Министерство образования Республики Беларусь

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет прикладной математики и информатики

Кафедра информационных систем управления

Трофимов Александр Сергеевич

**Паттерны проектирования**

**Стратегия, Наблюдатель, Декоратор, Фабрика, Шаблонный метод**

Реферат

Студента 1 курса 2 группы

**Преподаватель**

Баранов Геннадий Аркадьевич

Минск, 2017

**РЕФЕРАТ**

Реферат, 18с., 2 источника.

**Ключевые слова:** ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ, ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, СТРАТЕГИЯ, НАБЛЮДАТЕЛЬ, ДЕКОРАТОР, ФАБРИКА, ШАБЛОННЫЙ МЕТОД

**Объект исследования** – Шаблоны (паттерны) проектирования архитектуры компьютерных программ

**Цель работы** – изучить шаблоны проектирования и их применение

**Методы исследования** – изучение каталога шаблонов, применение их при написании приложений

**Область применения –** создание всевозможных приложений, программ, систем с использованием объектно-ориентированных языков программирования.

**ВВЕДЕНИЕ**

При проектировании архитектуры приложений разработчиками часто решаются схожие проблемы. Естественно, существует множество уже устоявшихся типов организации взаимосвязей между элементами в программе, которые применяются в зависимости от поставленной задачи. Такие решения оказались успешными в прошлом, а значит целесообразно использовать их для проектировки новых приложений. Во многих объектно-ориентированных системах можно встретить повторяющиеся шаблоны, состоящие из классов и взаимодействующих объектов. С их помощью решаются конкретные задачи проектирования, в результате чего дизайн приложения становится более гибким, элегантным, и им можно воспользоваться повторно.

Такие подходы к организации систем и называются шаблонами (от английского *pattern* – шаблон) проектирования. Проектировщик, знакомый с шаблонами, может сразу же применять их к решению новой задачи, не пытаясь каждый раз придумывать то, что уже было создано другими разработчиками. Паттерны проектирования упрощают повторное использование удачных проектных и архитектурных решений.

В данной работе предоставляется обзор следующих паттернов:

* «Стратегия»
* «Наблюдатель»
* «Декоратор»
* «Фабрика»
* «Шаблонный метод»

Конечено, количество шаблонов проектирования не ограничивается лишь только перечисленными выше. Всего их существует очень большое количество. И одной из самых сложных задач проектировщика является выбор используемых паттернов при моделировании системы приложения. Для этого необходимо ясно представлять все объекты, взаимодейстующие между собой, характер их связей, а также учитывать, как возможные изменения, которые будут внесены в систему, могут повлиять на архитектуру. С опытом приходит понимаение того, какими шаблонами воспользоваться, как их изменить и адаптировать к конкретной задаче. Но для этого необходимо обладать хорошими знаниями относительно существующих паттернов и их всевозможных вариантов использования.

**СТРАТЕГИЯ**

Очень важным пунктом в поддержании уже готовы приложений является их изменение в соответсвии с теми или иными требованиями. Встаёт вопрос: как именно проводить данные изменения, чтобы сделать это наиболее простым способом и с минимальной вероятностью внесения ошибок в код.

Допустим имеется приложение, которое симулирует поведение животных. Некоторые умеют плавать, некоторые – летать, некоторые – ходить. Как тогда реализовать архитектуру приложения? Можно создать абстрактный класс «Животное» с набором методов «Плавать», «Летать», «Ходить», которые будут переопределены у потомков базового класса. Но проблема заключается в том, что, допустим, некоторые рыбы плавают идентичным образом, значит, реализация метода «Плавать» повлечёт за собою дублирование кода. А для того, чтобы изменить их поведение при плавании, необходимо будет изменять реализацию всех таких классов, что увеличивает риск внесения ошибок в код. Некоторые животные, например медведи, не умеют летать. Как учесть этот факт? А что, если в процессе выполнения программы, нужно будет изменить модель поведения белки при полёте? То есть, допустим, превратить обыкновенную белку в белку-летягу. Конкретная реализация метода в классе-наследнике не даёт такие возможности.

На помощь приходит один из основных принципов проектирования систем: «Выделите части, которые изменяются и инкапсулируйте их, чтобы они не влияли на работу остального кода». В данном случае необходимо инкапсулировать этот самый полиморфный метод. Превосходным решением является создание интерфейса, в котором и будет находиться данный метод и создание соответствующих классов, которые будут лишь предоставлять реализацию определённой модели поведения. Например создадим интерфейс «ИПлавать» , который содержит метод «Плавать», и классы, соответсвенно предоставляющие реализацию данного метода. Тогда у объекта класса «Животное» можно создать поле типа интерфейса, а при конструировании объекта, передать ему соответствующую реализацию («Плавать на поверхности воды», «Плавать в глубине», «Не плавать»). Возможно без проблем изменять модель поведения объекта прямо во время выполнения программы. А изменения в поведение будут вноситься лишь в единственном месте во всём коде – в классе, предоставляющем соответственную реализацию.

Кроме того доступ к данному интерфейсу (моедли поведения) теперь будут иметь не только объекты класса «Животное», но и объекты совершенно другого типа, что решает другую проблему – повторного использования кода.

Данный пример демонстрирует возможности паттерна «Стратегия». Дадим ему конкретное определение:

Шаблон «**Стратегия**» определяет семейство алгоритмов, инкапсулирует их и обеспечиваех их взаимозаменяемость. Паттерн позволяет модифицировать алгоритмы независимо от их использования на стороне клиента.

Методика создания нового класса из уже существующих носит название **композиции**. Системы, созданные на базе композиции обладают большой гибкостью.

Такой подход позволяет абстрагироваться от явного указания действий, он позволяет перейти на более высокий уровень – архитектуры.

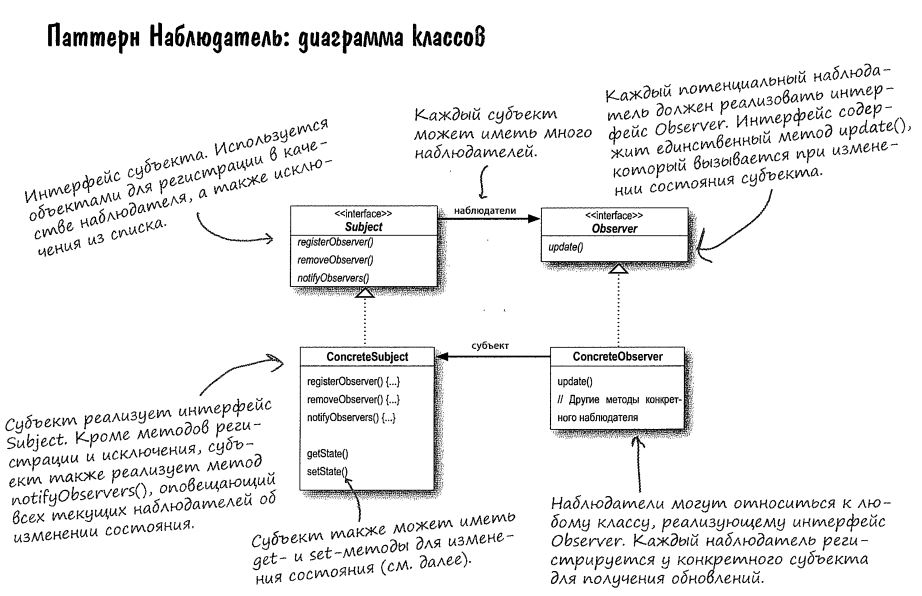
Динамическая композиция объектов позволяет добавлять новую функциональность посредством написания нового кода, что позволяет избежать необходимость изменения уже существующего. Поэтому, из-за того, что готовый код остаётся неизменным, риск возникновения ошибок или непредвиденных побочных эффектов снижается.

**НАБЛЮДАТЕЛЬ**

При создании оконных приложений необходимо добавление элементов взаимодействия с пользователем. Простейшим элементом является кнопка. Но как обеспечить взаимодействие между нажатием на кнопку и многочисленными процессами, которые могут запускаться в момент нажатия? В данном случае поможет использование шаблона «Наблюдатель»

Паттерн “Наблюдатель” определяет отношение “один-ко-многим” между объектами таким образом, что при изменении состояния одного объекта происходит автоматическое оповещение и обновление всех зависимых объектов.

Ниже приведена общая схема, иллюстрирующая данный шаблон:

****

В паттерне «Наблюдатель» субъект обладает состоянием и управляет им. Таким образом, существует один субъект, обладающий состоянием. С другой стороны, наблюдатели используют состояние, хотя и не обладают им. Они зависят от субъекта, который оповещает их об изменении состояния.

В нашем конкретном примере, объектом будет являться кнопка, которая обладает состоянием (она может быть либо нажата, либо нет), а в качестве наблюдателей выступают обработчики события нажатия кнопки.

Так как субъект является единоличным владельцем данных, работа наблюдателей зависи от субъекта, оповещающего их об изменении данных. Так формируется элегантная структура, в которой многие объекты используют одни и те же данные.

Если два объекта могут взаимодействовать, не обладая практически никакой информацией друг о друге, такие объекты называют **слабосвязанными**. В архитектуре паттерна «Наблюдатель» между субъектами и наблюдателями существует слабая связь, так как

* Единственное, что знает субъект о наблюдателе – это то, что тот реализует некоторый интерфейс, благодаря которому субъект может доставлять ему оповещения
* Новые наблюдатели могут добавляться в любой момент
* Добавление новых типов наблюдателей не требует модификации субъектов
* Субъекты и наблюдатели могут повторно использоваться независимо друг от друга
* Изменения в субъекте или наблюдателе не влияют на другую сторону

Важно заметить, что в программе не должен играть роль порядок получения оповещений наблюдателями, так как это приводит к появлению определённых зависимостей, что в свою очередь может повлечь ошибки и неправильные результаты, если заменить логику доставки уведомлений наблюдателям.

Таким образом, данный паттерн позволяет значительно облегчить процесс взаимодействия между различными объектами. Субъекты могут обновлять наблюдателей через единый интерфейс, причём для этого не требуется обладать абсолютно никакой информацией, за исключением знания, что наблюдатели реализуют данный интерфейс. Слабосвязанные структуры обладают большой гибкостью и хорошо выдерживают изменения.

**ДЕКОРАТОР**

Шаблон проектирования «Декоратор» динамически наделяет объект новыми возможностями и является гибкой альтернативой субклассированию в области расширения функциональности.

Данный паттерн нацелен на использование с базовым абстрактным классом. Его использование позволяет сделать классы открытыми для расширения функциональности, но быть закрытыми для вненсения изменений.

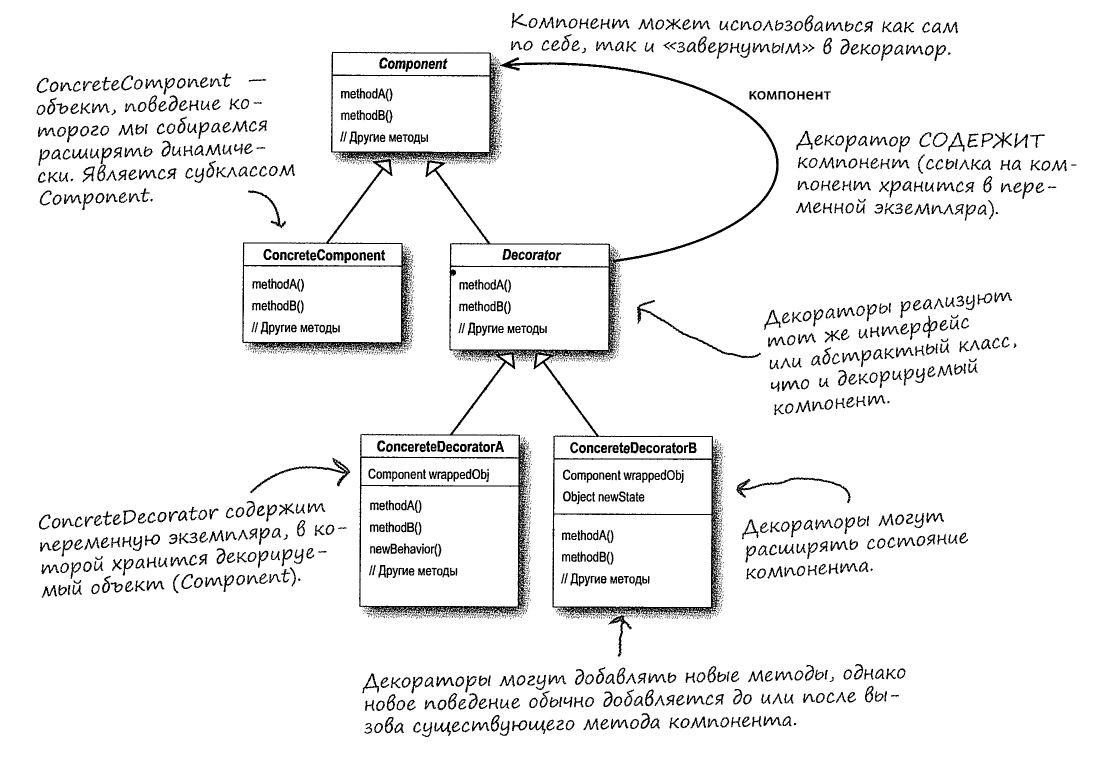
Основные принципы работы паттерна «Декоратор»:

* Декоратор имеет тот же тип, что и декорируемые объекты. Для реализации декораторов используют абстрактный суперкласс объектов, от которого наследуются реальные экземпляры (компоненты), а также абстрактный класс декораторов. При «обёртывании» компонентов в декораторы используются конструкторы декораторов, которые получают в качестве аргумента экземпляр объекта. В результатае создаётся новый экземпляр исходного компонента, но уже наделённый дополнительными свойствами. Однако тип объектов не изменяется.
* Декоратор добавляет своё поведение декорируемому объекту. Поэтому, если необходимо вызвать метод исходного объекта, который был модифицироваван декоратором, сначала вызывается метод последнего, который затем вызывает соответствующий метод декорируемого объекта.   
  Например, необходимо рассчитать стоимость кофе с добавлением взбитых сливок и шоколада. Мы «обернём» объект «Кофе» в декоратор «С шоколадом». Полученный объект «обернём» в «Со взбитыми сливками». Затем вызовем метод «Рассчитать стоимость» у последнего получившегося объекта. Он считает стоимость взбитых сливок и перенаправляет дальнейшее выполнение соответствующему методу объекта «С шоколадом». Он к имеющемуся результату добавляет стоимость шоколадной стружки и вызывает метод подсчёта стоимости объекта «Кофе», который добавляет стоимость чашки кофе и получает общий результат.
* К объекту можно добавить сразу несколько декораторов
* Так как у декоратора имеется тот же тип, что и у декорируемого объекта, возможна передача декорированного объекта вместо исходного
* Объект может быть декорирован в любой момент времени, так что можно декорировать объекты динамически и с произвольным количеством декораторов

Субклассирование базового абстрактного класса необходимо только для приведения к нужному типу, а не для наследования его поведения. Оно формируется посредством композиции декораторов с базовыми компонентами и другими декораторами.

При использовании наследования, то модель поведения определялась бы статически во время компиляции. То есть использовалось бы лишь то поведение, которое предоставляет суперкласс, либо то, что мы переопределили в классе-наследнике. Композиция позволяет смешивание декораторо во время выполнения программы. В любой момент можно добавить новые декораторы, которые наделят субклассновым поведением.

Использование интерфейса вместо базового абстрактного класса также возможно, так как наследуется фактически только лишь тип объекта. Но тогда изменяется базовая модель, к которой мы применяем наш шаблон проектирования, и получается другой паттерн. Поэтому в данной ситуации мы работаем с тем, что у нас имеется и получаем шаблон «Декоратор», использующий базовый абстрактный класс.

Возможны также ошибки и проблемы при испльзовании данного паттерна. Например, существет опасность, что клиент получит ссылку, которая не относится к самому внешнему декоратору, что влечёт за собой неверный результат при вызове метода декоратора, так как воздействие самого внешнего из них не учитывается. В паттерне «Декоратор» возрастает количество объектов, а следовательно, и опасность проблем, связанных с ошибками программирования. Однако, декораторы обычно используются как часть других паттернов («Фабрика», «Строитель»). Создание конкретных компонентов с декораторами в них хорошо инкапсулировано, а ошибки такого рода маловероятны.

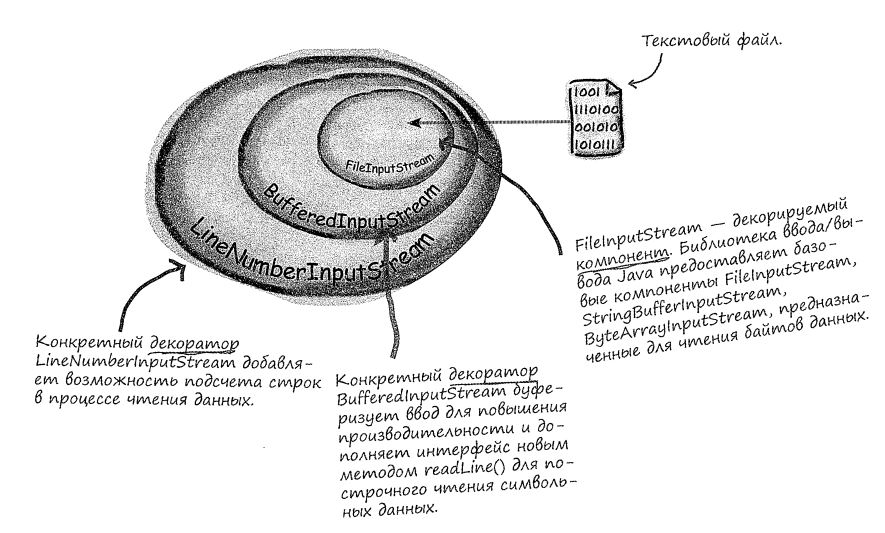
Введение декораторов усложняет код создания экземпляра компонента. Если в архитектуре участвуют декораторы, необъодимо не только создать компонент, но и «обернуть» его в несколько декораторов.

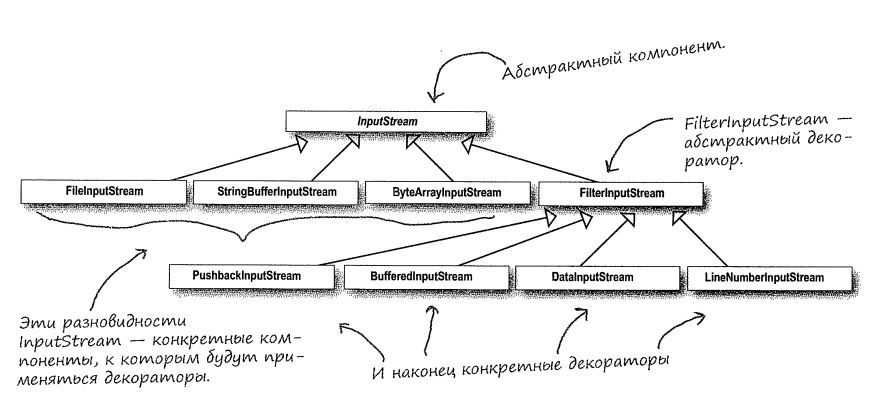
Если код зависит от типа конкретного компонента, декораторы нарушат его работоспособность. Пока код пишется для типа абстрактного компонента, использование декоратора остаётся прозрачным для кода. Если же начать программировать на уровне конкретных компонентов, стоит задуматься об архитектуре приложения и, возможно, изменить используемый шаблон проектирования.

Так как декораторы предназначены для расширения поведения декорируемых объектов, получение информации с различных уровней цепочки декораторов не входит в рамки назначения данного шаблона. Однако, при необходимости, добавить наобходимый функционал прослеживания добавленных паттернов возможно, хоть и не является целью применения данного шаблона.

Рассмотрим реальный пример использования паттерна «Декоратор».

Java .io – пакет Java API для облегчения работы со входными/выходными потоками. Этот пакет в основном базируется на шаблоне «Декоратор». Ниже приведён типичный набор объектов, использующих декораторы для расширения функциональности чтения данных из файла:

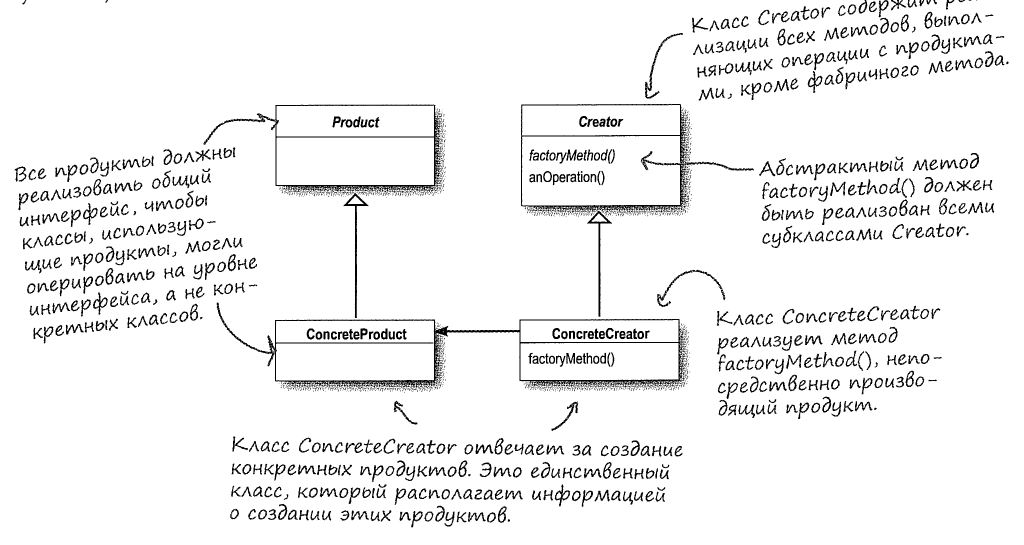
**** BufferedInputStream и LineNumberInputStream декорируют FileInputStream. Они также являются расширениями FilterInputStraem – абстрактного класса декоратора. Общая схема декорирования классов java.io выглядит следующим образом:

 Библиотека ввода/выода Java также подчёркивает один из недостатков шаблона «Декоратор»: иерархии, построенные с использованием этого паттерна, часто состоят из множества мелких классов, в которых бывает трудно разобраться разработчику.

**ФАБРИКА**

В программе, при создании экземпляров различных конкретных классов, может присутствовать выбор: объект какого класса создавать? Этот выбор может зависить отмножества условий. Вместо явного указания и, при необходимосити изменения (достаточно трудоёмкого процеса), в различных местах кода типов создаваемых объектов, процесс создания объектов можно инкапсулировать. Можно создать класс, единственной задачей которого будет создание экземпляров классов. Назовём такой класс Фабрикой. Он может быть использован совершенно разными классами для создания объектов. И будущие изменения реализация создания объектов будут вноситься лишь в одно место – в класс Простой Фабрики. Такая простая Фабрика не является полноценным паттерном, а скорее, это идиома программирования, она обеспечивает простой механизм изоляции клиентов от конкретных классов.

Существует два паттерна проектирования, основанных на реализации идеи Фабрики: «Фабричный метод» и «Абстрактная фабрика»

**«Фабричный метод»** определяет интерфейс создания объекта, но позволяет субклассам выбрать класс создаваемого экземпляра. Данный шаблон основан на наследовании: создание объектов делегируется субклассам.Создаётся инфраструктура, при которой именно субклассы определяют реализацию создания экземпляров

Из вриведённой выше диаграммы видно, что абстрактный класс Creator предоставляет интерфейс к методу создания объектов, также называемому «фабричным методом». Остальные методы, реализуемый в абстрактном классе Creator, работают с продуктами, созданными фабричным методом. Только субклассы фактическиреализуют фабричный метод и создают продукты.

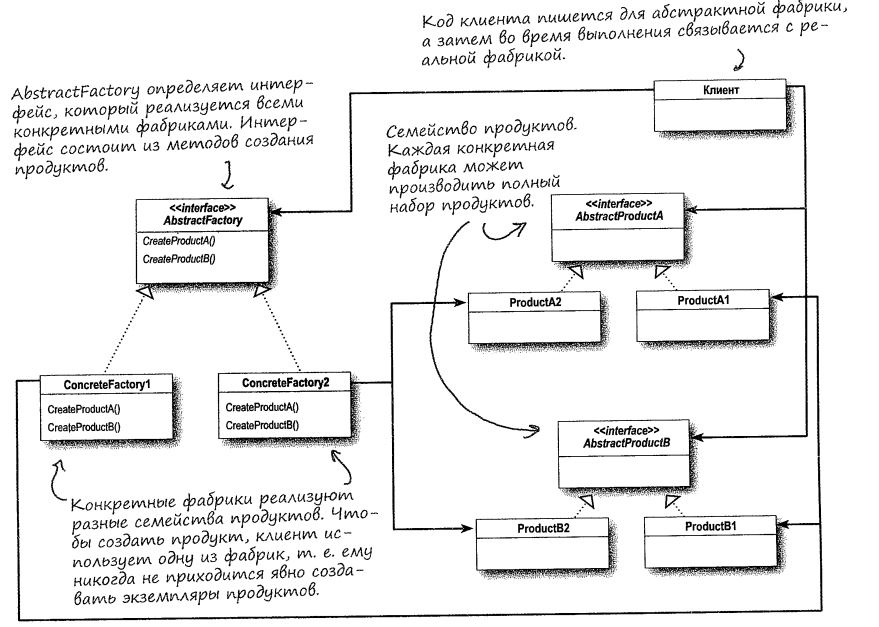
Данный паттерн пригодится даже если в иерархии всего один кокретный создатель – он отделяет реализацию продукта от его использования. Если позднее добавятся другие продукты или изменится реализация продукта, это не отразится на рабте класса-создателя.

Фабричный метод по умолчанию может создавать некий конкретный продукт, то есть быть конкретным, а не абстрактным. Это позволяет создавать продукты даже при отсутствии субклассов у класса-создателя.

Размещение кода создания в одном объекте или методе позволяет избежать дублирования кода и упрощает сопровождение. Кроме того, клиент зависит только от интерфейсов, а не от конкретных классов, необходимых для создания объектов. Программирование на уровне интерфейса делает код более гибким и упрощает его возможное расширение.

Шаблон «**Абстрактная фабрика»** предоставляет интерфейс создания семейств взаимсвязанных или взаимозависимых объектов без указания их конкретных классов. Этот паттерн основан на использовании композиции: создание объектов реализуется в методе, доступ к которому осуществляется через интерфейс фабрики.

Благодаря паттерну «Абстрактная фабрика» клиент может использовать абстрактный интерфейс для создания логически связанных продуктов, не располагая информацией о конкретных создаваемых продуктах. Таким образом, клиент отделяется от подробностей конкретного продукта. Следующая диаграмма разъясняет данную модель поведения:

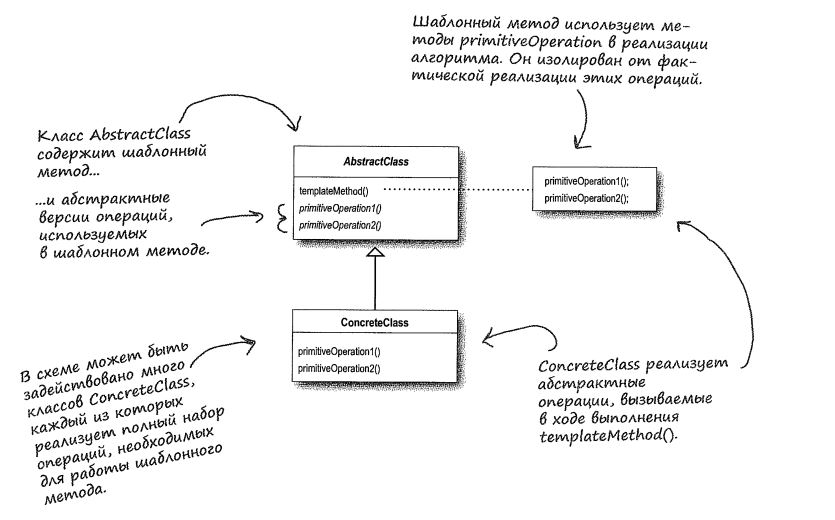
Различия между «Фабричным методом» (ФМ) и «Абстрактной фабрикой» (АФ):

* ФМ использует наследование,АФ – композицию
* Для создания объектов с помощью ФМ необходимо расширить класс и переопределить ФМ. Именно субкласс создаёт объекты. Клиенту достаточно знать абстрактный тип объектов. ФМ создаёт только один тип продуктов  
  АФ предоставляет абстрактный тип для создания целых семейств продуктов. Субклассы этого типа определяют способ создания продуктов. Чтобы использовать фабрику, нужно создать экземпляр и передать его коду, написанному для абстрактного класса.

Все фабричные паттерны обеспечивают слабую связанность за счёт сокращения зависимости приложения от конкретных классов.

**ШАБЛОННЫЙ МЕТОД**

Паттерн «Шаблонный метод» задаёт «скелет» алгоритма в методе, оставляя определение реализации некоторых шагов субклассам. Субклассы могут переопределять некоторые части алгоритма без изменения его струкруты. Данный паттерн существенно улучшает повторное использование кода.

Основной задачей данного паттерна является создание шаблона алгоритма. Он представляет собой метод базового класса, который не может быть переопределён в классах потомков. Данный метод использует другие методы в качестве шагов алгоритма. Некоторые из них определяются в виде абстрактных методов, реализуемых субклассами. Таким образом гарантируется неизменность структуры алгоритма при том, что часть реализации предоставляется субклассами.

Следующая диаграмма классов поясняет структуру данного паттерна:

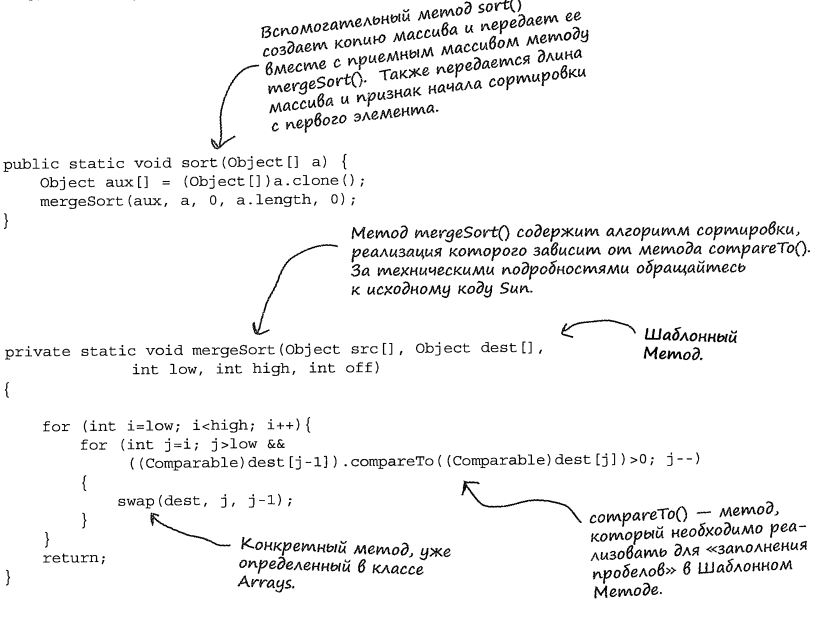
Метод, объявленный абстрактным базовым классом, который может быть переопределён в классе-наследнике (может иметь исходную реализацию, а может быть пустым), называется **перехватчиком**.

Перехватчики используются для реализации необязательной части алгоритма. Если эта часть не важна для реализации субкласса, он её пропускает. Также перехватчик может дать субклассу возможность среагировать на предстоящий или только что выполненный шаг шаблонного метода. Он даёт субклассам возможность «подключаться» к алгоритму в разных точках. Однако субкласс может также проигнорировать имеющиёся перехватчик.

Какие методы базового класса объявлять абстрактными, а какие делать перехватчиками? Если субкласс обязан предоставить реализацию метода, в базовом классе соответствующие методы делаются абстрактными. Для методов-перехватчиков переопределение в субклассах необязательно.

Каждый конкретный субкласс определяет полный набор абстрактных методов и предоставляет полнуюреализацию неопределённых шагов алгоритма шаблонного метода.

При написании шаблонных методов следует помнить о минимизации количества абстрактных методов, так как все они требуют реализации в субклассах. Иногда эта цель достигается за счёт укрупнения шагов алгоритма, но это приводит к потере гибкости. Также следует помнить, что некоторые шаги могут быть необязательными, их следует реализовывать в качестве перехватчиков, чтобы упростить реализацию субклассов.

Данный паттерн очень часто применяется на практике. Рассмотрим пример алгоритма сортировки массива на языке Java.

Шаблонный метод sort из класса Arrays определяет алгоритм, но мы должны указать ему, как сравнить два объекта, а для этого нужно реализовать метод compareTo. Сам массив при этом не субклассирует исходный алгоритм. Дело в том, что проектировщики sort хотели, чтобы метод мог использоваться для всех массивов, поэтому они реализовали sort в виде статического метода. Но это вполне нормальное решение – оно работает почти так же, как метод суперкласса. Но поскольку метод sort всё-таки не определяется в суперклассе, он должен знать, что метод compareTo был реализован, в противном случае у него не хватит информации для определения полного алгоритма сортировки. Для решения этой проблемы проектировщики воспользовались интерфейсом Compareable. От программиста требуется лишь реализовать интерфейс, состоящий из единственного метода compareTo. Последний сравнивает два объекта и возвращает информацию об их соотношении. Метод sort использует его для сравнения объектов в массиве.

В описании паттерна говорится об определении алгоритма с возможностью определения реализации отдельных его шагов в субклассах – сортировка массива под это определение не подходит. Но, как известно, на практике паттерны часто приходится подгонять под ограничения контекста реализации. Проектировщики метода sort столкнулись с препятствием: в общем случае субклассировать массив Java невозможно, а они хотели, чтобы метод sort мог использоваться для всех массивов (хотя разные массивы относятся к разным классам). Исходя из этого, они определили статический метод и поручили определение способа сортируемым объектам. Таким образом, несмотря на отличия от «канонических» шаблонных методов, это реализация вполне соответствует духу паттерна.

Такая реализация алгоритма сортировки напоминает скорее паттерн «Стратегия». Такое впечатление возникает из-за того, что в шаблоне «Стратегия» используется композиция, а мы действительно используем объект Arrays для сортировки массива. Однако в паттерне «Стратегия» включаемый класс реализует весь алгоритм, а алгоритм сортировки, реализованный в Arrays, неполон. Недостающий метод compareTo должен быть предоставлен классом. В этом состоит его сходство с паттерном «Шаблонный метод».

**Сравнение** «Шаблонного метода» (ШМ) и «Статегии» (С):

* С определяет семейство алгоритмов и обеспечивает их взаимозаменяемость. Инкапсуляция позволяет легко использовать разные алгоритмы на стороне клиента  
  ШМ определяет общую структуру алгоритма, поручая часть работы субклассам. Это позволяет данному паттерну выбирать разные реализации отдельных шагов алгоритма, сохраняя под контролем его структуру.
* Системы, использующие С, могут изменять алгоритмы прямо во время выполнения программы, просто переключаясь на другой объект стратегии  
  ШМ предоставляет основной механизм повторного использования кода, позволяющий задавать поведение в субклассах.
* С не зависит от реализации алгоритма в субклассах, в отличие от ШМ.
* Оба паттерна инкапсулируют алгоритмы. ШМ использует наследование, а С – композицию

**ЗАКЛЮЧЕНИЕ**

Знание шаблонов проектирования помогает понимать существующие объектно-ориентированные системы, в большинстве которых они применяются. Для использования паттернов не требуются особенные возможности языка программирования, всё можно реализовать на стандартных объектно-ориентированных языках. Ведь шаблоны проектирования подразумевают собой модель организации кода, выбор взаимосвязей между объектами, и, собственно, выделение определённых частей кода в отдельные объекты. Усилия, приложенные для соответствующей организации кода неизменно окупаются за счёт большей гибкости и возможности повторного использования кода.

**СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ**

1. Эрик Фримен, Элиабет Фримен, «Паттерны Проектирования» - СПб.: Питер, 2011
2. Эрик Гамма, Ричард Хелм, Ральф Джонсон, «Приёмы объектно-ориентированного программирования» - СПб.: Питер, 2010