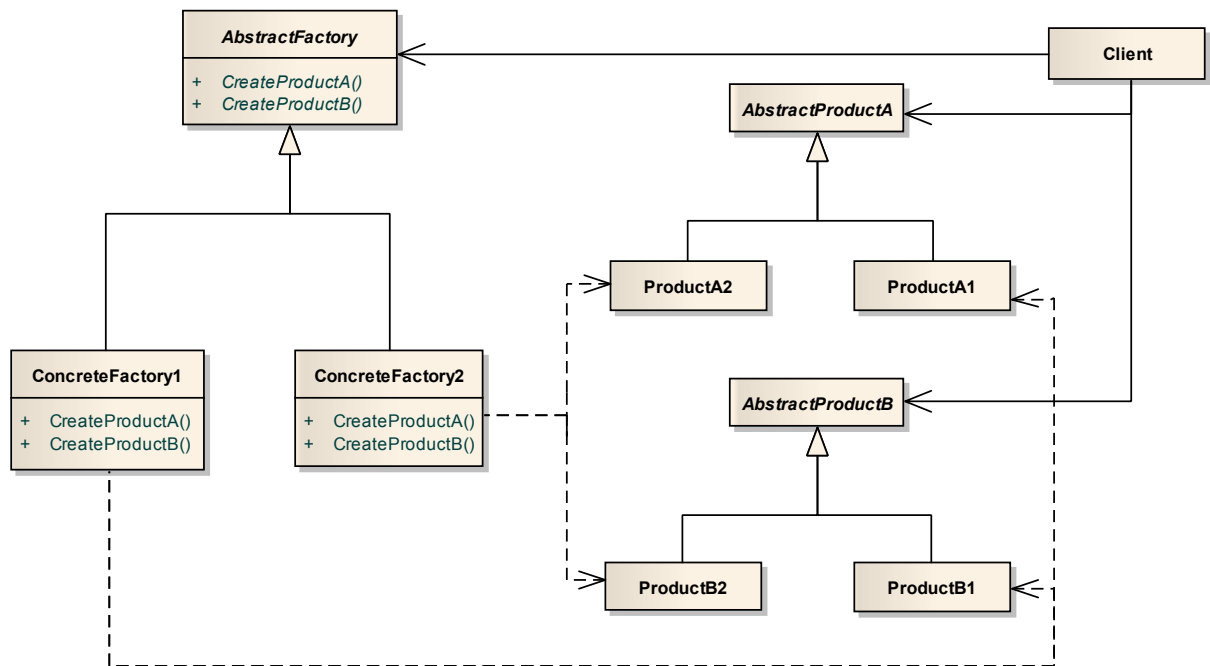


Рис. 3.1.2. Общая структура паттерна Abstract Factory

Участники паттерна:

- **AbstractFactory** – абстрактная фабрика
 - Представляет общий интерфейс для создания семейства продуктов.
- **ConcreteFactory** – конкретная фабрика
 - Реализует интерфейс **AbstractFactory** и создает семейство конкретных продуктов.
- **AbstractProduct** – абстрактный продукт
 - Представляет интерфейс абстрактного продукта, ссылку на который возвращают методы фабрик.
- **ConcreteProduct** – конкретный продукт
 - Реализует конкретный тип продукта, который создается конкретной фабрикой.

Отношения между участниками:

- Клиент знает только о существовании абстрактной фабрики и абстрактных продуктов.
- Для создания семейства конкретных продуктов клиент конфигурируется соответствующим экземпляром конкретной фабрики.

- Методы конкретной фабрики создают экземпляры конкретных продуктов, возвращая их в виде ссылок на соответствующие абстрактные продукты.

3.1.6. Результаты использования паттерна

Позволяет изолировать конкретные классы продуктов. Клиент знает о существовании только абстрактных продуктов, что ведет к упрощению его архитектуры.

Упрощает замену семейств продуктов. Для использования другого семейства продуктов достаточно конфигурировать клиентский код соответствующий конкретной фабрикой.

Дает гарантию сочетаемости продуктов. Так как каждая конкретная фабрика создает группу продуктов, то она и следит за обеспечением их сочетаемости.

Серьезным недостатком паттерна есть трудность поддержки нового вида продуктов. Для добавления нового продукта необходимо изменять всю иерархию фабрик, а также клиентский код.

3.1.7. Практический пример использования паттерна

Пусть стоит задача разработать программное обеспечение для магазина компьютерной техники. По мнению заказчика, одной из наиболее востребованных возможностей программы будет возможность быстрого создания конфигурации системного блока.

Для упрощения изложения предположим, что в состав конфигурации системного блока входят:

- 1) бокс (Box);
- 2) процессор (Processor);
- 3) системная плата (MainBoard);
- 4) жесткий диск (Hdd);
- 5) оперативная память (Memory).

Для каждой из этих составляющих определим абстрактный класс. Конкретные модели составляющих будем определять путем наследования от абстрактного базового класса. (Сразу же следует отметить, что такой подход не самый лучший с практической точки зрения.)

Класс, представляющий конфигурацию системного блока, назовем Pc (листинг 3.1.1).

Листинг 3.1.1. Класс Pc

```
/**
 * Класс персонального компьютера
 */
public class Pc
{
    public Box Box { get; set; }
    public Processor Processor { get; set; }
    public MainBoard MainBoard { get; set; }
    public Hdd Hdd { get; set; }
    public Memory Memory { get; set; }
}
```

Допустим, что наша программа должна создавать шаблоны типичных конфигураций двух типов: домашняя и офисная.

С логической точки зрения, важно, что бы все составляющие заданной конфигурации системного блока работали согласованно, поэтому ответственность за их создание надо положить на один класс-фабрику. Эта фабрика должна реализовать интерфейс IPcFactory (листинг 3.1.2). Заметим, что методы этого интерфейса возвращают ссылки на классы абстрактных продуктов.

Листинг 3.1.2. Интерфейс IPcFactory

```
/*
 * Интерфейс фабрики для создания конфигурации
 * системного блока персонального компьютера
 */
public interface IPcFactory
{
    Box CreateBox();
    Processor CreateProcessor();
    MainBoard CreateMainBoard();
    Hdd CreateHdd();
    Memory CreateMemory();
}
```

Для создания наборов компонентов типичных конфигураций определим классы конкретных фабрик HomePcFactory (листинг 3.1.3) и OfficePcFactory (листинг 3.1.4). В каждом из create-методов этих классов создается объект конкретного класса продукта, соответствующего типу конфигурации.

Листинг 3.1.3. Класс HomePcFactory

```
/*
 * Фабрика для создания "домашней" конфигурации
 * системного блока персонального компьютера
 */
public class HomePcFactory : IPcFactory {
    public Box CreateBox()
    {
        return new SilverBox();
    }
    public Processor CreateProcessor()
    {
        return new IntelProcessor();
    }
    public MainBoard CreateMainBoard()
    {
        return new MSIMainBord();
    }
    public Hdd CreateHdd()
    {
        return new SamsungHDD();
    }
    public Memory CreateMemory()
    {
        return new Ddr2Memory();
    }
}
```

Листинг 3.1.4. Класс OfficePcFactory

```
/*
 * Фабрика для создания "офисной" конфигурации
 * системного блока персонального компьютера
 */
public class OfficePcFactory : IPcFactory {
    public Box CreateBox()
    {
        return new BlackBox();
    }
    public Processor CreateProcessor()
    {
        return new AmdProcessor();
    }
    public MainBoard CreateMainBoard()
    {
        return new AsusMainBord();
    }
    public Hdd CreateHdd()
    {
        return new LGHDD ();
    }
    public Memory CreateMemory()
    {
        return new DdrMemory();
    }
}
```

```
}  
}
```

Определим класс `PcConfigurator`, отвечающий за конфигурирование объекта типа `Pc` выбранным семейством составляющих (листинг 3.1.5).

Листинг 3.1.5. Класс `PcConfigurator`

```
/*  
 * Класс, производящий конфигурирование  
 * системного блока персонального компьютера.  
 */  
public class PcConfigurator  
{  
    /*  
     * Фабрика составляющих персонального компьютера  
     */  
    public IPcFactory PcFactory { get; set; }  
    /*  
     * Метод конфигурирования системного блока  
     */  
    public void Configure(Pc pc)  
    {  
        pc.Box = PcFactory.CreateBox();  
        pc.MainBoard = PcFactory.CreateMainBoard();  
        pc.Hdd = PcFactory.CreateHdd();  
        pc.Memory = PcFactory.CreateMemory();  
        pc.Processor = PcFactory.CreateProcessor();  
    }  
}
```

Класс `PcConfigurator` принимает экземпляр конкретной фабрики и с помощью её методов создает составляющие персонального компьютера. При этом следует подчеркнуть, что `PcConfigurator` работает с интерфейсной ссылкой `IPcFactory`, т.е. он ничего не знает о конкретных фабриках конфигураций и конкретных составляющих. В этом и проявляется вся сила абстрактной фабрики – конкретную фабрику можно определять на этапе выполнения программы, и при этом клиентский код (в данном случае `PcConfigurator`) не зависит от конкретных фабрик или конкретных продуктов.

Полная диаграмма классов представленной реализации представлена на рис. 3.1.3.

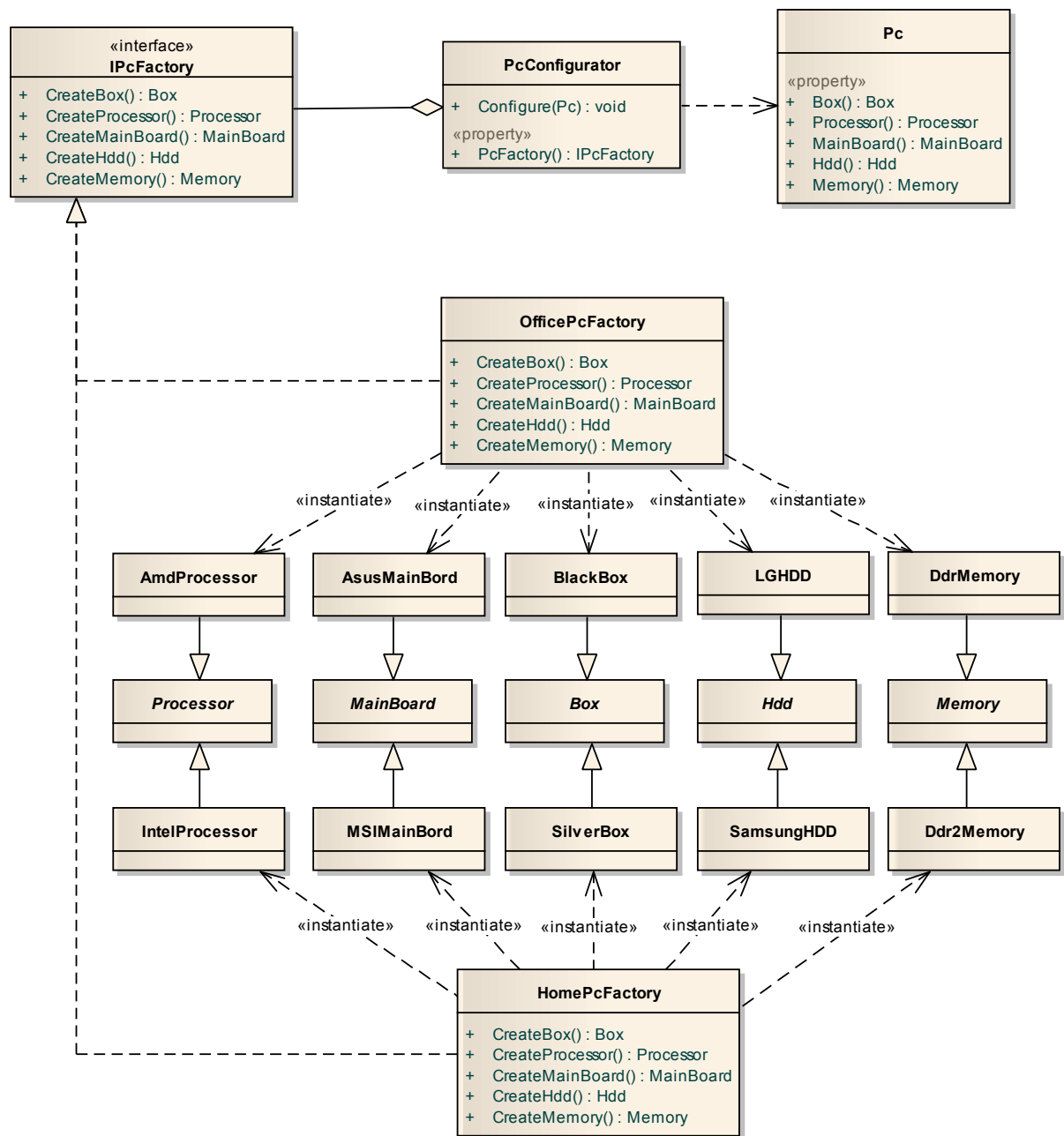


Рис. 3.1.3. Модель классов конфигуратора системного блока персонального компьютера

В представленной реализации легко увидеть типичный недостаток абстрактной фабрики: при изменении количества продуктов необходимо модифицировать всю иерархию фабрик, а также клиентский код.

Еще один подход (возможно, более гибкий) к реализации задачи типичных конфигураций персонального компьютера представлен в п. 3.4. Он базируется на использовании паттерна Prototype.

3.2. Builder

3.2.1. Название паттерна

Builder/Строитель.

Описан в работе [GoF95].

3.2.2. Цель паттерна

Отделяет процесс построения сложного объекта от его представления так, что в результате одного и того же процесса создания получаются разные представления объекта. То есть, клиентский код может порождать сложный объект, определяя для него не только тип, но и внутреннее содержимое. При этом клиент не обязан знать о деталях конструирования объекта.

3.2.3. Паттерн следует использовать когда...

- Общий алгоритм построения сложного объекта не должен зависеть от специфики каждого из его шагов.
- В результате одного и того же алгоритма конструирования надо получить различные продукты.

3.2.4. Причины возникновения паттерна

Представим себе, что мы имеем конвейер для выпуска автомобилей. Смысл конвейера заключается в пошаговом построении сложного продукта, которым в данном случае является автомобиль. Конвейер определяет общую последовательность шагов (т.е. алгоритм) конструирования. При этом специфика каждого из шагов определяется, главным образом, моделью собираемого автомобиля. Такое разделение общего алгоритма построения и специфики каждого из шагов позволят компании значительно сэкономить: на одном и том же конвейере могут выпускаться автомобили разных моделей с различными характеристиками.

С общей технологической точки зрения, автомобиль на конвейере проходит такие этапы:

1. Сборка кузова.
2. Установка двигателя.

3. Установка колес.
4. Покраска.
5. Подготовка салона.

Технические детали процессов, происходящих на каждом шаге, известны уже конкретной технологии производства модели автомобиля. Пусть завод может производить автомобили следующих моделей: автомобили класса «мини», спортивные автомобили, внедорожники. При такой постановке производственного процесса для компании не составит проблем дополнить спектр выпускаемых моделей новыми образцами без изменений общего конвейерного цикла.

Перенесемся теперь в мир объектно-ориентированного программирования.

Определим класс «Конвейер», который будет прототипом реального конвейера, т.е. будет определять общую последовательность шагов конструирования. Метод «Собрать» этого класса будет исполнять процесс конструирования посредством выполнения этих шагов без зависимости от технических деталей реализации каждого шага.

Ответственность за реализацию шагов конструирования положим на абстрактный класс, который назовем «ТехнологияМодели». В результате применения конкретных подклассов класса «ТехнологияМодели» мы получаем на выходе разные модели автомобилей, т.е. экземпляры разных классов автомобилей. В нашем случае определим такие подклассы класса «ТехнологияМодели»: «ТехнологияМиниАвто», «ТехнологияСпортивныйАвто», «ТехнологияВнедорожныйАвто». Каждая из этих технологий соответственно предусматривает выпуск таких моделей автомобилей: «МиниАвто», «СпортивныйАвто», «ВнедорожныйАвто».

Для начала производства автомобиля необходимо задать конкретную технологию для конвейера и вызвать метод «Собрать». После завершения процесса сборки готовый автомобиль можно получить у объекта технологии с помощью метода «ПолучитьРезультат()».

Диаграмма классов описанной объектно-ориентированной модели представлена на рис. 3.2.1.

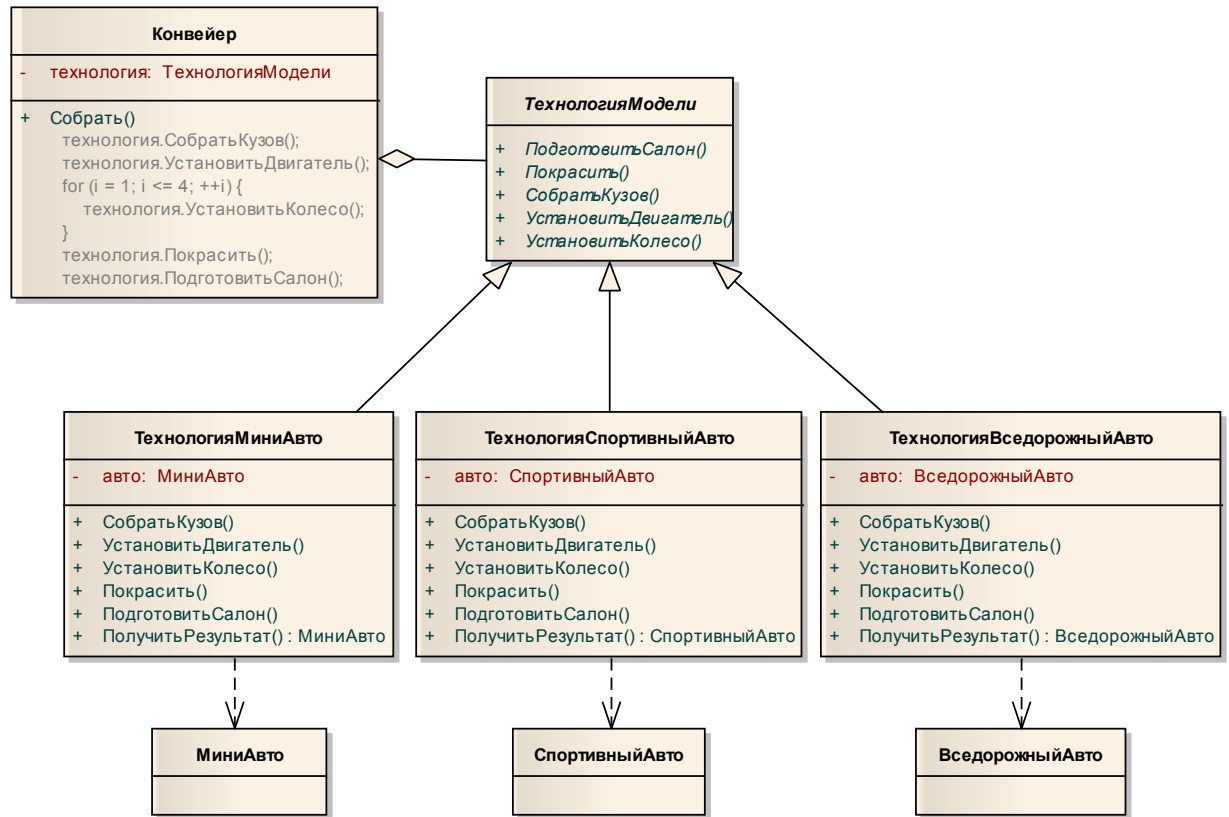


Рис. 3.2.1. Диаграмма классов модели конвейера по производству автомобилей

Построенная модель обладает рядом преимуществ:

- 1) конкретная технология конструирования строится по общему шаблону, реализуя действия, которые он определяет;
- 2) общий алгоритм процесса конструирования не зависит от деталей, специфических для конкретной технологии;
- 3) есть возможность без опасности роста сложности структуры модели реализовать под общий алгоритм большое количество конкретных технологий.

3.2.5. Структура паттерна

В общем случае структура паттерна Builder имеет вид, представленный на рис. 3.2.2.

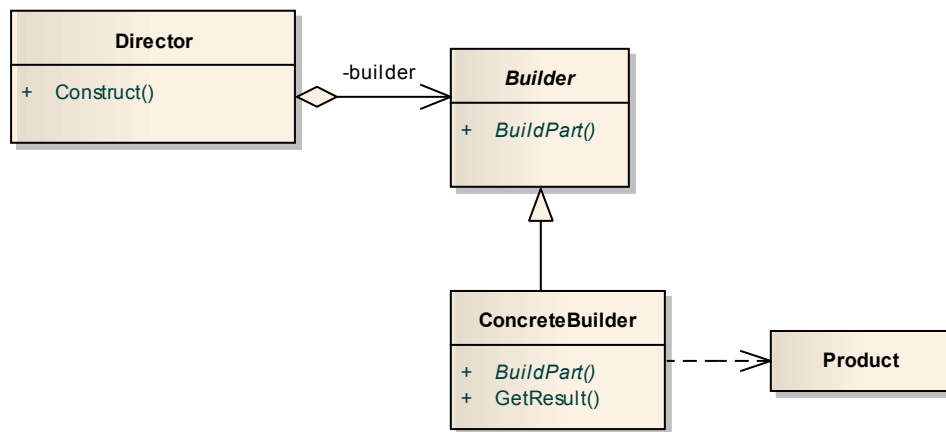


Рис. 3.2.2. Структура паттерна Builder

Участники паттерна:

- Builder (Технология Модели) – строитель
 - Обеспечивает интерфейс для пошагового конструирования сложного объекта (продукта) из частей.
- ConcreteBuilder (Технология МиниАвто и др.) – конкретный строитель
 - Реализует шаги построения сложного объекта, определенные в базовом классе Builder.
 - Создает результат построения (Product) и следит за пошаговым конструированием.
 - Определяет интерфейс для доступа к результату конструирования
- Director (Конвейер) - распорядитель
 - Определяет общий алгоритм конструирования, используя для реализации отдельных шагов возможности класса Builder.
- Product (МиниАвто и др.) – продукт
 - Сложный объект, который получается в результате конструирования.

Отношения между участниками:

- Клиент конфигурирует распорядителя (Director) экземпляром конкретного строителя.
- Распорядитель вызывает методы строителя для конструирования частей продукта.

- Конкретный строитель создает продукт и следит за его конструированием.
- Конкретный строитель представляет интерфейс для доступа к продукту.

3.2.6. Результаты использования паттерна

Есть возможность изменять внутреннюю структуру создаваемого продукта (или создать новый продукт). Так как продукт конструируется через абстрактный интерфейс класса Builder, для добавления нового продукта достаточно определить новый вид строителя (т.е. реализовать новый подкласс класса Builder).

Повышение модульности за счет разделения распорядителя и строителя. Каждый строитель имеет весь необходимый код для пошагового построения продукта. Поэтому он может использоваться разными распорядителями для построения вариантов продукта из одних и тех же частей.

Пошаговое построение продукта позволяет обеспечить более пристальный контроль над процессом конструирования (в отличие от других порождающих паттернов, которые создают продукт мгновенно).

3.2.7. Практический пример использования паттерна

Допустим, перед нами поставлена задача разработать программный модуль для манипулирования двумерными массивами действительных чисел. При этом следует учитывать, что строки массивов, с которыми будет работать наша программа, могут иметь различную длину. Главное предназначение разрабатываемого модуля – конвертирование массивов в различные форматы: текстовый файл и xml-файл. При этом в спецификации к модулю оговорено, что количество различных форматов, в которые должен быть преобразован массив, заранее не известно и, скорее всего, будет изменяться в процессе работы над программным проектом. Более того, клиентский код, который будет использовать разрабатываемый модуль, должен иметь возможность динамически (на этапе выполнения) выбирать формат, в который будет конвертирован массив.

Для начала определим класс JuggedArray (зубчатый массив) с минимальными возможностями (в случае надобности его интерфейс можно сделать более

дружественным), необходимыми для решения поставленной задачи (рис. 3.2.3). Реализация этого класса представлена в листинге 3.2.1.

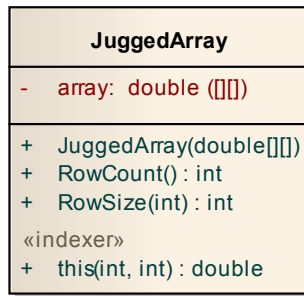


Рис. 3.2.3. Класс *JuggedArray*

Листинг 3.2.1. Класс *JuggedArray*

```
public class JuggedArray
{
    private double[][] array;
    public JuggedArray(double[][] array)
    {
        this.array = array;
    }
    public double this [int row, int col]
    {
        get
        {
            return array[row][col];
        }
        set
        {
            array[row][col] = value;
        }
    }
    public int RowCount()
    {
        return array.Length;
    }
    public int RowSize(int row)
    {
        return array[row].Length;
    }
}
```

Теперь приступим к главному вопросу: как реализовать функциональность конвертирования массива, удовлетворив все требования спецификации?

Первым и наиболее простым решением кажется добавление в класс *JuggedArray* методов *convertToText()* и *convertToXml()*, в которых будет реали-

зовано конвертирование в текстовый и xml форматы соответственно. Несмотря на кажущуюся простоту, предложенный подход имеет ряд недостатков:

- 1) при необходимости реализации нового формата конвертирования надо изменять интерфейс класса `JuggedArray`;
- 2) клиентский код, который будет использовать наш программный модуль, должен знать обо всех возможных методах конвертирования, что будет вносить в него излишнюю сложность;
- 3) теряется возможность детального контроля над процессом конструирования.

Можно использовать другой подход: переложить ответственность за конвертирование массива на специальный класс. Назовем его `JuggedArrayConverter`. В его интерфейсе определим метод `Convert()`, который будет строить представление массива в нужном формате и возвращать результат конвертирования. При этом отметим, что метод `Convert()` имеет тип результата – `object`, так как мы наперед не знаем, к какому типу данных будет принадлежать результат конвертирования. Метод `Convert()` реализует алгоритм конвертирования, который для всех форматов будет одинаковый. Но этот метод не должен уметь строить конкретный результат, так как результат полностью зависит от выбранного формата преобразования.

Ответственность за пошаговое построение результата конвертирования положим на специальный класс-строитель (`Builder`). Назовем этот класс `JuggedArrayBuilder`. Он будет базовым классом, от которого мы будем порождать подклассы для конкретных форматов конвертирования. Сам класс `JuggedArrayBuilder` определим как абстрактный, так как он предлагает только общий интерфейс строителя, не определяя его конкретной реализации.

Диаграмма классов предложенной реализации представлена на рис. 3.2.4.

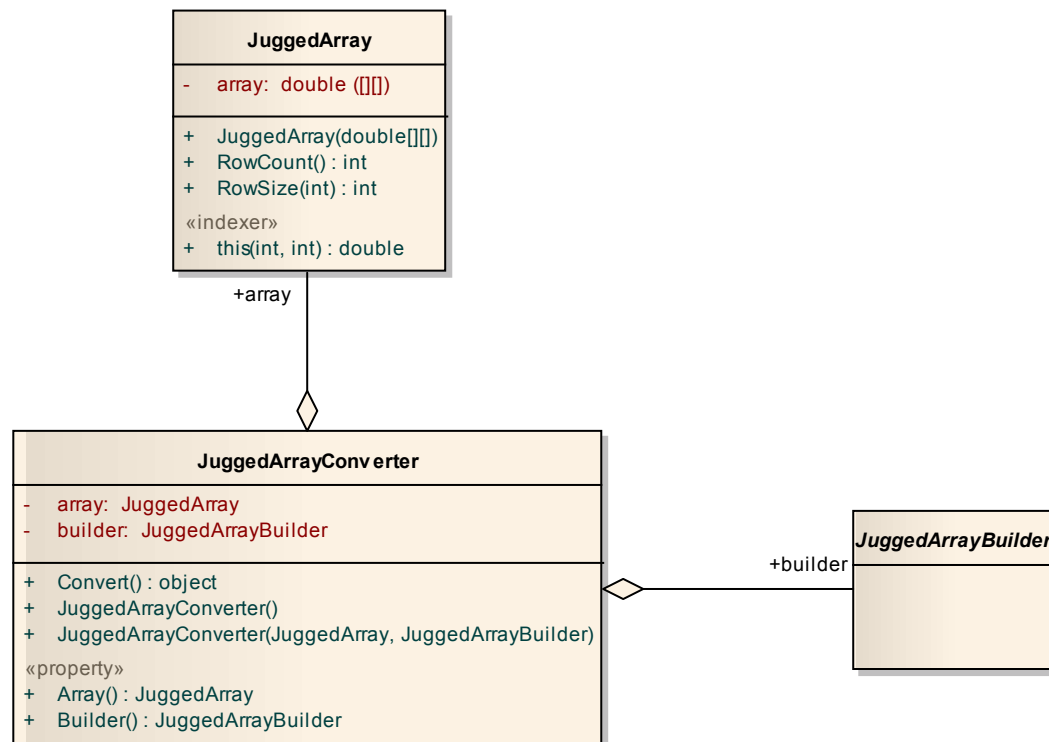


Рис. 3.2.4. Классы *JuggedArray*, *JuggedArrayConverter*, *JuggedArrayBuilder*

Теперь определим алгоритм пошагового конвертирования массива в реализации метода `Convert()` класса *JuggedArrayConverter* (листинг 3.2.2).

Как видим, метод `Convert()` определяет только общую схему алгоритма, а выполнение конкретных шагов построения обеспечивает экземпляр класса-строителя.

Все методы класса-строителя мы можем определить как абстрактные. Но надо учитывать тот факт, что реализация конкретных строителей может оставить некоторые из этих методов без реализации. Поэтому методы класса *JuggedArrayBuilder* мы определим как виртуальные, но с пустой реализацией. Исключением в этом случае будет метод `Result()`, который возвращает результат пошагового строения; ввиду необходимости его реализации в каждом конкретном строителе, мы сделаем его абстрактным. Полная реализация класса *JuggedArrayBuilder* представлена в листинге 3.2.3.

Листинг 3.2.2. Реализация алгоритма конвертирования массива

```

public object Convert()
{
    // инициализировать построение

```



```
builder.Initialize();
// начать построение
builder.Start();
for (int r = 0; r < this.array.RowCount(); ++r) {
    // начать новую строку массива
    builder.StartRow();
    for(int c = 0; c < this.array.RowSize(r); ++c){
        // добавить элемент массива
        builder.AddItem(array[r, c]);
    }
    // завершить строку массива
    builder.FinishRow();
}
// завершить построение
builder.Finish();
// получить результат построения
return builder.Result();
}
```

Листинг 3.2.3. Класс JuggedArrayBuilder

```
public abstract class JuggedArrayBuilder
{
    Public virtual void Initialize()
    {
    }
    public virtual void Start()
    {
    }
    public virtual void StartRow()
    {
    }
    public virtual void AddItem(double item)
    {
    }
    public virtual void FinishRow()
    {
    }
    public virtual void Finish()
    {
    }
    public abstract object Result();
}
```

Реализуем теперь строитель, с помощью которого мы конвертируем массив в текстовое представление (которое можно потом, например, сохранить в текстовый файл). Формат текстового представления должен подчиняться таким правилам:

- 1) отдельные строки массива должны разделяться символом «\n»;
- 2) элементы строки отделяются друг от друга пробелом.

Реализация класса TextJuggedArrayBuilder представлена в листинге 3.2.4.

Листинг 3.2.4. Класс TextJuggedArrayBuilder

```
public class TextJuggedArrayBuilder : JuggedArrayBuilder
{
    private StringBuilder text = null;
    public override void Initialize()
    {
        text = new StringBuilder();
    }
    public override void AddItem(double item)
    {
        text.Append(item);
        text.Append(" ");
    }
    public override void FinishRow()
    {
        text.AppendLine();
    }
    public override object Result()
    {
        return text;
    }
}
```

Как мы видим, методы `Start()` и `StartRow()` не реализованы в классе `TextJuggedArrayBuilder` (их реализация так и осталась пустой).

Теперь реализуем строитель xml-представления массива (листинг 3.2.5).

Листинг 3.2.5. Класс XmlJuggedArrayBuilder

```
public class XmlJuggedArrayBuilder : JuggedArrayBuilder
{
    private StringBuilder xml = null;

    public override void Initialize()
    {
        xml = new StringBuilder();
    }
    public override void Start()
    {
        xml.AppendLine("<JuggedArray>");
        xml.AppendLine("<rows>");
    }
    public override void StartRow()
    {
        xml.AppendLine("<row>");
    }
    public override void AddItem(double item)
    {
        xml.Append("<item>");
        xml.Append(item);
        xml.Append("</item>");
        xml.AppendLine();
    }
}
```

```

public override void FinishRow()
{
    xml.AppendLine("</row>");
}
public override void Finish()
{
    xml.AppendLine("</rows>");
    xml.AppendLine("</JuggedArray>");
}
public override object Result()
{
    return xml;
}
}

```

На рис. 3.2.5 изображена диаграмма классов окончательного варианта решения задачи.

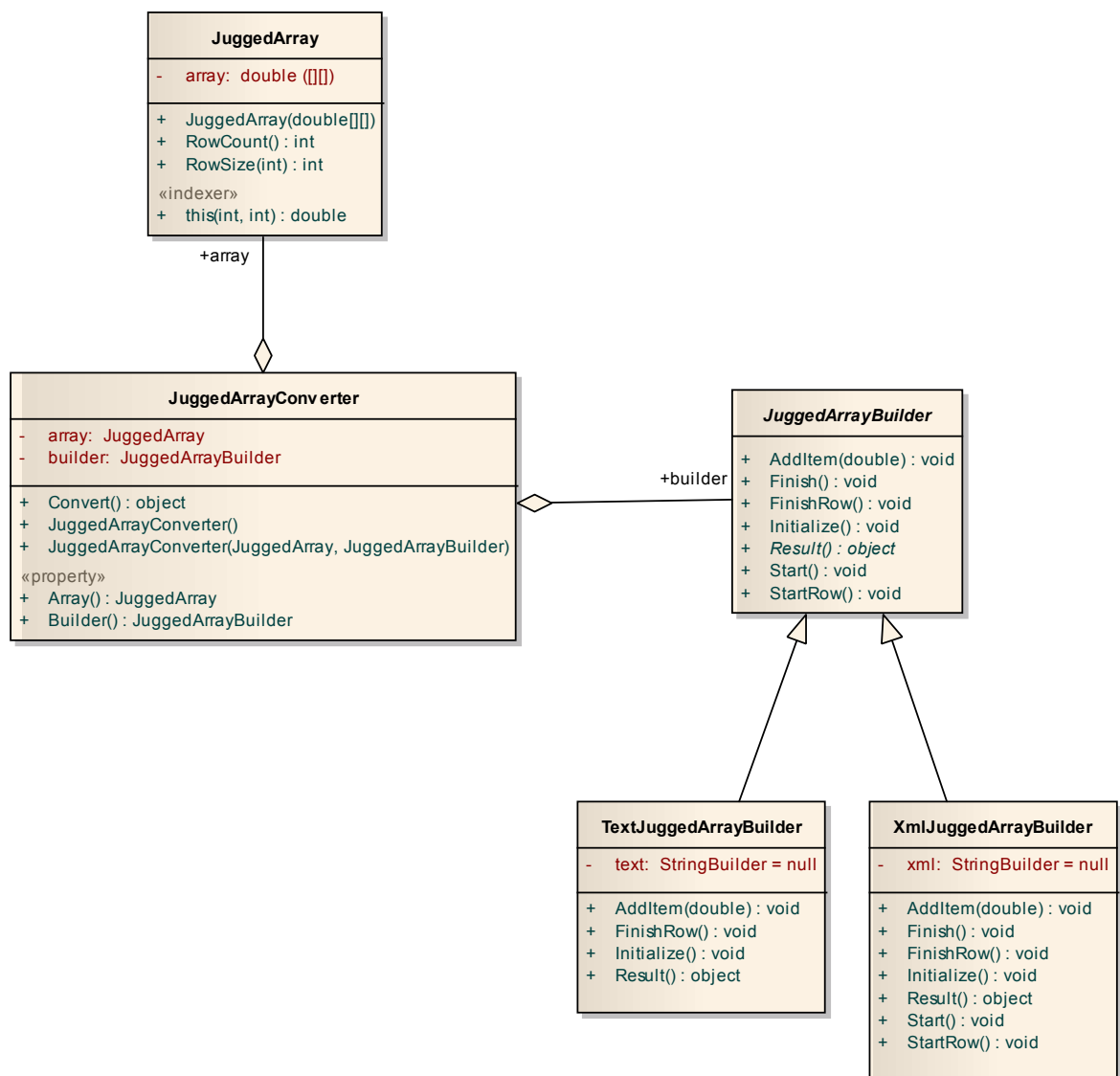


Рис. 3.2.5. Полный вариант диаграммы классов конвертера массивов

Клиентский код, демонстрирующий работу разработанных классов, представлен в листинге 3.2.6. Результат его работы изображен на рис. 3.2.6.

Листинг 3.2.6. Класс XmlJuggedArrayBuilder

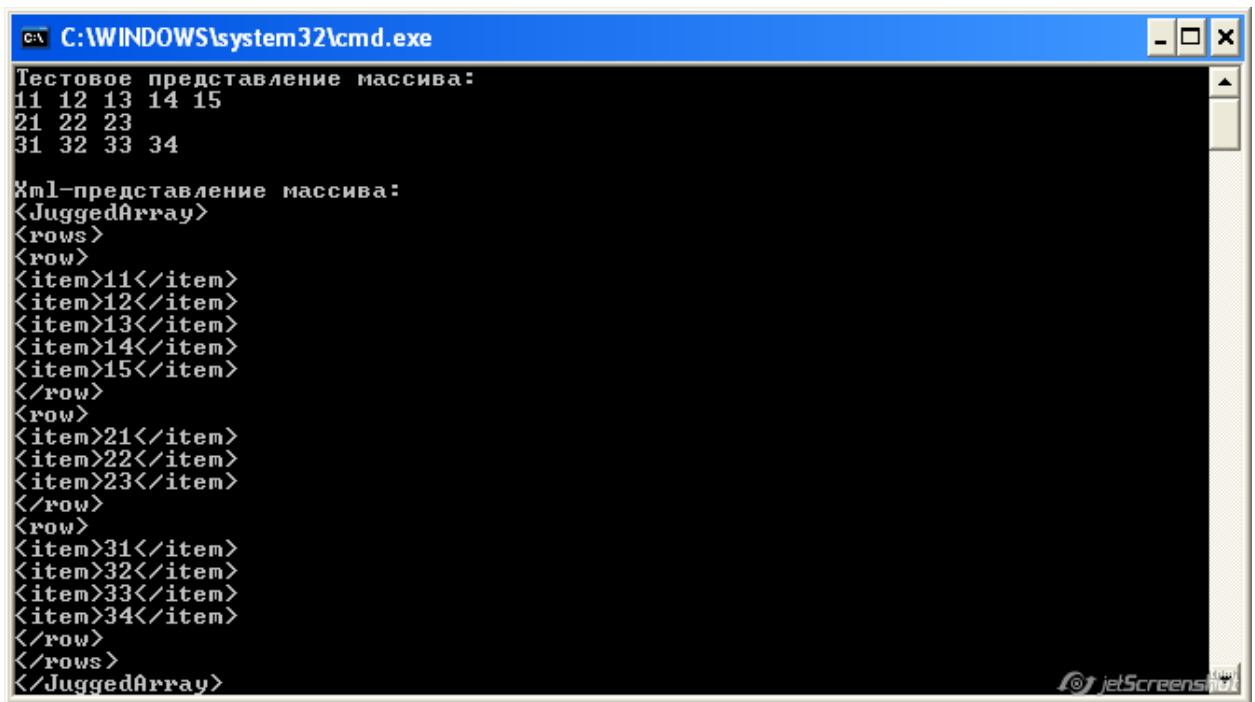
```
static void Main(string[] args)
{
    // определяем массив
    JuggedArray array = new JuggedArray(
        new double[][]
        {
            new double[] {11, 12, 13, 14, 15},
            new double[] {21, 22, 23},
            new double[] {31, 32, 33, 34}
        }
    );
    // создаем конвертер
    JuggedArrayConverter converter = new JuggedArrayConverter();
    // конфигурируем конвертер массивом
    converter.Array = array;

    Console.WriteLine("Текстовое представление массива:");
    // конфигурируем конвертер строителем текстового представления
    converter.Builder = new TextJuggedArrayBuilder();
    // проводим конвертацию
    object textArray = converter.Convert();
    // выводим результат построения на консоль
    Console.WriteLine(textArray);

    Console.WriteLine("Xml-представление массива:");
    // конфигурируем конвертер строителем xml-представления
    converter.Builder = new XmlJuggedArrayBuilder();
    // проводим конвертацию
    object xmlArray = converter.Convert();
    // выводим результат построения на консоль
    Console.WriteLine(xmlArray);
}
```

Представленная объектно-ориентированная модель дает следующие преимущества:

- 1) клиентский код не зависит от конкретного строителя;
- 2) есть возможность определить произвольное число строителей без модификации классов JuggedArray и JuggedArrayConverter.



```
C:\WINDOWS\system32\cmd.exe
Тестовое представление массива:
11 12 13 14 15
21 22 23
31 32 33 34

Xml-представление массива:
<JuggedArray>
<rows>
<row>
<item>11</item>
<item>12</item>
<item>13</item>
<item>14</item>
<item>15</item>
</row>
<row>
<item>21</item>
<item>22</item>
<item>23</item>
</row>
<row>
<item>31</item>
<item>32</item>
<item>33</item>
<item>34</item>
</row>
</rows>
</JuggedArray>
```

Рис. 3.2.6. Результат выполнения кода из листинга 3.2.6

Один из наиболее заметных недостатков нашей реализации заключается в том, что наперед невозможно определить тип результата, возвращаемого строителем.

3.3. Factory Method

3.3.1. Название паттерна

Factory Method / Фабричный метод

Также известный под именем: Virtual Constructor / Виртуальный конструктор.

Описан в работе [GoF95].

3.3.2. Цель паттерна

Определяет интерфейс для порождения объекта, но при этом оставляет за своими подклассами выбор того, какой объект надо создавать.

3.3.3. Паттерн следует использовать когда...

- Класс не должен зависеть от конкретного типа создаваемого продукта.
- Классу не известен конкретный тип продукта, который ему надо создавать.

- Конкретные типы создаваемых продуктов могут/должны определяться в подклассах.

3.3.4. Причины возникновения паттерна

Пусть разрабатывается игра типа «стрелялка». В процессе игры герой может выбрать разные типы огнестрельного оружия. Тип оружия определяет частоту выстрелов, калибр пули, тип пули (например, обычная или разрывная). Для простоты изложения идеи будем считать, что конкретный тип оружия может стрелять только одним типом пули.

Стоит следующий вопрос. Как связать конкретный класс оружия с типом пули, которыми он стреляет?

Логично было бы положить обязанность за создание экземпляра пули на объект класса оружия, из которого производится выстрел. Таким образом, каждый из конкретных типов оружия будет знать, к какому классу принадлежат его пули. Диаграмма классов, демонстрирующая такой подход, представлена на рис. 3.3.2.

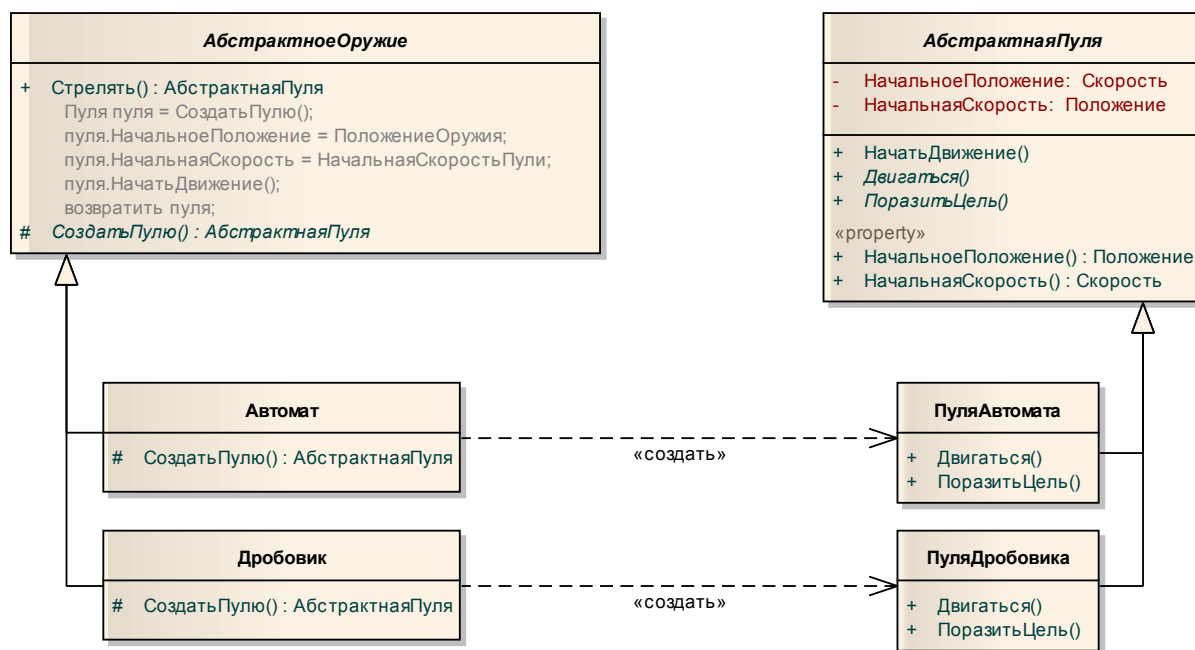


Рис. 3.3.1. Реализация взаимосвязи между типами оружия и типами пули

Изложенный подход обладает такими преимуществами:

- 1) клиентскому коду не надо заботиться о конфигурировании экземпляра оружия специфическим ему типом пули;
- 2) каждая пуля при выстреле конфигурируется параметрами (начальная скорость, начальное положение), о которых знает только оружие, из которого производится выстрел;
- 3) клиентскому коду должно быть известно только об абстрактной пуле, т.е. с каждым конкретным экземпляром пули клиент взаимодействует через интерфейс АбстрактнаяПуля (что позволяет упростить его архитектуру).

Один из основных недостатков предложенного подхода заключается в том, что тип пули для конкретного класса оружия определяется статично на этапе компиляции и не может изменяться на этапе выполнения.

Реализация продемонстрированной идеи изложена в п. 3.3.5.

3.3.5. Структура паттерна

Общая структура паттерна Factory Method, представлена на рисунке 3.3.2.

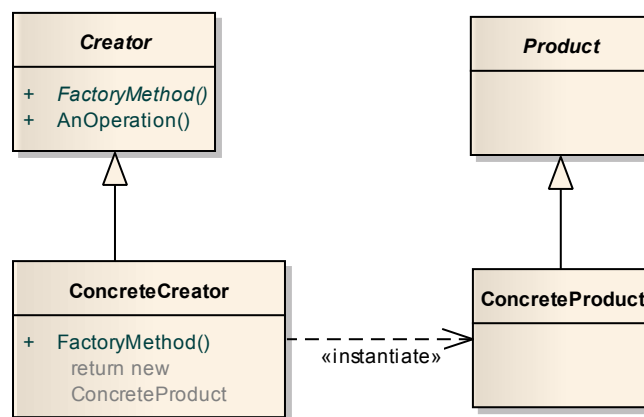


Рис. 3.3.2. Общая структура паттерна Factory Method

Участники паттерна:

- Creator (АбстрактноеОружие) – абстрактный создатель
 - Представляет абстрактный метод для создания экземпляра продукта, т.е. делегирует создание продукта своим подклассам.
- ConcreteCreator (Автомат, Дробовик) – конкретный создатель
 - Реализует метод создания экземпляра продукта.
- Product (АбстрактнаяПуля) – абстрактный продукт

- Представляет абстрактный интерфейс продукта, через который с ним работает Creator.
- ConcreteProduct (ПуляАвтомата, ПуляДробовика) – конкретный продукт
 - Реализует интерфейс абстрактного продукта.

Отношения между участниками:

- Creator представляет абстрактный метод FactoryMethod() для создания экземпляра продукта, который возвращает ссылку на Product.
- Creator возлагает ответственность за создание экземпляра конкретного продукта на свои подклассы.
- ConcreteCreator реализует метод FactoryMethod(), обеспечивая создание объекта класса ConcreteProduct.

3.3.6. Результаты использования паттерна

Класс Creator не зависит от конкретного типа создаваемых продуктов.

За создание продукта отвечают подклассы. В этом аспекте есть как преимущества, так и недостатки. Преимущества заключаются в том, что класс Creator постепенно уточняет конкретные продукты в своих подклассах. Недостаток: для реализации возможности создавать новый тип продуктов необходимо создавать новый подкласс Creator, что может привести к неоправданному росту числа его подклассов.

Позволяет объединить две параллельные иерархии классов создателей и продуктов.

Возможно определение реализации фабричного метода по умолчанию. Зачастую, фабричный метод представляется как абстрактный, что требует его обязательно определения в подклассах. Если для него определить реализацию по умолчанию, то он будет дополнительной возможностью модифицировать возможности класса при наследовании.

Конкретный класс создаваемого продукта может передаваться как тип-параметр класса Creator или его подклассов. В таком случае фабричный метод будет создавать экземпляры обобщенного типа-параметра, и для определения нового типа продуктов не обязательно прибегать к наследованию. Такой вариант реализации

может быть легко использован в C++ или C#, но будет неприемлем, например, для Java (по крайней мере, для версий 1.6 и младше).

3.3.7. Практический пример использования паттерна

Реализуем идею, представленную в п. 3.3.2.

Описание классов пуль представлено в листинге 3.3.1. Следует обратить внимание на абстрактные методы `HitTarget()`, который реализует попадание в цель, и `Movement()`, реализующий траекторию полета пули. Реализация этих методов определяется в подклассах класса `AbstractBullet`. Это значит, что наследование от класса `AbstractBullet`, в частности, обеспечивает реализацию поведения, специфического для конкретного типа пули.

Листинг 3.3.1. Реализация классов пуль

```
/*
 * Вектор в трехмерном пространстве.
 * Используется для определения направления.
 */
public struct Vector3D
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Z { get; set; }
}

/*
 * Точка в трехмерном пространстве.
 * Используется для определения положения.
 */
public struct Point3D
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Z { get; set; }
}

/*
 * Класс абстрактной пули.
 */
public abstract class AbstractBullet
{
    /*
     * Калибр пули.
     */
    public double Caliber { get; set; }
    /*
     * Текущие координаты пули.
     */
    public Point3D Location { get; set; }
    /*
```

```
        * Текущее направление пули.
        */
public Vector3D Direction { get; set; }
/*
    * Начало движения пули.
    */
public void StartMovement()
{
    // Реализация начала движения
}
/*
    * Метод поражения цели.
    * Так как разные типы пуль поражают цель по-разному,
    * то метод должен быть реализован в подклассах.
    */
public abstract void HitTarget(object target);
/*
    * Метод, реализующий движение пули.
    * Так как разные типы пуль имеют разную траекторию движения,
    * то метод должен быть реализован в подклассах.
    */
public abstract void Movement();
}
/*
    * Класс пули для автоматического оружия.
    */
public class AutomaticBullet : AbstractBullet
{
    public override void HitTarget(object target)
    {
        // реализация поражения цели target
    }

    public override void Movement()
    {
        // реализация алгоритма движения пули
    }
}
/*
    * Класс пули для дробовика.
    */
public class ShotgunBullet : AbstractBullet
{
    public override void HitTarget(object target)
    {
        // реализация поражения цели target
    }

    public override void Movement()
    {
        // реализация алгоритма движения пули
    }
}
```

Реализация классов оружия представлена в листинге 3.3.2. Метод Shoot() этого класса реализует выстрел. Технология выстрела одинакова для всех типов оружия. Для создания пули, специфической для конкретного типа оружия, вызывается фабричный метод CreateBullet(), который реализуется в каждом из подклассов. Этот метод возвращает ссылку на AbstractBullet, под которой в каждом конкретном случае будет представлен объект соответствующего класса пули.

Листинг 3.3.2. Реализация классов оружия

```
/*
 * Класс абстрактного оружия
 */
public abstract class AbstractWeapon
{
    /*
     * Фабричный метод для создания пули.
     */
    protected abstract AbstractBullet CreateBullet();

    /*
     * Текущее положение оружия
     */
    public Point3D Location { get; protected set; }
    /*
     * Направление оружия
     */
    public Vector3D Direction { get; protected set; }
    /*
     * Калибр оружия
     */
    public double Caliber { get; protected set; }

    /*
     * Метод, производящий выстрел.
     * Возвращает экземпляр созданной пули.
     */
    public AbstractBullet Shoot()
    {
        // создание объекта пули с помощью фабричного метода
        AbstractBullet bullet = CreateBullet();
        // настройка пули на текущие параметры оружия
        bullet.Caliber = this.Caliber;
        bullet.Location = this.Location;
        bullet.Direction = this.Direction;
        // начать движение пули
        bullet.StartMovement();
        // вернуть экземпляр пули
        return bullet;
    }
}
```

```
/*
 * Класс автоматического оружия.
 */
public class AutomaticWeapon : AbstractWeapon
{
    public AutomaticWeapon()
    {
        this.Caliber = 20;
    }

    /*
     * Реализация фабричного метода.
     * Создает экземпляр пули,
     * специфический для текущего типа оружия.
     */
    protected override AbstractBullet CreateBullet()
    {
        return new AutomaticBullet();
    }
}

/*
 * Класс дробовика.
 */
public class Shotgun : AbstractWeapon
{
    public Shotgun()
    {
        this.Caliber = 50;
    }

    /*
     * Реализация фабричного метода.
     * Создает экземпляр пули,
     * специфический для текущего типа оружия.
     */
    protected override AbstractBullet CreateBullet()
    {
        return new ShotgunBullet();
    }
}
```

Предложенный фрагмент реализации модели «оружие — пуля» демонстрирует только её архитектурные особенности и не претендует на функциональную полноту (в реальной игровой программе, скорее всего, технология выстрела выглядела бы намного сложнее).

Диаграмма классов описанной модели представлена на рис. 3.3.2.

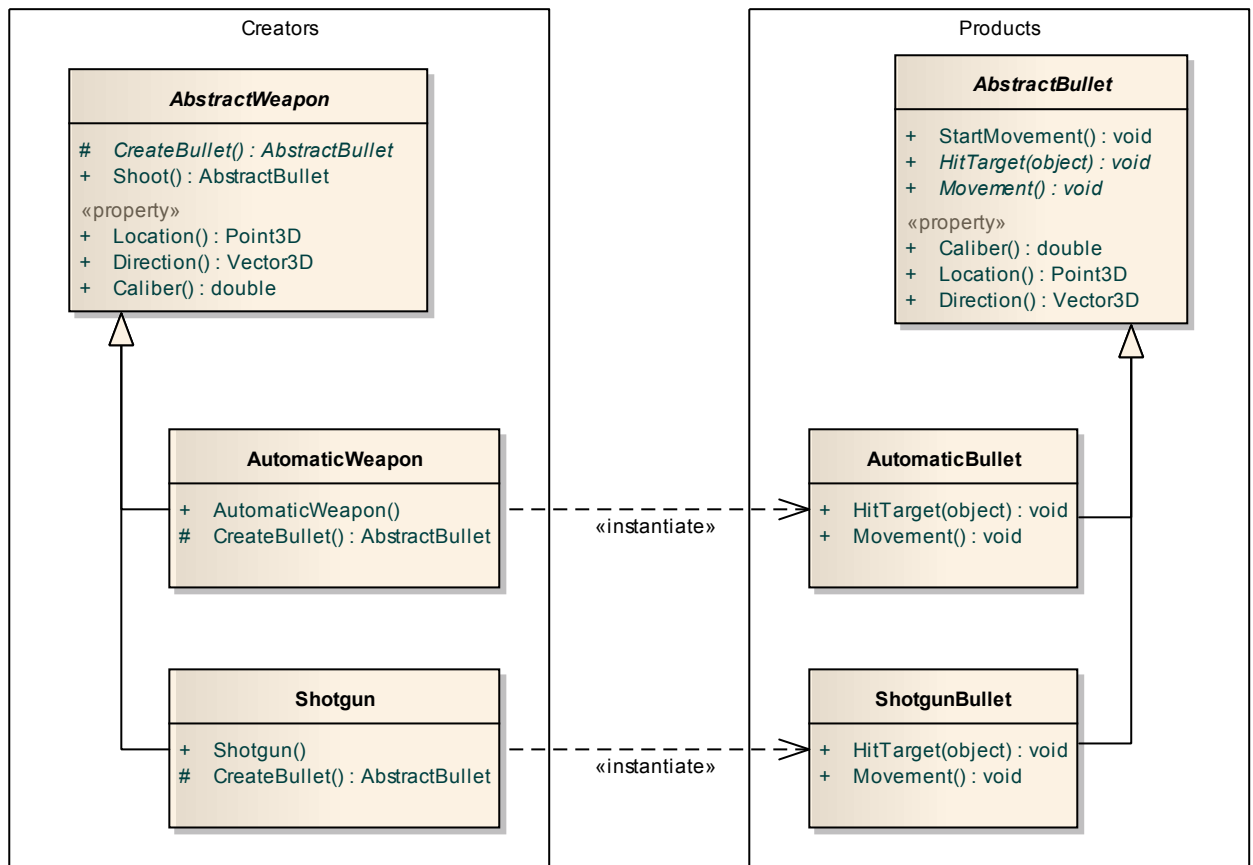


Рис.3.3.2. Диаграмма классов модели «оружие – пуля»

Предложенный подход обладает такими преимуществами:

1) риск того, что оружие останется без объекта пули в момент выстрела или что будут производиться пули не характерного оружию типа, сведен к минимуму, так как за создание пули отвечает само оружие;

2) клиентский код, использующий оружие и пулю, не зависит от того, каким типом пули производится выстрел, ему достаточно только знать о классе `AbstractBullet`;

3) ответственность за конфигурирование оружия конкретным типом производимых пуль ложится не на клиентский код, а на конкретный класс оружия, что позволяет упростить архитектуру клиента.

Недостатки предложенного решения:

1) тип пуль, которыми стреляет конкретный вид оружия, определяется статически на этапе компиляции, его нельзя изменить на этапе выполнения, поэтому заданный тип оружия может производить только один вид пуль;

2) для реализации возможности стрелять новыми пулями надо реализовать новый класс оружия, что, в конечном итоге, может привести к росту иерархии классов оружия.

3.4. Prototype

3.4.1. Название паттерна

Prototype/Прототип.

Описан в работе [GoF95].

3.4.2. Цель паттерна

Описывает виды создаваемых объектов с помощью экземпляра-прототипа и порождает новые объекты путем копирования этого прототипа.

3.4.3. Паттерн следует использовать когда...

- Клиентский код должен создавать объекты, ничего не зная об их классе, или о том, какие данные они содержат.
- Классы создаваемых объектов определяются во время выполнения (например, при динамической загрузке).
- Экземпляры класса могут пребывать в не очень большом количестве состояний, поэтому может оказаться значительно удобнее создать несколько прототипов и клонировать их вместо прямого создания экземпляра класса.

3.4.4. Причины возникновения паттерна

Допустим, что перед нами стоит задача разработать игру «Тетрис», в частности, «Стройку». Для тех, кто не когда не играл эту замечательную игру, вкратце изложим суть. С верхней части игровой области падают строительные блоки разной формы. Задача игрока заключается в том, чтобы не позволить строительным блокам заполнить всю игровую область снизу доверху. Для этого надо стараться располагать блоки так, чтобы полностью заполнить горизонтальную линию, так как полностью заполненные линии исчезают.

Набор разных форм строительных блоков фиксирован. Программа случайным образом выбирает форму последующего строительного блока, создает

блок в соответствии с выбранной формой и начинает продвигать его по игровому полю.

Стоит следующий вопрос. Как создавать экземпляры строительных блоков, минимизировав при этом зависимость клиентского кода от типа формы блока?

Можно предложить следующее решение. Пусть каждый объект строительного блока умеет создавать копию самого себя через некий универсальный интерфейс. В программе есть список блоков-прототипов, каждый из которых представляет одну из возможных форм. При необходимости создания следующего блока его прототип случайным образом выбирается из списка, после чего создается новый экземпляр строительного блока посредством клонирования выбранного прототипа. Поскольку предусмотрен универсальный интерфейс для клонирования, то клиентский код полностью не зависит от того, строительный блок какой формы ему надо клонировать.

3.4.5. Структура паттерна

Общая структура паттерна представлена на рис. 3.4.1.

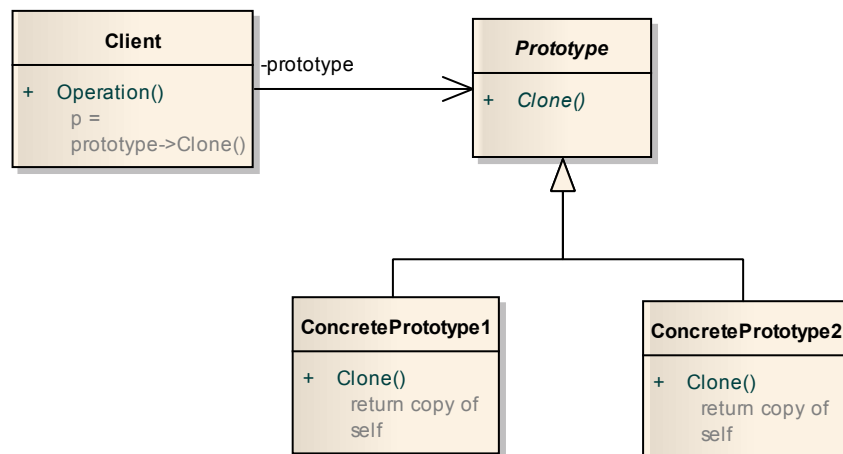


Рис. 3.4.1. Общая структура паттерна Prototype

Участники паттерна:

- **Prototype** – прототип

- Определяет интерфейс для клонирования самого себя (в языке C# для таких целей можно использовать стандартный интерфейс `ICloneable`).
- `ConcretePrototype` – конкретный прототип
 - Реализует операцию клонирования самого себя.
- `Client` – клиент
 - Создает новый объект, посылая запрос прототипу копировать самого себя.

Отношения между участниками:

- Клиентский код обращается к прототипу с просьбой создать копию самого себя.

3.4.6. Результаты использования паттерна

Как и все порождающие паттерны, прототип скрывает от клиента конкретные классы продуктов, уменьшая тем самым его сложность. Новые классы продуктов можно создавать практически без модификаций клиентского кода.

Добавление/удаление новых типов продуктов во время выполнения. Объекты-прототипы можно создавать и удалять на этапе выполнения. Такое решение представляет большую гибкость по сравнению с `Abstract Factory` или `Factory Method`.

Определение новых типов продуктов без необходимости наследования. Для создания нового типа продукта надо определить его прототип, что, в общем случае, не требует наследования. Это, зачастую, позволяет избавиться от больших иерархий классов, которые требуются при использовании паттернов `Abstract Factory` или `Factory Method`.

Использование диспетчера прототипов. Зачастую для управления прототипами в системе используется специальный диспетчер, в котором регистрируются прототипы. Класс-диспетчер позволяет получать необходимый прототип за его именем, а также динамически добавлять или удалять прототипы.

Отсутствие параметров в методе `Clone()` обеспечивает наиболее общий интерфейс для клонирования.

Реализация метода Clone() – наиболее трудная часть при использовании паттерна. Особенности проявляются тогда, когда надо клонировать объекты, которые он инкапсулирует, а также в случае наличия круговых ссылок. Зачастую, базовая реализация клонирования уже реализована на уровне основной библиотеке языка программирования (например, в C# или Java).

3.4.7. Практический пример использования паттерна

Представим себе, что перед нами стоит задача разработать программное обеспечение для магазина компьютерной техники. По мнению заказчика, одной из наиболее востребованных возможностей программы будет возможность быстрого создания плана конфигурации системного блока на основе уже имеющихся проверенных шаблонов. Предполагается, что в реализуемой программе будет палитра типичных конфигураций. Для создания экземпляра конфигурации под конкретного клиента пользователю программы надо «вытащить» её прототип из палитры типичных конфигураций. В результате чего программа создаст экземпляр конфигурации, сделав копию прототипа. Созданную копию пользователь будет иметь возможность, в случае надобности, немного модифицировать и представить клиенту.

Отметим, что похожая задача уже рассматривалась нами в п. 3.1 при изучении паттерна Abstract Factory.

Для простоты изложения данной идеи предположим, что в состав конфигурации системного блока входят такие модули:

1. Бокс (Box).
2. Оперативная память (Memory).
3. Жесткий диск (Hdd).
4. Материнская плата (Mainboard).
5. Процессор (Processor).
6. Видеокарта (Videocard).

Классы аппаратных модулей логично было бы наследовать от общего абстрактного класса AbstractDevice.

Объекты представленных классов должны иметь возможность клонировать сами себя. Воспользуемся для этого стандартным механизмом клонирования C#. Реализуем в классе `AbstractDevice` стандартный интерфейс `ICloneable` и соответственно его метод `Clone()`. Код класса `AbstractDevice` представлен в листинге 3.4.1.

Листинг 3.4.1. Класс `AbstractDevice`

```
public abstract class AbstractDevice : ICloneable
{
    public string Producer { get; set; }

    public virtual object Clone()
    {
        return MemberwiseClone();
    }
}
```

Следует обратить внимание, что метод `Clone()` возвращает тип `object`. Это обеспечивает возможность реализации интерфейса `ICloneable` для любого класса C#.

В реализации метода `Clone()` вызывается метод `MemberwiseClone()`, принадлежащий классу `object`. Этот метод обеспечивает следующие:

- 1) создание объекта того класса, на экземпляре которого он вызван;
- 2) автоматическое создание поверхностной копии объекта.

Напомним, что при поверхностном копировании объекта создаются копии полей типов-значений, а для типов-ссылок происходит только копирование ссылки на объект, а не самого объекта. В нашем случае в классе `AbstractDevice` есть одно поле типа `string`. Вызов метода `MemberwiseClone()` обеспечивает копирование ссылки на объект соответствующий строки. Но в силу того, что строки в C# являются неизменяемыми объектами, нет смысла создавать глубокую копию, т.е. клонировать и сам объект строки, для свойства `Producer`.

Диаграмма классов, представляющая аппаратные устройства, изображена на рис. 3.4.2. Назначение атрибутов этих классов соответствуют их названию, поэтому не требуют дополнительных комментариев.

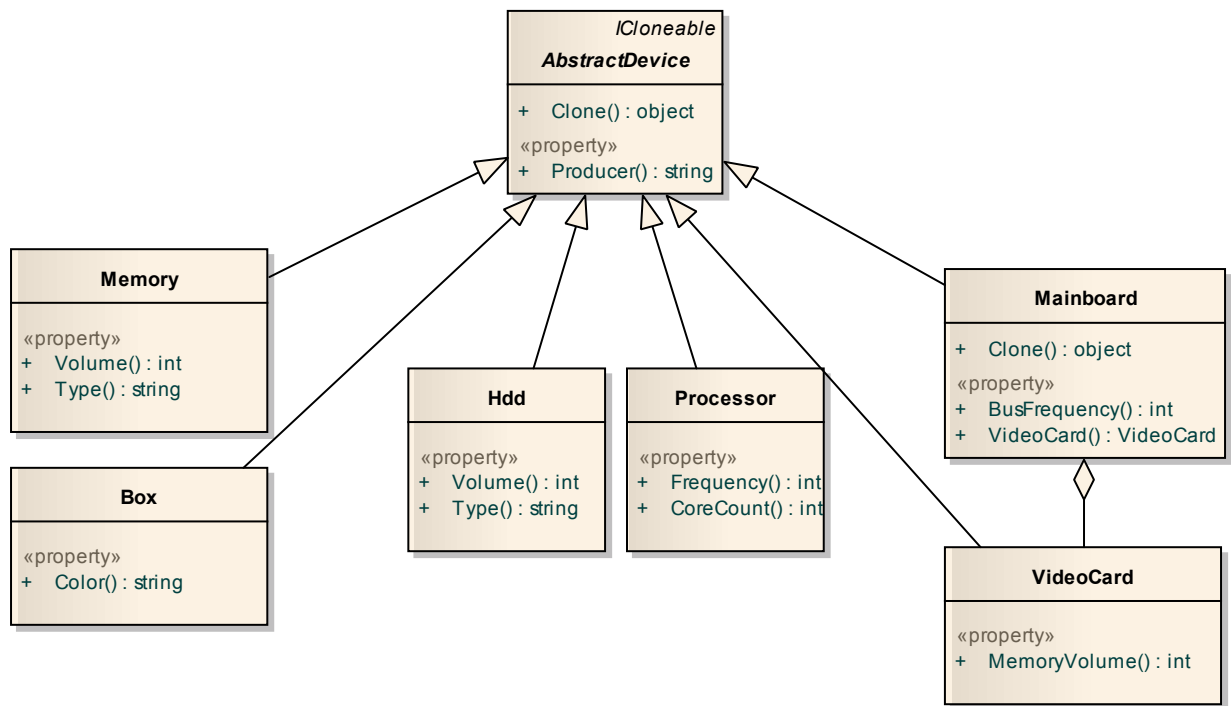


Рис. 3.4.2. Классы аппаратных устройств

Типы полей всех конкретных классов аппаратных устройств, кроме Mainboard, принадлежат к типам-значениям либо к типу string, поэтому для клонирования их объектов не надо переопределять метод Clone() родительского класса. Как мы знаем, материнская плата может содержать встроенную видеокарту. Поскольку объект класса VideoCard не является типом-значением, то для класса Mainboard нужно переопределить метод Clone() так, чтобы он обеспечивал глубокое копирование. Реализация класса Mainboard представлена в листинге 3.4.2.

Листинг 3.4.2. Класс Mainboard

```

public class Mainboard : AbstractDevice
{
    public int BusFrequency { get; set; }
    public VideoCard VideoCard { get; set; }

    public override object Clone()
    {
        Mainboard newObj = (Mainboard)base.Clone();
        newObj.VideoCard =
            this.VideoCard != null ?
            (VideoCard) this.VideoCard.Clone() : null;
        return newObj;
    }
}
  
```

Как видно из листинга 3.4.2, метод Clone() класса Mainboard сначала вызывает метод Clone() базового класса, что обеспечивает создание объекта класса Mainboard и заполнения его полей копиями полей текущего объекта (т.е. создание поверхностной копии). Следующим шагом является клонирование поля Videocard. Поскольку системная плата может и не содержать встроенной видеокарты, то с помощью тернарного оператора проверяется это поле на равенство null, и если оно не равно null, проводится его копирование посредством вызова метода Clone() класса Videocard.

Рассмотрим теперь класс PersonalComputer (рис. 3.4.3).

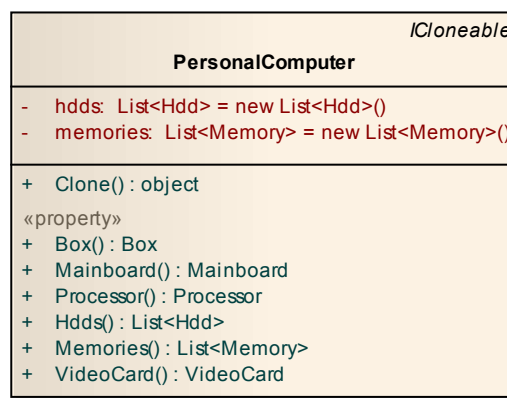


Рис. 3.4.3. Класс PersonalComputer

Обратим внимание на реализацию его метода клонирования (листинг 3.4.3).

Листинг 3.4.3. Реализация метода Clone() класса PersonalComputer

```

public object Clone()
{
    PersonalComputer newPc =
        (PersonalComputer) this.MemberwiseClone();
    newPc.Box = (Box) this.Box.Clone();
    newPc.Mainboard = (Mainboard) this.Mainboard.Clone();
    newPc.Processor = (Processor) this.Processor.Clone();
    newPc.memories = new List<Memory>();
    foreach (Memory m in this.memories)
    {
        newPc.memories.Add((Memory)m.Clone());
    }
    foreach (Hdd h in this.hdds)
    {
        newPc.hdds.Add((Hdd)h.Clone());
    }
    newPc.VideoCard = (VideoCard) this.VideoCard.Clone();
}
  
```

```
        return newPc;  
    }
```

Вначале посредством метода `MemberwiseClone()` создается поверхностная копия объекта класса `PersonalComputer`. Потом создается копия для каждого из его полей, принадлежащих к типу-ссылке. Следует отметить, что при клонировании полей полезно было бы проверять ссылки на равенство `null`, так как вызов метода `Clone()` для `null`-значения вызовет исключение типа `NullPointerException`.

Палитру типичных конфигураций представим в виде класса `PcPrototypes`, который определим как одиночку, или `Sigleton` (см. пункт 3.5), т.е. класс, для которого можно создать только один объект. Такая реализация определена тем, что в нашей программе должен существовать только один экземпляр палитры.

Объект класса `PcPrototypes` дает возможность выбрать прототип конфигурации на основе её имени. В листинге 3.4.4 представлен фрагмент реализации класса `PcPrototypes`.

Листинг 3.4.4. Фрагмент реализации класса `PcPrototypes`

```
/*  
 * Класс палитры типичных конфигураций  
 */  
public class PcPrototypes  
{  
    /*  
     * Единственный экземпляр палитры типичных конфигураций  
     */  
    private static PcPrototypes instance = null;  
    /*  
     * Свойство доступа к единственному экземпляру  
     * палитры типичных конфигураций  
     */  
    public static PcPrototypes Instance  
    {  
        get  
        {  
            if (instance == null)  
            {  
                // если палитра еще не создана, она создается  
                instance = new PcPrototypes();  
            }  
            return instance;  
        }  
    }  
}  
/*  
 * Словарь, в котором хранятся прототипы конфигураций  
 */
```

```
private Dictionary<string, PersonalComputer> pcPrototypes
    = new Dictionary<string, PersonalComputer>();

/*
 * Защищенный конструктор
 * запрещает создавать объект PcPrototypes
 * из внешнего кода
 */
protected PcPrototypes()
{
    // инициализация палитры прототипами конфигураций
    InitializePcPrototypes();
}

/*
 * Доступ к прототипу конфигурации по его имени
 */
public PersonalComputer this [string key]
{
    get
    {
        return pcPrototypes[key];
    }
    set
    {
        pcPrototypes[key] = value;
    }
}

/*
 * Список имен прототипов конфигураций в палитре
 */
public List<string> Keys
{
    get
    {
        return pcPrototypes.Keys.ToList();
    }
}

/*
 * Начальная инициализация палитры конфигураций
 */
private void InitializePcPrototypes()
{
    pcPrototypes["Home"] = CreateHomeConfig();
    pcPrototypes["Office"] = CreateOfficeConfig();
    pcPrototypes["Gamer"] = CreateGamerConfig();
}

...
}
```

Простейший вариант клиентского кода, демонстрирующий использование разработанной палитры прототипов, представлен в листинге 3.4.5.

Листинг 3.4.5. Клиентский код, использующий палитру прототипов конфигураций

```
static void Main(string[] args)
{
    // Ввести имя прототипа конфигурации
    Console.Write("Enter config prototype name: ");
    string prototypeName = Console.ReadLine();
    // Получение прототипа конфигурации из палитры по его имени
    PersonalComputer prototype =
        PcPrototypes.Instance[prototypeName];
    if (prototype != null)
    {
        // Создание конфигурации на основе выбранного прототипа
        PersonalComputer pc = (PersonalComputer)prototype.Clone();
        // Вывод состава конфигурации на консоль
        PrintPc(pc);
    }
    else
    {
        Console.WriteLine("Error: incorrect config name");
    }
}
```

Суть кода из листинга 3.4.5 такова:

- 1) из консоли вводится имя прототипа конфигурации;
- 2) этот прототип выбирается с палитры;
- 3) на основе выбранного прототипа создается объект класса PersonalComputer.

Представленная идея реализации палитры конфигураций обладает такими преимуществами:

- 1) клиентский код избавлен от необходимости определять начальное состояние объекта конфигурации, так как оно определено в прототипе;
- 2) клонирование прототипов реализуется посредством унифицированного интерфейса, предлагающего метод Clone() для создания копии самого себя;
- 3) в отличие от паттерна Abstract Factory, количество продуктов, создаваемых посредством прототипов, не определяется статично, оно может меняться динамически в процессе выполнения программы.

Отметим также, что для представленной архитектурной модели не является проблемой возможность модификации палитры типичных конфигураций во время работы программы.

3.5. Singleton

3.5.1. Название паттерна

Singleton/Одиночка.

Описан в работе [GoF95].

3.5.2. Цель паттерна

Предоставить гарантии, что у класса существует только один экземпляр, и даёт единую точку доступа к нему.

3.5.3. Паттерн следует использовать когда...

- Должен существовать только один экземпляр заданного класса, доступный всему клиентскому коду.

3.5.4. Причины возникновения паттерна

Часто в программировании случается ситуация, когда надо определить некоторую переменную, которая доступна глобально и может существовать только в одном экземпляре. Например, для доступа в систему бухгалтерского учета организации пользователю необходимо пройти аутентификацию, указав свое имя и пароль. После успешной аутентификации пользователь получает доступ к системе. При этом в каждый момент времени системе должна быть известны персональные данные пользователя: его полное имя, уровень доступа и, возможно, другие. Для реализации такой возможности следует определить некоторый объект, который доступен из каждой точки системы (например, для определения того, есть ли доступ в пользователя к определенным возможностям системы) и может существовать только в одном экземпляре (так как одновременно в системе не могут аутентифицироваться два и больше пользователей).

В языке C++ это можно реализовать посредством использования глобальной переменной или статического поля класса. Но такой подход имеет недостатки:

- 1) клиентский код имеет свободный доступ к этой переменной, что может привести к её несанкционированным изменениям;

2) в клиентском коде существует потенциальная возможность создания более одного экземпляра такой переменной.

Следует отметить, что в полностью объектно-ориентированных языках, какими являются C# и Java, вовсе исключена возможность определения глобальных переменных, поэтому в таких языках для определения глобального состояния можно использовать только статические поля класса.

Для преодоления недостатков представленной идеи воспользуемся таким подходом:

1. Возможность создания собственного экземпляра должен иметь только сам класс. В таком случае легко проконтролировать количество созданных экземпляров и, при необходимости, ограничить его одним объектом.

2. Доступ к своему единственному экземпляру также должен предоставлять сам класс.

3. Предусмотреть возможность модификации поведения объекта-одиночки через наследование так, что при подмене единственного экземпляра объектом его подкласса, клиентский код не должен требовать модификаций.

Техническая реализация предложенного подхода может выглядеть так:

1. Для запрета создания экземпляров класса внешним кодом его конструктор (или конструкторы) определяется как `protected`.

2. Доступ к экземпляру класса осуществляется посредством его публичного статического метода.

3. Члены класса единственного экземпляра (кроме переменной, в которой сохраняется ссылка на единственный объект, и метода предоставления доступа к этой переменной) должны быть нестатическими. Это позволит, при надобности, модифицировать поведение класса, наследовав от него подклассы.

3.5.5. Структура паттерна

Общая структура паттерна представлена на диаграмме 3.5.1.

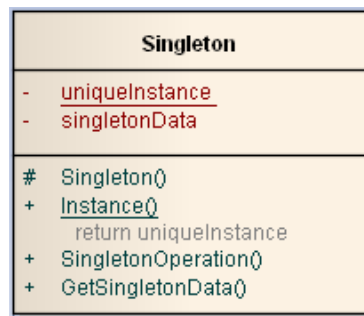


Рис. 3.5.1. Общая структура паттерна Singleton

Участники паттерна:

- Singleton – одиночка
 - Обеспечивает создание только одного экземпляра самого себя, ссылка на который сохраняется в статической переменной `uniqueInstance`, и глобальный доступ к нему через статический метод `Instance()`.
 - Запрещает клиентскому коду создавать собственные экземпляры, запретив ему доступ к своему конструктору (конструктор одиночки определяется как защищенный или приватный).

Отношения:

- Клиентский код имеет возможность доступа к экземпляру Singleton только через его статический метод `Instance()`.

3.5.6. Результаты использования паттерна

Есть возможность полного контроля доступа клиента к единственному экземпляру. Что, например, может дать возможность подсчитать количество клиентских обращений или запрещать доступ в случае отсутствия соответствующих прав.

Единственные экземпляры объекта обладают большими возможностями, нежели обычные глобальные переменные.

Допускается уточнение поведения посредством наследования.

Допускается возможность создания более одного экземпляра. Для этого достаточно модифицировать метод `Instance()`.

Использование одиночки представляет более гибкие возможности, нежели статические функции классов или статические классы C#, так как такие программные конструкции, как правило, обладают рядом ограничений, делающих не возможным определение виртуальных функций с последующим замещением их в подклассах.

Использование глобальных переменных и одиночек способствует созданию «лгущих» интерфейсов, т.е. интерфейсов, которые не показывают явных зависимостей с другими необходимыми для его функционирования объектами.

При реализации одиночки в многопоточной среде следует также учитывать тот факт, что два параллельных потока могут одновременно получать доступ к методу создания единственного экземпляра, что может привести к созданию более одного экземпляра одиночки. Во избежание ситуации «борьбы за ресурсы», следует код, ответственный за создание объекта, поместить в критическую секцию, предоставив тем самым доступ к нему только одному потоку.

3.5.7. Практический пример использования паттерна

Для каждой программы полезно вести специальный журнал, в котором фиксируются определенные события. В такой журнал, например, можно записывать все исключительные ситуации, возникшие в процессе работы. Потом такие записи можно тщательно проанализировать и обнаружить проблемные или дефектные части системы. Или, в случае жалоб пользователей, можно определить характер и причину сбоев. Подобного рода журналы также часто используются для вывода сообщений в процессе отладки программы.

В нашем случае к программному журналу выставляются следующие требования:

1. Сообщения журнала должны выводиться в специальный текстовый файл.
2. В программе должен присутствовать только один экземпляр журнала.
3. Журнал должен работать в многопоточной среде.

Для реализации класса журнала, назовем его `Logger`, воспользуемся идеей паттерна `Singleton`.

В классе `Logger` объявим статическое поле типа `Logger` с названием `instance`, в котором будет храниться единственный экземпляр. Для доступа к этому полю из внешнего кода создадим статическое свойство `Instance`. Задачей этого свойства является возвращение ссылки на единственный экземпляр. В случае равенства `instance` `null` свойство `Instance` обеспечивает создание его объекта. Таким способом реализуется создание журнала по требованию: если в процессе работы программы не было обращений к журналу, то его экземпляр не создается. Такая реализация демонстрирует подход позднего связывания одиночки.

Для исключения возможности создания более одного экземпляра `Logger` в процессе работы программы возможность создавать его объект должен иметь только сам класс `Logger`. С этой целью единственный конструктор класса объявляется защищенным.

Полная реализация класса `Logger` представлена в листинге 3.5.1.

Листинг 3.5.1. Реализация класса `Logger`

```
/*
 * Класс журнала событий программы.
 * Предназначение - запись событий в специальный текстовый файл.
 * В программе может существовать только в одном экземпляре.
 */
public class Logger
{
    // Объект для синхронизации
    private static object sync = new object();

    private static Logger instance = null;
    /*
     * Свойство, предоставляющее доступ к единственному экземпляру.
     * Демонстрирует пример позднего связывания одиночки.
     */
    public static Logger Instance
    {
        get
        {
            /*
             * Критическая секция.
             * Исключает возможность одновременного создания
             * нескольких экземпляров Logger
             * из параллельных потоков.
             */
            lock (sync)
            {
                if (instance == null)
                {
```

```
        instance = new Logger();
    }
    return instance;
}
}
}
/*
 * Защищенный конструктор.
 * Его вызов не возможен из внешнего кода.
 * Следовательно только сам класс Logger
 * может создавать собственные экземпляры.
 */
protected Logger()
{
}
/*
 * Метод вывода сообщения в журнал событий.
 */
public void PutMessage(string message)
{
    /*
     * Критическая секция,
     * исключающая возможность одновременной записи
     * в файл журнала несколькими параллельными потоками.
     */
    lock (sync)
    {
        using (StreamWriter writer =
            new StreamWriter(
                new FileStream("log.txt",
                    FileMode.Append, FileAccess.Write)))
        {
            writer.WriteLine("{0}: {1}", DateTime.Now, message);
        }
    }
}
}
```

Напомним, что класс `Logger` должен использоваться в многопоточной среде. Поэтому следует обезопаситься от проблем, которые могут возникнуть в следствии борьбы за ресурсы:

- 1) в случае одновременного доступа к свойству `Instance` двух потоков, когда экземпляр еще не создан, возможна ситуация создания двух экземпляров;
- 2) когда больше одного потока смогут одновременно получить доступ к файлу журнала, возможно искажение данных.

Для избежания этих ситуаций определена статическая переменная `sync`, используемая в методах доступа к свойству `Instance` и `PutMessage()` для синхронизации критических секций.

Продemonстрируем теперь возможность раннего связывания при создании экземпляра одиночки (листинг 3.5.2).

Листинг 3.5.2. Реализация раннего связывания при создании экземпляра одиночки

```
public class Logger
{
    ...
    /*
     * Единственный экземпляр класса.
     * Создается при инициализации класса.
     * Демонстрирует пример раннего связывания одиночки.
     */
    private readonly static Logger instance = new Logger();
    /*
     * Свойство, предоставляющее доступ к единственному экземпляру.
     */
    public static Logger Instance
    {
        get
        {
            return instance;
        }
    }
    ...
}
```

Заметим, что на этот раз статическое поле `instance`, указывающее на единственный экземпляр, инициализируется при загрузке класса. Более того, его значение не может изменяться в процессе работы программы, так как оно объявлено `readonly`. При такой реализации также упрощается метод доступа к полю, в частности теперь нет необходимости использовать в нем критическую секцию. Недостатком использования раннего связывания есть то, что объект-одиночка создается и в том случае, когда он ни разу не используется в программе. При такой реализации также следует учитывать особенности C#: создание экземпляра одиночки происходит не после начала работы программы, а при статической инициализации класса `Logger`, которая происходит на этапе выполнения непосредственно перед началом его использования (если на этапе выполнения класс ни разу не использовался, то его статическая инициализация также не происходит).

3.6. Анализ и сравнение порождающих паттернов

Есть два наиболее распространенных способа параметризации системы классами объектов, которые создаются.

Первый способ заключается в применении наследования: для создания нового объекта создается новый класс. Такой подход соответствует паттерну Factory Method. Как уже говорилось ранее, основными недостатками этого подхода являются статическое определение классов создаваемых объектов и опасность создания большого количества классов, которые ничего не делают, кроме как реализуют фабричный метод, посредством которого создаются объекты.

Второй способ основывается на композиции: определяется объект, которому известно о классах объектов-продуктов. Этот объект делается параметром системы и может изменяться на этапе выполнения. Такой подход характерный для паттернов Abstract Factory, Builder, Prototype. Каждый из этих паттернов создает специальный «фабричный объект», который создает продукты. В случае Abstract Factory такой объект отвечает за создание семейства продуктов. При использовании Builder фабричный объект отвечает за пошаговое создание сложного продукта. Паттерн Prototype предусматривает объект, который умеет клонировать сам себя.

Несколько обособленно среди других порождающих паттернов стоит Singleton. Его предназначение – создание глобального объекта-одиночки. Часто абстрактная фабрика реализуется как одиночка, если нет смысла создавать для нее более одного экземпляра. Singleton также может использоваться при реализации паттерна Prototype. Пример этому – реализация палитры конфигураций, описанная в пункте 3.4.7.

Поскольку Singleton представляет собой аналог глобальной переменной, а работать с глобальными переменными всегда надо с особой осторожностью, то в современном программировании его относят к антипаттернам и стараются по возможности не использовать.

4. Домашнее задание

1. Спроектировать универсальный каркас многодокументного редактора.

Редактор должен представлять основные функции работы с документом:

1. Создание.
2. Открытие.
3. Сохранение.
4. Сохранение под новым именем.
5. Печать.
6. Закрытие.

Предложенный объектно-ориентированный дизайн каркаса редактора должен без изменений использоваться для разработки редакторов документов различных типов.

2. На основании каркаса, разработанного в задаче 1, спроектировать редактор, предназначенный для работы с текстовыми документами.

3. На основании каркаса, разработанного в задачи 1, спроектировать редактор, предназначенный для работы с графическими документами различных форматов. Редактор обязательно должен иметь возможность сохранять изображение в выбранном графическом формате, а также иметь палитру инструментов для обработки изображения.

Рекомендованные информационные источники

- [GoF95] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб: Питер, 2001. – 386 с.
- [DPWiki] [Шаблоны проектирования \(Wikipedia\)](#)
- [DPOverview] [Обзор паттернов проектирования](#)