ШАБЛОНЫ ПРОЕКТИРОВАНИЯ

Лекция 1 –
29.01.2021

Основные определения предмета курса и основные понятия. Часть 1

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

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

«Низкоуровневые» шаблоны, учитывающие специфику конкретного языка программирования, называются **идиомами**. Это хорошие решения проектирования, характерные для конкретного языка или программной платформы, и потому не универсальные.

На наивысшем уровне существуют **архитектурные шаблоны**, они охватывают собой архитектуру всей программной системы.

Алгоритмы по своей сути также являются **шаблонами**, но не проектирования, а **вычисления**, так как решают вычислительные задачи.

Предистория

В 1991 годух Джеймс Коплин (James Coplien) трудился над разработкой идиом для программирования на C++ и опубликовал в 1991 году книгу **Advanced C++ Idioms**.
В 1991 году **Эрих Гамма** 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**. В этой книге описаны **23 шаблона проектирования**. Также команда авторов этой книги известна общественности под названием «**Банда четырёх**» (англ. **Gang of Four**, часто сокращается до **GoF**). Именно эта книга стала причиной роста популярности шаблонов проектирования.

ПОЛЬЗА ШАБЛОНОВ ПРОЕКТИРОВАНИЯ

Основная польза от использования шаблонов состоит в **снижении сложности разработки за счёт готовых абстракций** для решения целого класса проблем:

– Шаблон даёт решению свое имя, что облегчает коммуникацию между разработчиками, позволяя ссылаться на известные шаблоны.

– Снижается количество ошибок, т.к. за счёт шаблонов производится унификация деталей решений: модулей, элементов проекта.

– Применение шаблонов концептуально сродни использованию готовых библиотек кода. Правильно сформулированный шаблон проектирования позволяет, отыскав удачное решение, пользоваться им снова и снова.

– Набор шаблонов помогает разработчику выбрать возможный, наиболее

Зачем применять паттерны проектирования

Рано или поздно все начинающие программисты сталкиваются с проблемой структурирования своего кода. Те из них, кто действительно хочет улучшить качество своего кода, среди прочих вариантов решений проблемы, начинают изучать паттерны проектирования!

Согласно истории появления паттернов, можно сказать, что паттерны проектирования это некоторые объектно-ориентированные архитектурные шаблоны, которые были когда-то и кем-то определены.

Итак, на первый взгляд, это просто случай в программировании, наиболее удачно решенный и определенный как в архитектурный **ООП** шаблон проектирования! Паттерны объединяются в группы:

– по принципу их работы (**Порождающие**, **Структурные**, **Поведенческие**....),

– по области применения (**Паттерны баз данных**, **паттерны корпоративных приложений**...)

Однако, начав изучать паттерны проектирования (а их действительно очень много), в них можно запутаться, а также в области их применения. Вот тут-то и возникает гораздо более важный вопрос, – почему паттерны проектирования такие, какие они есть?!

Дело в том, что **паттерн проектирования** – это не просто **реализация ООП** на практике. **Это реализация ООП в соответствии с принципами проектирования.**

Принципы проектирования (наиболее важные):

1) Предпочитай **композицию наследованию**.

Само по себе наследование – не очень гибкий инструмент, поэтому, если на стадии проектирования вариативное поведение в классе не было выявлено и инкапсулировано в другой класс, можно ждать проблем, связанных с тем, что класс не сможет изменить свое поведение там, где это крайне необходимо, поэтому нужен следующий принцип:

2) Инкапсулируй то, что изменяется.

Если поведение класса приобретает какую-то вариативность, надо это свойство и**нкапсулировать, вынеси его в отдельный класс и применить композицию** (п.1).

3) Класс должен иметь только **одну ответственность**.

Нужно избегать создания классов-богов, которые делают все. К**ласс должен отвечать за что-то одно**.

4) Принцип **закрытости-открытости**. Класс должен **быть закрыт для изменения и открыт для добавления.** К примеру, нужно изменить функционал, – создайте новый класс на основе старого, – **наследуйте**, или используйте паттерн **декоратор**, – но ни в коем случае не меняйте (смотреть п.2).

5) Программируйте на уровне **интерфейса** или абстрактного класса, а не **реализации**.

6) Код должен зависеть от **абстракции**, а не от **конкретной реализации.**

7) Классы никогда не обращаются с вызовами к абстрактному классу, сначала последний обращается к ним. Этот принцип предотвращает разложение зависимостей.

8) Стремитесь к **слабой связанности объектов**. Иначе ваш класс невозможно

Все вышеприведенные **принципы проектирования** неразрывно связаны с современными технологиями **объектно-ориентированного анализа (ООАП)** и **паттернами проектирования.**

Паттерны объектно-ориентированного анализа и проектирования (ООАП)

Паттерны, их классификация

При реализации проектов по разработке программных систем и моделированию бизнес-процессов встречаются ситуации, когда решение проблем в различных проектах имеют сходные структурные черты. Выявление похожих схем или структур в рамках ООАП привели к появлению понятия паттерна, которое из абстрактной категории превратилось в непременный атрибут современных **CASE-средств** .

Паттерны **ООАП** различаются **степенью детализации** и **уровнем абстракции**.

Существует **общая классификация паттернов по категориям их применения:**

Архитектурные паттерны

Паттерны проектирования

Паттерны анализа

Паттерны тестирования

Архитектурные паттерны (**Architectural patterns**) – множество предварительно определенных подсистем со спецификацией их ответственности, правил и базовых принципов установления отношений между ними.

Архитектурные паттерны предназначены для спецификации фундаментальных схем структуризации программных систем. Наиболее известными паттернами этой категории являются паттерны **GRASP** (**General Responsibility Assignment Software Pattern**). Эти паттерны относятся к уровню системы и подсистем, но не к уровню классов. Как правило, формулируются в обобщенной форме, используют обычную терминологию и не зависят от области приложения. Паттерны этой категории систематизировал и описал К. Ларман.

Паттерны проектирования (**Design patterns**) – специальные схемы для уточнения структуры подсистем или компонентов программной системы и отношений между ними.

Паттерны проектирования описывают общую структуру взаимодействия элементов программной системы, которые реализуют исходную проблему проектирования в конкретном контексте.

Наиболее известными паттернами этой категории являются паттерны **GoF** (**Gang of Four**), названные в **честь Э. Гаммы, Р. Хелма, Р. Джонсона** и **Дж. Влиссидеса**, которые систематизировали их и представили общее описание. **Паттерны GoF включают в себя 23 паттерна. Эти паттерны не зависят от языка реализации, но их реализация зависит от области приложения**.

Паттерны анализа (**Analysis patterns**) – специальные схемы для представления общей организации процесса моделирования.

Паттерны анализа относятся к одной или нескольким предметным областям и описываются в терминах предметной области. Наиболее известными паттернами этой группы являются паттерны бизнес-моделирования **ARIS** (**Architecture of Integrated Information Systems**), которые характеризуют абстрактный уровень представления бизнес-процессов. В дальнейшем паттерны анализа конкретизируются в типовых моделях с целью выполнения аналитических оценок или имитационного моделирования бизнес-процессов.

Паттерны тестирования (**Test patterns**) – специальные схемы для представления общей организации процесса тестирования программных систем.

К этой категории паттернов относятся такие паттерны, как **тестирование черного ящика**, **белого ящика**, **отдельных классов, системы**. Паттерны этой категории систематизировал и описал М. Гранд. Некоторые из них реализованы в инструментальных средствах, наиболее известными из которых является **IBM Test Studio**. В связи с этим паттерны тестирования иногда называют **стратегиями** или **схемами тестирования**.

Паттерны реализации (**Implementation patterns**) – совокупность компонентов и других элементов реализации, используемых в структуре модели при написании программного кода.

Эта категория паттернов делится на следующие подкатегории:

паттерны организации программного кода,

паттерны оптимизации программного кода,

паттерны устойчивости кода,

паттерны разработки графического интерфейса пользователя и др.

Паттерны этой категории описаны в работах М. Гранда, К. Бека, Дж. Тидвелла и др. Некоторые из них реализованы в популярных интегрированных средах программирования в форме **шаблонов создаваемых проектов**. В этом случае выбор шаблона программного приложения позволяет получить некоторую заготовку программного кода.

3. Паттерны проектирования классов/объектов

3.1 Структурные паттерны проектирования классов/обьектов

3.1.1 Адаптер (Adapter) – GoF

3.1.2 Декоратор (Decorator) или Оболочка (Wrapper) – GoF

3.1.3 Заместитель (Proxy) или Суррогат (Surrogate) – GoF

3.1.4 Информационный эксперт (Information Expert) – GRASP

3.1.5 Компоновщик (Composite) – GoF

3.1.6 Мост (Bridge), Handle (описатель) или Тело (Body) – GoF

3.1.7 Низкая связанность (Low Coupling) – GRASP

3.1.8 Приспособленец (Flyweight) – GoF

3.1.9 Устойчивый к изменениям (Protected Variations) – GRASP

3.1.10 Фасад (Facade) – GoF

3.2 Паттерны проектирования поведения классов/объектов

3.2.1 Интерпретатор (Interpreter) – GoF

3.2.2 Итератор (Iterator) или Курсор (Cursor) – GoF

3.2.3 Команда (Command), Действие (Action) или Транзакция (Transiction) – GoF

3.2.4 Наблюдатель (Observer), Опубликовать – подписаться (Publish - Subscribe) или Delegation Event Model – GoF

3.2.5 Не разговаривайте с неизвестными (Don't talk to strangers) – GRASP

3.2.6 Посетитель (Visitor) – GoF

3.2.7 Посредник (Mediator) – GoF

3.2.8 Состояние (State) – GoF

3.2.9 Стратегия (Strategy) – GoF

3.2.10 Хранитель (Memento) – GoF

3.2.11 Цепочка обязанностей (Chain of Responsibility) – GoF

3.2.12 Шаблонный метод (Template Method) – GoF

3.2.13 Высокое зацепление (High Cohesion) – GRASP

3.2.14 Контроллер (Controller) – GRASP

3.2.15 Полиморфизм (Polymorphism) – GRASP

3.2.16 Искусственный (Pure Fabrication) – GRASP

3.3 Порождающие паттерны проектирования

3.3.1 Абстрактная фабрика (Abstract Factory, Factory), др. название Инструментарий (Kit) – GoF

3.3.2 Одиночка (Singleton) – GoF

3.3.3 Прототип (Prototype) – GoF

3.3.4 Создатель экземпляров класса (Creator) – GRASP

3.3.5 Строитель (Builder) – GoF

3.3.6 (Фабричный метод) Factory Method или Виртуальный конструктор (Virtual Constructor) – GoF

4 Архитектурные системные паттерны

4.1 Структурные паттерны

4.1.1 Репозиторий

4.1.2 Клиент/сервер

4.1.3 Объектно - ориентированный, Модель предметной области (Domain Model), модуль таблицы (Data Mapper)

4.1.4 Многоуровневая система (Layers) или абстрактная машина

4.1.5 Потоки данных (конвейер или фильтр)

4.2 Паттерны управления

4.2.1 Паттерны централизованного управления

4.2.1.1 Вызов - возврат (сценарий транзакции – частный случай).

4.2.1.2 Диспетчер

4.2.2 Паттерны управления, основанные на событиях

4.2.2.1 Передача сообщений

4.2.2.2 Управляемый прерываниями

4.2.3 Паттерны, обеспечивающие взаимодействие с базой данных

4.2.3.1 Активная запись (Active Record)

4.2.3.2 Единица работы (Unit Of Work)

4.2.3.3 Загрузка по требованию (Lazy Load)

4.2.3.4 Коллекция обьектов (Identity Map)

4.2.3.5 Множество записей (Record Set)

4.2.3.6 Наследование с одной таблицей (Single Table Inheritance)

4.2.3.7 Наследование с таблицами для каждого класса (Class Table Inheritance)

4.2.3.8 Оптимистическая автономная блокировка (Optimistic Offline Lock)

4.2.3.9 Отображение с помощью внешних ключей

4.2.3.10 Отображение с помощью таблицы ассоциаций (Association Table Mapping)

4.2.3.11 Пессимистическая автономная блокировка (Pessimistic Offline Lock)

4.2.3.12 Поле идентификации (Identity Field)

4.2.3.13 Преобразователь данных (Data Mapper)

4.2.3.14 Сохранение сеанса на стороне клиента (Client Session State)

4.2.3.15 Сохранение сеанса на стороне сервера (Server Session State)

4.2.3.16 Шлюз записи данных (Row Data Gateway)

4.2.3.17 Шлюз таблицы данных (Table Data Gateway)

5 Паттерны интеграции корпоративных информационных систем

5.1 Структурные паттерны интеграции

5.1.1 Взаимодействие "точка - точка"

5.1.2 Взаимодействие "звезда" (интегрирующая среда)

5.1.3 Смешанный способ взаимодействия

5.2 Паттерны по методу интеграции

5.2.1 Интеграция систем по данным (data-centric).

5.2.2 Функционально-центрический (function-centric) подход.

5.2.3 Объектно-центрический (object-centric).

5.2.4 Интеграция на основе единой понятийной модели предметной области (concept-centric).

5.3 Паттерны интеграции по типу обмена данными

5.3.1 Файловый обмен

5.3.2 Общая база данных

5.3.3 Удаленный вызов процедур

5.3.4 Обмен сообщениями

ЛЕКЦИЯ 1. КРАТКИЕ ВЫВОДЫ

Итак, любой паттерн проектирования, представляет собой формализованное описание часто встречающейся задачи проектирования, удачное решение данной задачи, а также рекомендации по применению этого решения в различных ситуациях.

Кроме того, паттерн проектирования обязательно имеет общеупотребимое наименование.

Правильно сформулированный паттерн проектирования позволяет, отыскав однажды удачное решение, пользоваться им снова и снова.

Следует подчеркнуть, что важным начальным этапом при работе с паттернами является адекватное моделирование рассматриваемой предметной области. Это является необходимым как для получения должным образом формализованной постановки задачи, так и для выбора подходящих паттернов проектирования.

Сообразное использование паттернов проектирования дает разработчику ряд преимуществ:

– Модель системы, построенная в терминах паттернов проектирования, фактически является структурированным выделением тех элементов и связей, которые значимы при решении поставленной задачи.

– Помимо этого, модель, построенная с использованием паттернов проектирования, более проста и наглядна в изучении, чем стандартная модель. Она позволяет глубоко и всесторонне проработать архитектуру разрабатываемой системы с использованием специального языка.

– Применение паттернов проектирования повышает устойчивость системы к изменению требований и упрощает неизбежную последующую доработку системы. Кроме того, особенно велика роль использования паттернов при интеграции

Образец (шаблон) проектирования

Образец (шаблон) проектирования – повторно используемое решение типичной проблемы проектирования.

Состоит из*:

- Имени
- (Абстрактной) формулировки задачи, для решения которой применим шаблон
- (Абстрактного) решения – описание элементов дизайна, их поведения и отношений между ними
- Результаты – последствия применения образца, побочные эффекты, в т.ч. нежелательные

*Э. Гамма и др., Приемы объектно-ориентированного проектирования: шаблоны проектирования

- **Паттерны уровня классов** описывают отношения между классами и их подклассами.

Примечание. Такие отношения выражаются с помощью наследования, поэтому они **статичны**, то есть, зафиксированы на этапе компиляции.

- **Паттерны уровня объектов** описывают отношения между объектами, которые могут изменяться во время выполнения и, потому более динамичны.

- **Порождающие паттерны классов** частично делегируют ответственность за создание объектов своим подклассам.

- **Порождающие паттерны объектов** передают ответственность другому объекту.

Примечание. Почти все паттерны в какой-то мере используют наследование. Поэтому к категории «паттерны классов» отнесены только те, что сфокусированы лишь на отношениях между классами. Большинство паттернов действуют на уровне объектов.

Классификация шаблонов

- Структурные – решают задачи, связанные с отношением между классами (или другими сущностями)
- Поведенческие – решают задачи поведения классов
- Порождающие – решают задачи создания экземпляров классов и их инициализации

Рассмотрим **23 паттерна**, с которых все началось и которые сейчас должен знать каждый программист. Они разделены на три группы (для каждого паттерна приводится английское название из книги **GoF** и устоявшийся русский перевод).

«Порождающие шаблоны»

В этой группе собраны паттерны, описывающие **разные способы создания объектов.**

Прежде всего, это **«Фабричный метод»** (**Factory Method**), прием определения интерфейса создания объектов, при этом выбранный класс воплощается в подклассах.

Шаблон **«Абстрактная фабрика»** (**Abstract Factory**) определяет интерфейс для создания семейств, связанных между собой или независимых объектов, конкретные классы которых неизвестны.

С помощью шаблона **«Строитель»** (**Builder**) можно отделить процесс конструирования сложного объекта от его конкретного представления и при этом использовать один и тот же процесс для создания различных представлений.

«Прототип» (**Prototype**) описывает виды разрабатываемых объектов с помощью прототипа и создает новые путем его копирования.

Применение шаблона **«Одиночка»** (**Singleton**) гарантирует, что некоторый класс может иметь только один экземпляр (и предоставляет глобальную точку доступа к нему).

«Структурные паттерны»

В этой группе собраны паттерны, которые **позволяют менять структуру взаимодействия классов.**

«Адаптер» (**Adapter**) позволяет адаптировать интерфейс класса к конкретной ситуации.

Средствами шаблона **«Мост»** (**Bridge**) можно отделить интерфейс класса и его реализацию.

«Компоновщик» (**Composite**) объединяет объекты в древовидную структуру для представления иерархии от частного к целому. Компоновщик позволяет клиентам единообразно обращаться к отдельным объектам и группам объектов.

Паттерн **«Оформитель»** (**Decorator**, также известен как **Wrapper**, «**Оболочка**») позволяет динамически добавлять новое поведение к объекту.

«Фасад» (**Facade**) позволяет скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.

Шаблон **«Приспособленец»** (**Flyweight**) используется для облегчения работы с большим числом мелких объектов.

«Заместитель» (**Proxy**) позволяет контролировать доступ к объекту, перехватывая все вызовы к нему.

«Шаблоны поведения»

В группе «Паттерны поведения» собраны шаблоны, ответственные **за реализацию поведения объектов.**

«Цепочка ответов» (Chain of Response) позволяет пропустить запрос через цепочку объектов.

«Команда» (Command) инкапсулирует команду в объект.

«Интерпретатор» (Interpreter) позволяет создать общее декларативное решение для часто изменяющихся условий задачи.

«Итератор» (Iterator) организует последовательный доступ к коллекции.

«Посредник» (Mediator) определяет упрощенный механизм взаимодействия классов.

«Напоминание» (Memento) задает принципы, позволяющие записывать и восстанавливать внутреннее состояние объекта.

Средствами шаблона **«Наблюдатель»** (Observer) можно оповещать об изменениях множества объектов.

«Состояние» (State) — менять поведение объекта при изменении его состояния.

«Стратегия» (Strategy) инкапсулирует алгоритм внутри класса.

Паттерн **«Шаблонный метод»** (Template Method) выделяет конкретные шаги в алгоритме и опирается на подклассы для их реализации.

Средствами паттерна **«Посетитель»** (Visitor) в класс добавляются новые

№	Название паттерна	Перевод	Назначение паттерна
1	Abstract Factory	Абстрактная фабрика	Предоставляет интерфейс для создания множества связанных между собой или независимых объектов, конкретные классы которых неизвестны.
2	Adapter(синоним - Wrapper)	Адаптер (Обертка)	Преобразует существующий интерфейс класса в другой интерфейс, который понятен клиентам. При этом обеспечивает совместную работу классов, невозможную без данного паттерна из-за несовместимости интерфейсов.
3	Bridge	Мост	Отделяет абстракцию класса от его реализации, благодаря чему появляется возможность независимо изменять то и другое.
4	Builder	Строитель	Отделяет создание сложного объекта от его представления, позволяя использовать один и тот же процесс разработки для создания различных представлений.
5	Chain of Responsibility	Цепочка обязанностей	Позволяет избежать жесткой зависимости отправителя запроса от его получателя, при этом объекты-получатели связываются в цепочку, а запрос передается по цепочке, пока какой-то объект его не обработает.
6	Command	Команда	Инкапсулирует запрос в виде объекта, обеспечивая параметризацию клиентов типом запроса, установление очередности запросов, протоколирование запросов и отмену выполнения операций.
7	Composite	Компоновщик	Группирует объекты в иерархические структуры для представления отношений типа "часть-целое", что позволяет клиентам работать с единичными объектами так же, как с группами объектов.
8	Decorator	Декоратор	Применяется для расширения имеющейся функциональности и является альтернативой порождению подклассов на основе динамического назначения объектам новых операций.
9	Facade	Фасад	Предоставляет единый интерфейс к множеству операций или интерфейсов в системе на основе унифицированного интерфейса для облегчения работы с системой.
10	Factory Method	Фабричный метод	Определяет интерфейс для разработки объектов, при этом объекты данного класса могут быть созданы его подклассами.
11	Flyweight	Приспособленец	Использует принцип разделения для эффективной поддержки большого числа мелких объектов.
12	Interpreter	Интерпретатор	Для заданного языка определяет представление его грамматики на основе интерпретатора предложений языка, использующего это представление.
13	Iterator	Итератор	Дает возможность последовательно перебрать все элементы составного объекта, не раскрывая его внутреннего представления.
14	Mediator	Посредник	Определяет объект, в котором инкапсулировано знание о том, как взаимодействуют объекты из некоторого множества. Способствует уменьшению числа связей между объектами, позволяя им работать без явных ссылок друг на друга и независимо изменять схему взаимодействия.
15	Memento	Хранитель	Дает возможность получить и сохранить во внешней памяти внутреннее состояние объекта, чтобы позже объект можно было восстановить точно в таком же состоянии, не нарушая принципа инкапсуляции.
16	Observer	Наблюдатель	Специфицирует зависимость типа "один ко многим" между различными объектами, так что при изменении состояния одного объекта все зависящие от него получают извещение и автоматически обновляются.
17	Prototype	Прототип	Описывает виды создаваемых объектов с помощью прототипа, что позволяет создавать новые объекты путем копирования этого прототипа.
18	Proxy	Заместитель	Подменяет выбранный объект другим объектом для управления контроля доступа к исходному объекту.
19	Singleton	Одиночка	Для выбранного класса обеспечивает выполнение требования единственности экземпляра и предоставления к нему полного доступа.
20	State	Состояние	Позволяет выбранному объекту варьировать свое поведение при изменении внутреннего состояния. При этом создается впечатление, что изменился класс объекта.
21	Strategy	Стратегия	Определяет множество алгоритмов, инкапсулируя их все и позволяя подставлять один вместо другого. При этом можно изменять алгоритм независимо от клиента, который им пользуется.
22	Template Method	Шаблонный метод	Определяет структуру алгоритма, перераспределяя ответственность за некоторые его шаги на подклассы. При этом подклассы могут переопределять шаги алгоритма, не меняя его общей структуры.
23	Visitor	Посетитель	Позволяет определить новую операцию, не меняя описаний классов, у объектов которых она вызывается.

Основные принципы ООП: абстракция, наследование, инкапсуляция и полиморфизм

Абстракция (**abstraction**) – характеристика сущности, которая отличает ее от других сущностей. **Абстракция** **определяет** границу представления соответствующего элемента модели, применяют для определения фундаментальных понятий ООП – **класс** и **объект**.

Класс представляет собой абстракцию совокупности реальных объектов, которые имеют общий набор свойств и обладают одинаковым поведением. **Объект** в контексте ООП рассматривается как **экземпляр соответствующего класса**.

Объекты, которые не имеют идентичных свойств или не обладают одинаковым поведением, по определению, не могут быть отнесены к одному классу.

Классы можно организовать в виде иерархической структуры, которая по внешнему виду напоминает схему классификации в понятийной логике. Иерархия понятий строится следующим образом. В качестве наиболее общего понятия или категории берется понятие, имеющее **наибольший объем** и, соответственно, **наименьшее содержание**. Это самый высокий уровень абстракции для данной иерархии. Затем данное общее понятие конкретизируется, то есть уменьшается его объем и увеличивается содержание. Принцип, в соответствии с которым знание о наиболее общей категории разрешается применять для более частной категории, называется **наследованием**.

Наследование тесно связано с иерархией классов, определяющей, какие классы следует считать наиболее абстрактными и общими по отношению к другим классам. При этом если общий или родительский класс (предок) обладает фиксированным набором свойств и поведением, то производный от него класс (потомок) должен содержать этот же набор свойств и подобное поведение, а также дополнительные, которые будут характеризовать уникальность полученного класса. В этом случае говорят, что **производный класс наследует свойства и поведение родительского класса**.

Инкапсуляция характеризует сокрытие отдельных деталей внутреннего устройства классов от внешних по отношению к нему объектов или пользователей.

Конкретная реализация присущих классу свойств и методов, которые определяют его поведение, является собственным делом данного класса. Отдельные свойства и методы класса могут быть невидимы за его пределами, это относится к базовой идее введения различных категорий видимости для элементов класса.

Инкапсуляция ведет свое происхождение от деления модулей в некоторых языках программирования на две части или секции: **интерфейс** и **реализацию**.

В интерфейсной секции модуля описываются все объявления функций и процедур, а возможно и типов данных, доступных за пределами модуля.

В другой секции модуля, называемой **реализацией**, содержится программный код, который определяет конкретные способы реализации объявленных в

Полиморфизм применительно к **ООП** означает, что действия, выполняемые одноименными методами, могут различаться в зависимости от того, к какому из классов относится тот или иной метод.

Важное замечание! **Полиморфизм объектно-ориентированных языков связан с перегрузкой функций, но не тождествен ей.**

Следует иметь в виду, что имена методов и свойств тесно связаны с классами, в которых они описаны. Это обстоятельство исключает случайное применение метода для решения несвойственной ему задачи.

Наиболее существенным обстоятельством в развитии методологии **ООП** явилось осознание того, что процесс написания программного кода может быть отделен от процесса проектирования структуры программы. Прежде, чем начать программирование классов, их свойств и методов, необходимо определить сами эти классы.

Более того, нужно дать ответы на следующие вопросы:

– сколько и какие классы нужно определить для решения поставленной задачи,

– какие свойства и методы необходимы для придания классам требуемого поведения,

– а также установить взаимосвязи между классами.

Эта совокупность задач не столько связана с написанием кода, сколько с общим анализом требований к будущей программе, а также с анализом конкретной предметной области, для которой разрабатывается программа. Эти обстоятельства исследует специальная методология, получившая название **методологии**

КРИТИКА (по мнению Стива Макконнелла)

Хотя изменение кода под известный шаблон может упростить понимание кода, с применением шаблонов могут быть связаны две сложности.

1) Слепое следование некоторому выбранному шаблону может привести к усложнению программы.
2) 2) У разработчика может возникнуть желание попробовать некоторый шаблон в деле без особых оснований.

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ

Лекция 2 –
5.02.2021

Паттерны проектирования классов/объектов

- **1 Порождающие паттерны проектирования**
- 1.1 Абстрактная фабрика (Abstract Factory, Factory), др. название Инструментарий (Kit) - GoF
- 1.2 Одиночка (Singleton) - GoF
- 1.3 Прототип (Prototype) - GoF
- 1.4 Создатель экземпляров класса (Creator) - GRASP
- 1.5 Строитель (Builder) - GoF
- 1.6 (Фабричный метод) Factory Method или Виртуальный конструктор (Virtual Constructor) - GoF
- **2. Структурные паттерны проектирования классов/обьектов**
- 2.1 Адаптер (Adapter) - GoF
- 2.2 Декоратор (Decorator) или Оболочка (Wrapper) - GoF
- 2.3 Заместитель (Proxy) или Суррогат (Surrogate) - GoF
- 2.4 Информационный эксперт (Information Expert)- GRASP
- 2.5 Компоновщик (Composite) - GoF
- 2.6 Мост (Bridge), Handle (описатель) или Тело (Body) - GoF
- 2.7 Низкая связанность (Low Coupling) - GRASP
- 2.8 Приспособленец (Flyweight) - GoF
- 2.9 Устойчивый к изменениям (Protected Variations) - GRASP
- 2.10 Фасад (Facade) - GoF
- **3. Паттерны проектирования поведения классов/обьектов**
- 3.1 Интерпретатор (Interpreter) - GoF
- 3.2 Итератор (Iterator) или Курсор (Cursor) - GoF
- 3.3 Команда (Command), Действие (Action) или Транзакция (Транзакция) - GoF
- 3.4 Наблюдатель (Observer), Опубликовать - подписаться (Publish - Subscribe) или Delegation Event Model - GoF
- 3.5 Не разговаривайте с неизвестными (Don't talk to strangers) - GRASP
- 3.6 Посетитель (Visitor) - GoF
- 3.7 Посредник (Mediator) - GoF
- 3.8 Состояние (State) - GoF
- 3.9 Стратегия (Strategy) - GoF
- 3.10 Хранитель (Memento) - GoF
- 3.11 Цепочка обязанностей (Chain of Responsibility) - GoF
- 3.12 Шаблонный метод (Template Method) - GoF
- 3.13 Высокое зацепление (High Cohesion) - GRASP
- 3.14 Контроллер (Controller) - GRASP
- 3.15 Полиморфизм (Polymorphism) - GRASP
- 3.16 Искусственный (Pure Fabrication) - GRASP
- 3.17 Перенаправление (Indirection) – GRASP

- **4 Архитектурные системные паттерны**
- 4.1 Структурные паттерны
- 4.1 Репозиторий
- 4.2 Клиент/сервер
- 4.3 Обьектно - ориентированный, Модель предметной области (Domain Model), модуль таблицы (Data Mapper)
- 4.4 Многоуровневая система (Layers) или абстрактная машина
- 4.5 Потоки данных (конвейер или фильтр)
- **5 Паттерны управления**
- 5.1 Паттерны централизованного управления
- 5.1.1 Вызов - возврат (сценарий транзакции - частный случай).
- 5.1.2 Диспетчер
- 5.2 Паттерны управления, основанные на событиях
- 5.2.1 Передача сообщений
- 5.2.2 Управляемый прерываниями
- 5.2.3 Паттерны, обеспечивающие взаимодействие с базой данных
- 5.3.1 Активная запись (Active Record)
- 5.3.2 Единица работы (Unit Of Work)
- 5.3.3 Загрузка по требованию (Lazy Load)
- 5.3.4 Коллекция обьектов (Identity Map)
- 5.3.5 Множество записей (Record Set)
- 5.3.6 Наследование с одной таблицей (Single Table Inheritance)
- 5.3.7 Наследование с таблицами для каждого класса (Class Table Inheritance)
- 5.3.8 Оптимистическая автономная блокировка (Optimistic Offline Lock)
- 5.3.9 Отображение с помощью внешних ключей
- 5.3.10 Отображение с помощью таблицы ассоциаций (Association Table Mapping)
- 5.3.11 Пессимистическая автономная блокировка (Pessimistic Offline Lock)
- 5.3.12 Поле идентификации (Identity Field)
- 5.3.13 Преобразователь данных (Data Mapper)
- 5.3.14 Сохранение сеанса на стороне клиента (Client Session State)
- 5.3.15 Сохранение сеанса на стороне сервера (Server Session State)
- 5.3.16 Шлюз записи данных (Row Data Gateway)
- 5.3.17 Шлюз таблицы данных (Table Data Gateway)
- **6 Паттерны интеграции корпоративных информационных систем**
- 6.1 Структурные паттерны интеграции
- 6.1.1 Взаимодействие "точка - точка"
- 6.1.2 Взаимодействие "звезда" (интегрирующая среда)
- 6.1.3 Смешанный способ взаимодействия
- 6.2 Паттерны по методу интеграции
- 6.2.1 Интеграция систем по данным (data-centric).
- 6.2.2 Функционально-центрический (function-centric) подход.
- 6.2.3 Объектно-центрический (object-centric).
- 6.2.4 Интеграция на основе единой понятийной модели предметной области (concept-centric).
- 6.3 Паттерны интеграции по типу обмена данными
- 6.3.1 Файловый обмен
- 6.3.2 Общая база данных
- 6.3.3 Удаленный вызов процедур
- 6.3.4 Обмен сообщениями

Образец (шаблон) проектирования

Образец (шаблон) проектирования – повторно используемое решение типичной проблемы проектирования.

Состоит из*:

- Имени
- (Абстрактной) формулировки задачи, для решения которой применим шаблон
- (Абстрактного) решения – описание элементов дизайна, их поведения и отношений между ними
- Результаты – последствия применения образца, побочные эффекты, в т.ч. нежелательные

*Э. Гамма и др., Приемы объектно-ориентированного проектирования: шаблоны проектирования

Классификация шаблонов

- Структурные – решают задачи, связанные с отношением между классами (или другими сущностями)
- Поведенческие – решают задачи поведения классов
- Порождающие – решают задачи создания экземпляров классов и их инициализации

- **Паттерны уровня классов** описывают отношения между классами и их подклассами.

Примечание. Такие отношения выражаются с помощью наследования, поэтому они **статичны**, то есть, зафиксированы на этапе компиляции.

- **Паттерны уровня объектов** описывают отношения между объектами, которые могут изменяться во время выполнения и, потому более динамичны.

- **Порождающие паттерны классов** частично делегируют ответственность за создание объектов своим подклассам.

- **Порождающие паттерны объектов** передают ответственность другому объекту.

Примечание. Почти все паттерны в какой-то мере используют наследование. Поэтому к категории «паттерны классов» отнесены только те, что сфокусированы лишь на отношениях между классами. Большинство паттернов действуют на уровне объектов.

Порождающие шаблоны

- Одиночка
- Абстрактная фабрика
- Прототип
- Строитель
- Фабричный метод

Уровень объектов

Уровень классов

Порождающие шаблоны

- Паттерны, описывающие разные способы создания объектов

- **«Фабричный метод» (Factory Method)** - прием определения интерфейса создания объектов, при этом выбранный класс воплощается в подклассах.
- **«Абстрактная фабрика» (Abstract Factory)** - определяет интерфейс для создания семейств, связанных между собой или независимых объектов, конкретные классы которых неизвестны.
- **«Строитель» (Builder)** - можно отделить процесс конструирования сложного объекта от его конкретного представления и при этом использовать один и тот же процесс для создания различных представлений.
- **«Прототип» (Prototype)** - описывает виды разрабатываемых объектов с помощью прототипа и создает новые путем его копирования.
- **«Одиночка» (Singleton)** - гарантирует, что некоторый класс может иметь только один экземпляр (и предоставляет глобальную точку доступа к нему).

- **Другие способы классификации паттернов**
- Некоторые паттерны часто используются вместе. Например, **компоновщик** применяется с **итератором** или **посетителем**.
- Некоторыми паттернами предлагаются альтернативные решения. Так, **прототип** нередко можно использовать вместо **абстрактной фабрики**.
- Применение части паттернов приводит к схожему дизайну, хотя изначально их назначение различно. Например, структурные диаграммы компоновщика и декоратора похожи.
- Классифицировать паттерны можно и по их ссылкам, которые на рис. 1 изображены графически.

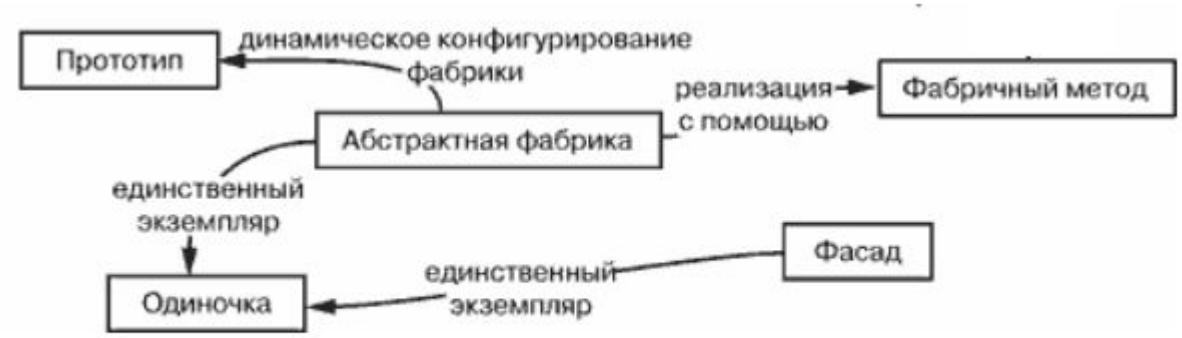


Рис. 1. Отношения между паттернами проектирования

Фабричный метод (Factory method)

Назначение паттерна Factory Method

Часто требуется создавать объекты самых разных типов.

Пример задач:

Система должна оставаться **расширяемой** путем добавления объектов новых типов. Непосредственное использование выражения **new** является нежелательным, так как в этом случае код создания объектов с указанием конкретных типов может получиться разбросанным по всему приложению. Тогда такие операции как добавление в систему объектов новых типов или замена объектов одного типа на другой будут затруднительными.

Паттерн **Factory Method** позволяет системе оставаться **независимой** как от самого процесса порождения объектов, так и от их типов.

Заранее известно, когда нужно создавать объект, но неизвестен его тип.

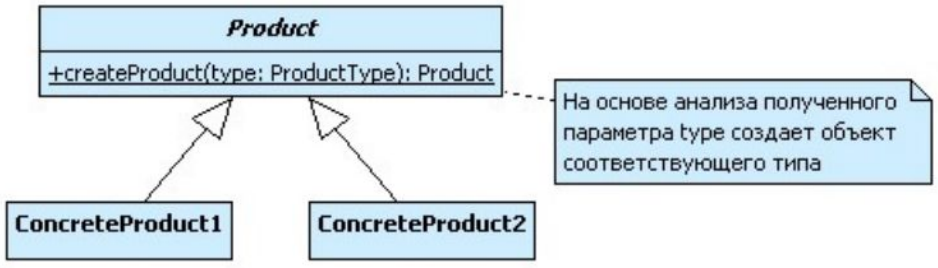
Описание паттерна Factory Method

- Для того, чтобы система оставалась независимой от различных типов объектов, паттерн **Factory Method** использует **механизм полиморфизма** – классы всех конечных типов наследуют от одного абстрактного базового класса, предназначенного для полиморфного использования.
- В этом базовом классе определяется единый интерфейс, через который пользователь будет оперировать объектами конечных типов.
- Для обеспечения относительно простого добавления в систему новых типов паттерн **Factory Method** локализует создание объектов конкретных типов в **специальном классе-фабрике**.
- **Методы** этого класса, посредством которых создаются объекты конкретных классов, называются **фабричными**.

Существуют две разновидности паттерна Factory Method:

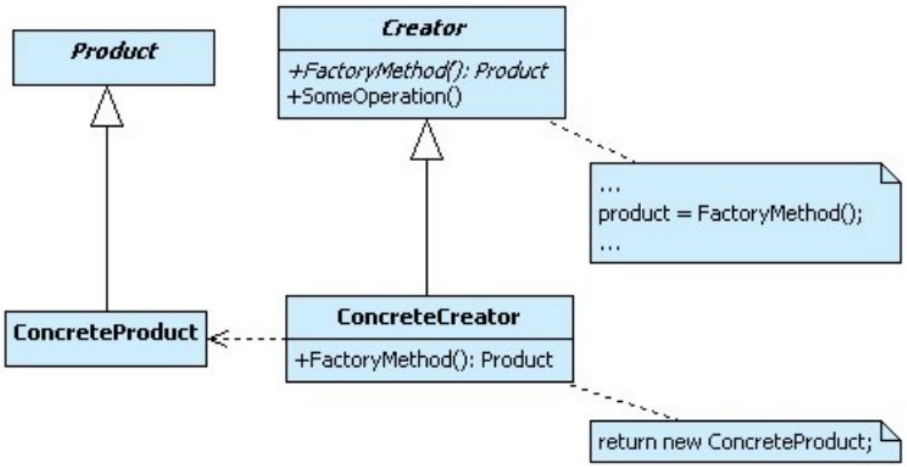
I. Обобщенный конструктор, когда в том же самом полиморфном базовом классе, от которого наследуют производные классы всех создаваемых в системе типов, определяется статический фабричный метод. В качестве параметра в этот метод должен передаваться идентификатор типа создаваемого объекта.

UML-диаграмма классов паттерна Factory Method. Обобщенный конструктор



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

UML-диаграмма классов паттерна Factory Method. Классическая реализация



Результаты применения паттерна Factory Method

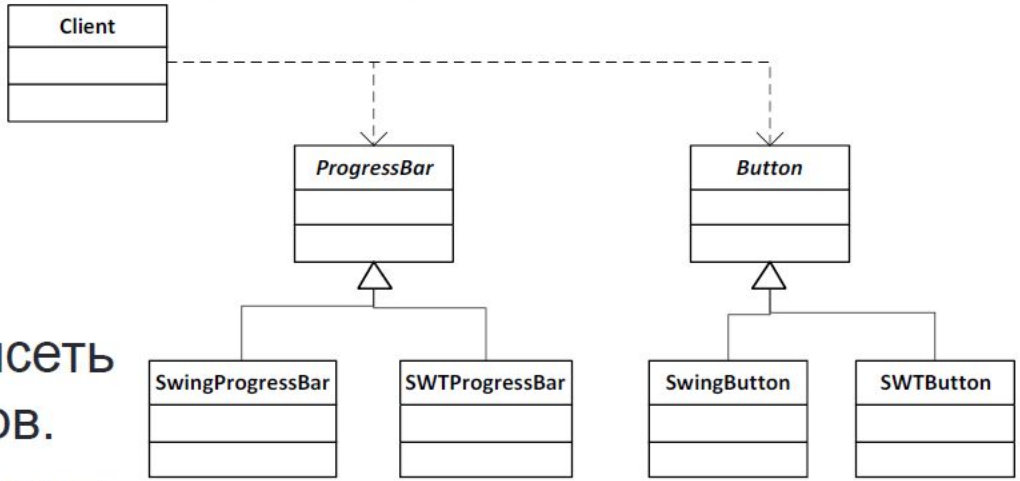
Достоинства паттерна Factory Method

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

Недостатки паттерна Factory Method

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

Abstract Factory: пример задачи

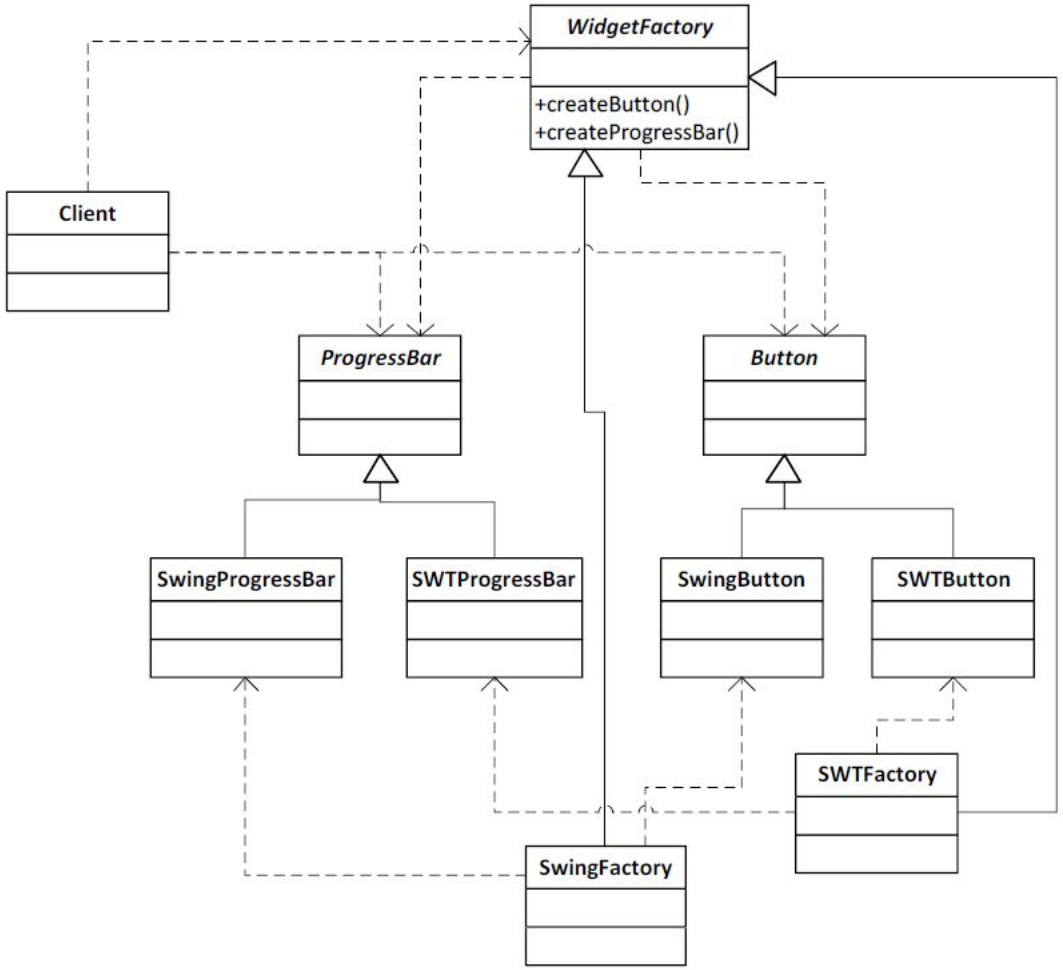


Клиент не должен зависеть от реализации виджетов.

Клиент может использовать их через базовые платформенно-независимые классы.

Как дать клиенту возможность инстанцировать эти классы?

Abstract Factory: пример решения



Назначение паттерна Abstract Factory

- **Используйте паттерн Abstract Factory (абстрактная фабрика) если:**
- - Система должна оставаться **независимой** как от процесса создания новых объектов, так и от типов порождаемых объектов. Непосредственное использование выражения **new** в коде приложения нежелательно.
- - Необходимо создавать группы или семейства взаимосвязанных объектов, исключая возможность одновременного использования объектов из разных семейств в одном контексте.
- **Примеры групп взаимосвязанных объектов**
- **1.** Пусть некоторое приложение с поддержкой графического интерфейса пользователя рассчитано на использование на различных платформах, при этом внешний вид этого интерфейса должен соответствовать принятому стилю для той или иной платформы.
- Например, если это приложение установлено на Windows-платформу, то его кнопки, меню, полосы прокрутки должны отображаться в стиле, принятом для Windows. Группой взаимосвязанных объектов в этом случае будут элементы графического интерфейса пользователя для конкретной платформы.
- **2.** Рассмотрим текстовый редактор с многоязычной поддержкой, у которого имеются функциональные модули, отвечающие за расстановку переносов слов и проверку орфографии. Если, открыт документ на русском языке, то должны быть подключены соответствующие модули, учитывающие специфику русского языка. Ситуация, когда для такого документа одновременно используются модуль расстановки переносов для русского языка и модуль проверки орфографии для немецкого языка, исключается.

  Здесь группой взаимосвязанных объектов будут соответствующие модули,

Описание паттерна Abstract Factory

- Паттерн **Abstract Factory** реализуется на основе фабричных методов (см. паттерн **Factory Method**).

- Любое семейство или группа взаимосвязанных объектов характеризуется несколькими общими типами создаваемых продуктов, при этом сами продукты таких типов будут различными для разных семейств.

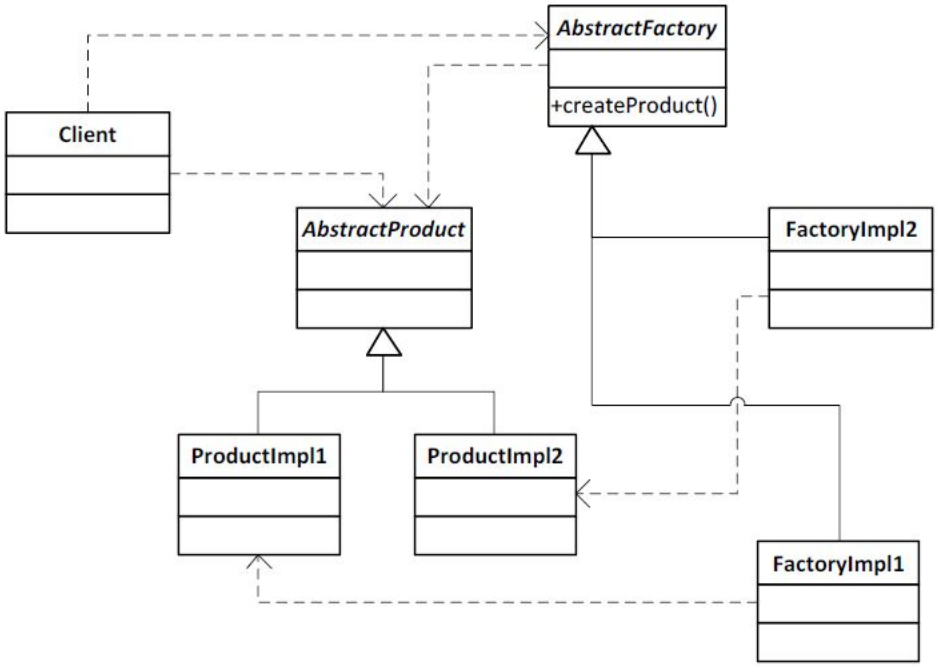
- Для того чтобы система оставалась независимой от специфики того или иного семейства продуктов необходимо использовать общие интерфейсы для всех основных типов продуктов.

- Для решения задачи по созданию семейств взаимосвязанных объектов паттерн **Abstract Factory** вводит понятие абстрактной фабрики.
- Абстрактная фабрика представляет собой некоторый **полиморфный базовый класс**, назначением которого является **объявление интерфейсов фабричных методов**, служащих для создания продуктов всех основных типов (один фабричный метод на каждый тип продукта). Производные от него классы, реализующие эти интерфейсы, предназначены для создания продуктов всех типов внутри семейства или группы.

Abstract Factory: шаблон

Задача: обеспечить «полиморфный» конструктор

Решение: конструирование выполняется с помощью объекта дополнительного полиморфного класса

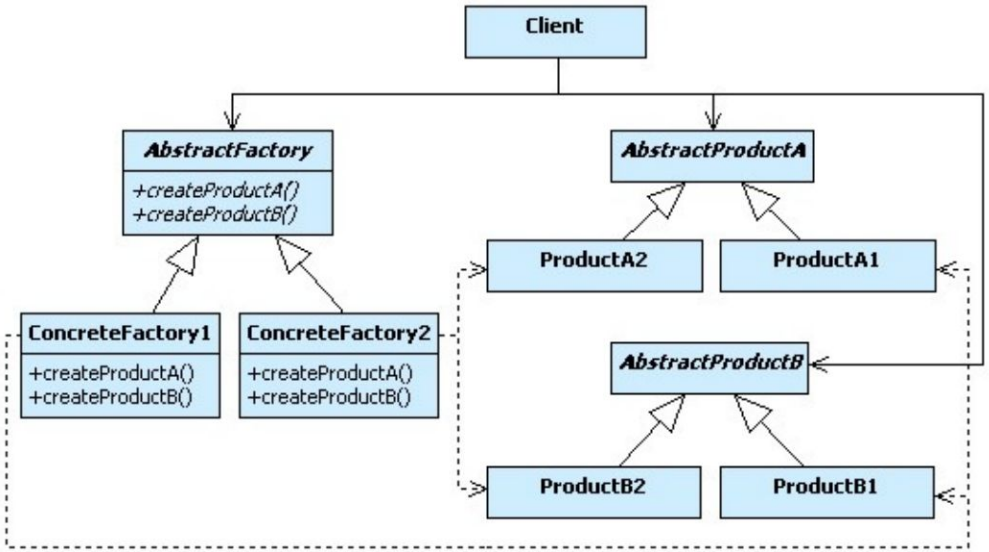


Последствия:

Фабрика имеет право выбросить исключение (в отличие от конструктора)

Проблема инстанцирования фабрики (кто побреет брадобрея?)

UML-диаграмма классов паттерна Abstract Factory



Результаты применения паттерна Abstract Factory

- **Достоинства паттерна Abstract Factory**
- Скрывает сам процесс порождения объектов, а также делает систему независимой от типов создаваемых объектов, специфичных для различных семейств или групп (пользователи оперируют этими объектами через соответствующие абстрактные интерфейсы).
- Позволяет быстро настраивать систему на нужное семейство создаваемых объектов. В случае многоплатформенного графического приложения для перехода на новую платформу, то есть для замены графических элементов (кнопок, меню, полос прокрутки) одного стиля другим достаточно создать нужный подкласс абстрактной фабрики. При этом условие невозможности одновременного использования элементов разных стилей для некоторой платформы будет выполнено автоматически.
- **Недостатки паттерна Abstract Factory**
- Трудно добавлять новые типы создаваемых продуктов или заменять существующие, так как интерфейс базового класса абстрактной фабрики фиксирован. Например, если нужно будет ввести новый вид объекта, то надо будет добавить новый фабричный метод, объявив его интерфейс в полиморфном базовом классе AbstractFactory и реализовав во всех подклассах. Снять это ограничение можно следующим образом. Все создаваемые объекты должны наследовать от общего абстрактного базового класса, а в единственный фабричный метод в качестве параметра необходимо передавать идентификатор типа объекта, который нужно создать.
- Однако в этом случае необходимо учитывать следующий момент. Фабричный метод создает объект запрошенного подкласса, но при этом возвращает его с интерфейсом общего абстрактного класса в виде ссылки или указателя, поэтому для такого объекта будет затруднительно выполнить какую-либо операцию, специфичную для подкласса.

Singleton

Задача: обеспечить создание ровно одного экземпляра класса.

Решение:

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

- **Назначение паттерна Singleton**
- Часто в системе могут существовать сущности только в единственном экземпляре,

Пример. Система ведения системного журнала сообщений или драйвер дисплея.

В таких случаях необходимо уметь создавать единственный экземпляр некоторого типа, предоставлять к нему доступ извне, запрещать создание нескольких экземпляров того же типа. Паттерн Singleton предоставляет такие возможности.

- **Описание паттерна Singleton**

Архитектура паттерна Singleton основана на идее использования глобальной переменной, имеющей следующие важные свойства:

– Такая переменная доступна всегда.

– Время жизни глобальной переменной - от запуска программы до ее завершения.

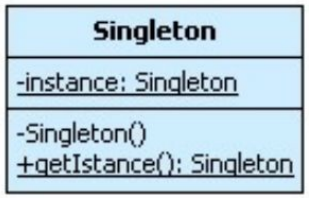
– Предоставляет глобальный доступ, то есть, такая переменная может быть доступна из любой части программы.

Использовать глобальную переменную некоторого типа непосредственно невозможно, так как существует проблема обеспечения единственности экземпляра, а именно, возможно создание нескольких переменных того же самого типа (например, стековых).

Для решения этой проблемы паттерн **Singleton** возлагает контроль над созданием единственного объекта на сам класс. Доступ к этому объекту осуществляется через статическую функцию-член класса, которая возвращает указатель или ссылку на него. Этот объект будет создан только при первом обращении к методу, а все последующие вызовы просто возвращают его адрес.

Для обеспечения уникальности объекта, конструкторы и оператор присваивания

UML-диаграмма классов паттерна Singleton



> Паттерн Singleton часто называют усовершенствованной глобальной переменной.

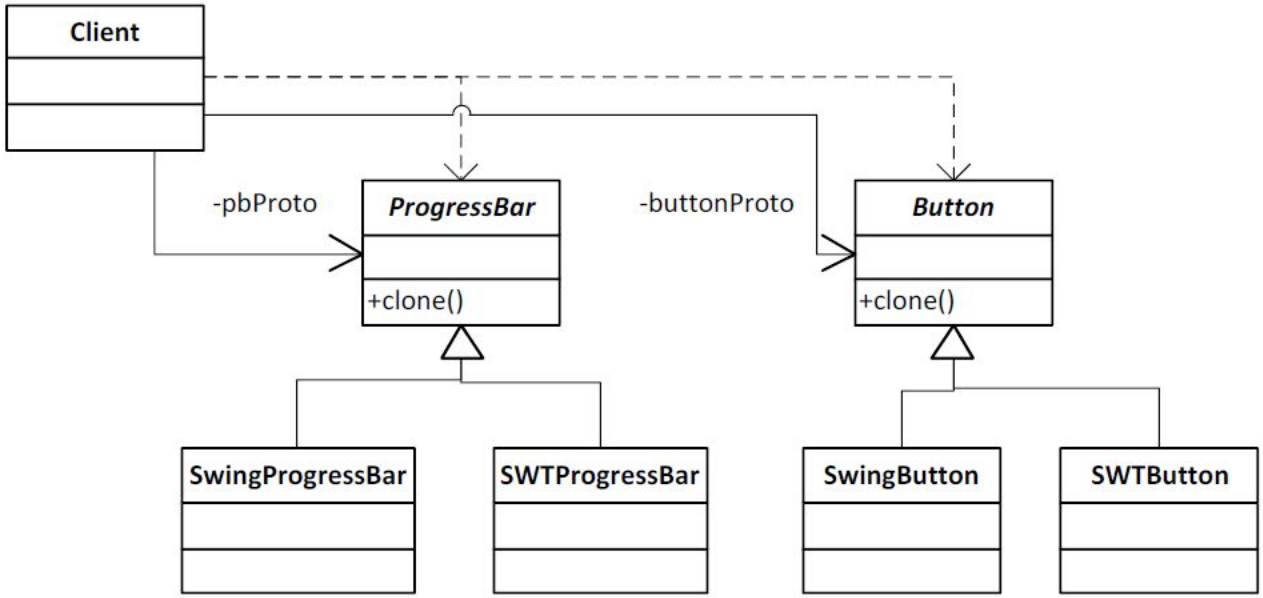
- **Результаты применения паттерна Singleton**

- **Достоинства паттерна Singleton**
- Класс сам контролирует процесс создания единственного экземпляра.
- Паттерн легко адаптировать для создания нужного числа экземпляров.
- Возможность создания объектов классов, производных от Singleton.
- **Недостатки паттерна Singleton**
- В случае использования нескольких взаимозависимых одиночек их реализация может резко усложниться.

Prototype

Пример задачи: см. «Abstract Factory»

Решение: объекты создаются как клоны объектов-прототипов



Prototype: анализ

Преимущества:

- простой дизайн
- прототип обладает свойствами класса, его можно применять для создания динамических объектных моделей в статических языках

Недостатки:

- лишние, «мусорные» объекты

В JavaScript прототипы полностью заменяют классы.

Назначение паттерна Prototype

Паттерн Prototype (прототип) можно использовать в следующих случаях:

- Система должна оставаться независимой как от процесса создания новых объектов, так и от типов порождаемых объектов. Непосредственное использование выражения new в коде приложения считается нежелательным (подробнее об этом в разделе Порождающие паттерны).
- Необходимо создавать объекты, точные классы которых становятся известными уже на стадии выполнения программы.
- Паттерн **Factory Method** также делает систему независимой от типов порождаемых объектов, но для этого он вводит параллельную иерархию классов: для каждого типа создаваемого объекта должен присутствовать соответствующий класс-фабрика, что может быть нежелательно. Паттерн **Prototype** лишен этого недостатка.

Описание паттерна Prototype

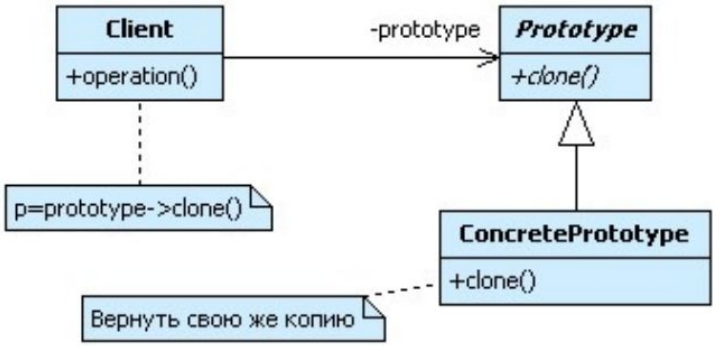
- Для создания новых объектов паттерн **Prototype** использует прототипы.

Прототип - это уже существующий в системе объект, который поддерживает операцию клонирования, то есть умеет создавать копию самого себя. Таким образом, для создания объекта некоторого класса достаточно выполнить операцию **clone()** соответствующего прототипа.

- Паттерн **Prototype** реализует подобное поведение следующим образом:

- все классы, объекты которых нужно создавать, должны быть подклассами одного общего абстрактного базового класса.
- Этот базовый класс должен объявлять интерфейс метода **clone().**
- Также здесь могут объявляться виртуальными и другие общие методы, например, **initialize()** в случае, если после клонирования нужна инициализация вновь созданного объекта.
- Все производные классы должны реализовывать метод **clone()**. В языке C++ для создания копий объектов используется **конструктор копирования**, однако, в общем случае, создание объектов при помощи операции копирования не является обязательным.

UML-диаграмма классов паттерна Prototype



- **Результаты применения паттерна Prototype**

- **Достоинства паттерна Prototype**
- Для создания новых объектов клиенту необязательно знать их конкретные классы.
- Возможность гибкого управления процессом создания новых объектов за счет возможности динамических добавления и удаления прототипов в реестр.

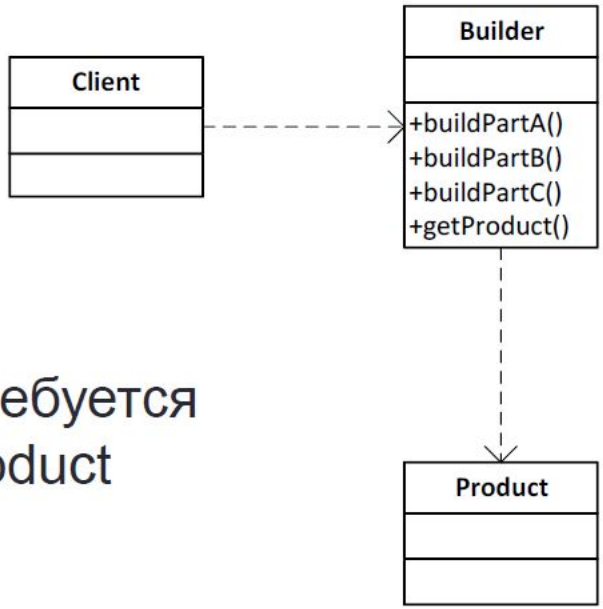
- **Недостатки паттерна Prototype**
- Каждый тип создаваемого продукта должен реализовывать операцию клонирования **clone().** В случае, если требуется глубокое копирование объекта (объект содержит ссылки или указатели на другие объекты), это может быть непростой задачей.

Builder: задача

Необходимо сконструировать
сложный объект (например HTML-документ),
что невозможно сделать атомарным вызовом.

При этом необходимо, чтобы при получении доступа
к объекту он был консистентен.

Builder: шаблон

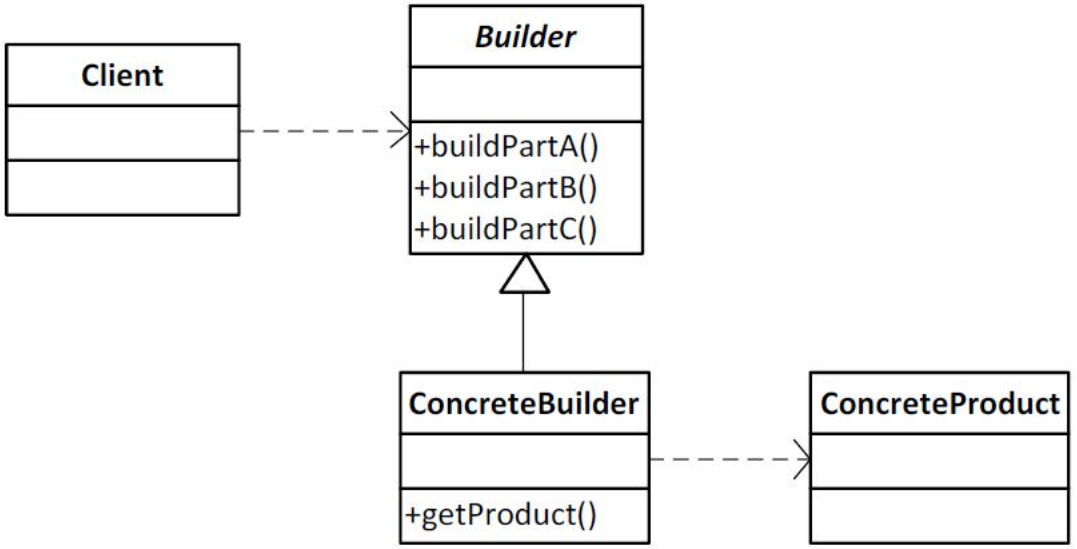


Решение:

Клиент использует build*-методы для инициализации объекта. Далее, объект востребуется атомарно посредством getProduct

Результаты:

Builder (GOF-вариант)



Задача:

Сконструировать сложный объект,
причем алгоритм конструирования
не должен зависеть от того, какой конкретно объект
будет получен в результате

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ

Лекция 3

Образец (шаблон) проектирования

Образец (шаблон) проектирования – повторно используемое решение типичной проблемы проектирования.

Состоит из*:

- Имени
- (Абстрактной) формулировки задачи, для решения которой применим шаблон
- (Абстрактного) решения – описание элементов дизайна, их поведения и отношений между ними
- Результаты – последствия применения образца, побочные эффекты, в т.ч. нежелательные

*Э. Гамма и др., Приемы объектно-ориентированного проектирования: шаблоны проектирования

Классификация шаблонов

- Структурные – решают задачи, связанные с отношением между классами (или другими сущностями)
- Поведенческие – решают задачи поведения классов
- Порождающие – решают задачи создания экземпляров классов и их инициализации

Порождающие шаблоны

- Одиночка
- Абстрактная фабрика
- Прототип
- Строитель
- Фабричный метод

Уровень объектов

Уровень классов

Порождающие шаблоны

- Паттерны, описывающие разные способы создания объектов

- **«Фабричный метод» (Factory Method)** - прием определения интерфейса создания объектов, при этом выбранный класс воплощается в подклассах.
- **«Абстрактная фабрика» (Abstract Factory)** - определяет интерфейс для создания семейств, связанных между собой или независимых объектов, конкретные классы которых неизвестны.
- **«Строитель» (Builder)** - можно отделить процесс конструирования сложного объекта от его конкретного представления и при этом использовать один и тот же процесс для создания различных представлений.
- **«Прототип» (Prototype)** - описывает виды разрабатываемых объектов с помощью прототипа и создает новые путем его копирования.
- **«Одиночка» (Singleton)** - гарантирует, что некоторый класс может иметь только один экземпляр (и предоставляет глобальную точку доступа к нему).

Резюме лекции 2

- **Паттерн Factory Method**

развивает тему фабрики объектов дальше, перенося создание объектов в специально предназначенные для этого классы. В его классическом варианте вводится полиморфный класс **Factory**, в котором определяется интерфейс фабричного метода, подобного **creatorObject ( )**, а ответственность за создание объектов конкретных классов переносится на производные от Factory классы, в которых этот метод переопределяется.

Эта функция получает в качестве аргумента тип объекта, который нужно создать, создает его и возвращает соответствующий указатель на базовый класс.

- **Паттерн Abstract Factory**

использует несколько фабричных методов и предназначен для создания целого семейства или группы взаимосвязанных объектов.

Паттерн Prototype

создает новые объекты с помощью прототипов. **Прототип** – некоторый объект, умеющий создавать по запросу копию самого себя.

- **Паттерн Singleton**

контролирует создание единственного экземпляра некоторого класса и предоставляет доступ к нему.

- **Паттерн Builder**

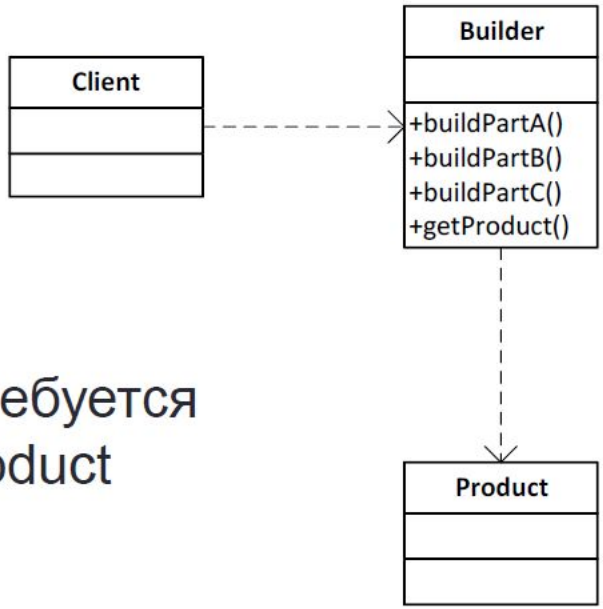
определяет процесс поэтапного конструирования сложного объекта, в результате которого могут получаться разные представления этого объекта.

Builder: задача

Необходимо сконструировать
сложный объект (например HTML-документ),
что невозможно сделать атомарным вызовом.

При этом необходимо, чтобы при получении доступа
к объекту он был консистентен.

Builder: шаблон

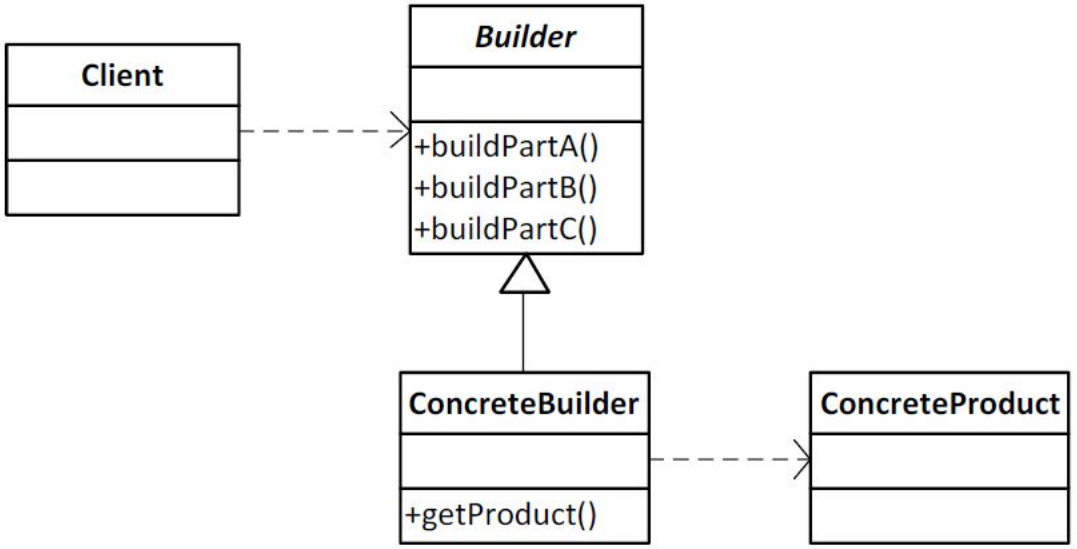


Решение:

Клиент использует build*-методы для инициализации объекта. Далее, объект востребуется атомарно посредством getProduct

Результаты:

Builder (GOF-вариант)



Задача:

Сконструировать сложный объект,
причем алгоритм конструирования
не должен зависеть от того, какой конкретно объект
будет получен в результате

Назначение паттерна Builder

- **Паттерн Builder может помочь в решении следующих задач:**

1. В системе могут существовать сложные объекты, создание которых за одну операцию затруднительно или невозможно. Требуется поэтапное построение объектов с контролем результатов выполнения каждого этапа.

2. Данные должны иметь несколько представлений.

Классический пример.

Пусть есть некоторый исходный документ в формате RTF (Rich Text Format), в общем случае содержащий текст, графические изображения и служебную информацию о форматировании (размер и тип шрифтов, отступы и др.). Если этот документ в формате RTF преобразовать в другие форматы (например, Microsoft Word или простой ASCII-текст), то полученные документы и будут представлениями исходных данных.

Описание паттерна Builder

- **Паттерн Builder** отделяет алгоритм поэтапного конструирования сложного продукта (объекта) от его внешнего представления так, что с помощью одного и того же алгоритма можно получать разные представления этого продукта.

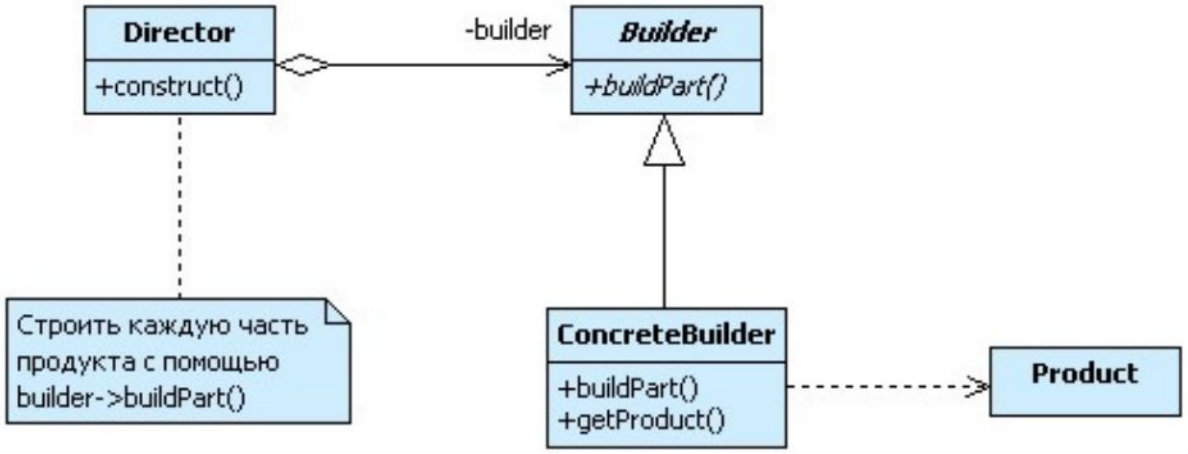
Поэтапное создание продукта означает его построение по частям. После того как построена последняя часть, продукт можно использовать.

- Для этого паттерн **Builder** определяет алгоритм поэтапного создания продукта в специальном классе **Director** (распорядитель).
- Ответственность за координацию процесса сборки отдельных частей продукта возлагает на иерархию классов **Builder**.
- В этой иерархии базовый класс **Builder** объявляет интерфейсы для построения отдельных частей продукта, а соответствующие им подклассы **ConcreteBuilder** реализуют подходящим образом.

Например

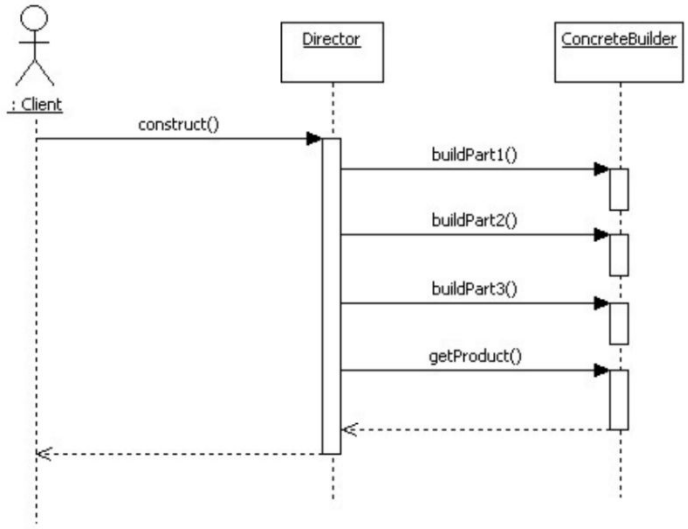
- создают или получают нужные ресурсы
- сохраняют промежуточные результаты

UML-диаграмма классов паттерна Builder



- Класс **Director** содержит указатель или ссылку на **Builder**, который перед началом работы должен быть сконфигурирован экземпляром **ConcreteBuilder**, определяющим соответствующе представление.
- После этого **Director** может обрабатывать клиентские запросы на создание объекта. Получив такой запрос, с помощью имеющегося экземпляра строителя **Director** строит продукт по частям, а затем возвращает его пользователю.

UML-диаграмма последовательности паттерна Builder



Для получения разных представлений некоторых данных с помощью паттерна **Builder** распорядитель **Director** должен использовать соответствующие экземпляры **ConcreteBuilder**.

Использование в задаче преобразования RTF-документов в документы различных форматов.

Для ее решения класс **Builder** объявляет интерфейсы для преобразования отдельных частей исходного документа, таких как текст, графика и управляющая информация о форматировании, а производные классы **WordBuilder**, **AsciiBuilder** и другие их реализуют с учетом особенностей того или иного формата. Так, например, конвертор **AsciiBuilder** должен учитывать тот факт, что простой текст не может содержать изображений и управляющей информации о форматировании, поэтому соответствующие методы будут пустыми.

По запросу клиента распорядитель **Director** будет последовательно вычитывать данные из RTF-документа и передавать их в выбранный ранее конвертор, например, **AsciiBuilder**. После того как все данные прочитаны, полученный новый документ в виде ASCII-теста можно вернуть клиенту. Следует отметить, для того чтобы заменить формат исходных данных (здесь RTF) на другой, достаточно использовать другой класс распорядителя.

Результаты применения паттерна Builder

- **Замечание.** Интересно сравнить паттерн **Builder** с паттерном **Abstract Factory**, который также может использоваться для создания сложных продуктов.
- Паттерн **Abstract Factory** акцентирует внимание на создании семейств некоторых объектов.
- Паттерн **Builder** подчеркивает поэтапное построение продукта. При этом класс **Builder** скрывает все подробности построения сложного продукта так, что **Director** ничего не знает о его составных частях.
- **Достоинства паттерна Builder**
- Возможность контролировать процесс создания сложного продукта.
- Возможность получения разных представлений некоторых данных.
- **Недостатки паттерна Builder**
- **ConcreteBuilder** и создаваемый им продукт жестко связаны между собой, поэтому при внесеннии изменений в класс продукта скорее всего придется соотвествующим образом изменять и класс **ConcreteBuilder**.

Сквозной пример реализации и применения порождающих паттенов

Разработка стратегии "Пунические войны" (Рим и Карфаген (264 — 146 г. до н. э.).

- Персонажи игры могут быть воины трех типов: пехота, конница и лучники.

Казалось бы, для этого достаточно использовать следующую иерархию классов.
Полиморфный базовый класс Warrior определяет общий интерфейс,
а производные от него классы Infantryman, Archer и Horseman реализуют особенности каждого вида воина.
Сложность заключается в том, что хотя код системы и оперирует готовыми объектами через соответствующие
общие интерфейсы, в процессе игры требуется создавать новые персонажи, указывая их конкретные типы.
Если код их создания рассредоточен по всему приложению, то добавлять новые типы персонажей или заменять
существующие будет затруднительно. В таких случаях на помощь приходит **фабрика объектов**,
локализующая создание объектов.
Работа фабрики объектов напоминает функционирование виртуального конструктора. Можно создавать объекты нужных классов, не указывая напр
Пример справа демонстрирует простейший

```
class Warrior
{
  public:
     virtual void info() = 0;
     virtual ~Warrior() {}
};

class Infantryman: public Warrior
{
  public:
      void info() { cout << "Infantryman" << endl;
};

class Archer: public Warrior
{
  public:
     void info() { cout << "Archer" << endl; }
};

class Horseman: public Warrior
{
  public:
     void info() { cout << "Horseman" << endl; }
};
```

```
enum Warrior_ID { Infantryman_ID=0, Archer_ID, Horseman_ID };

Warrior * createWarrior( Warrior_ID id  )
{
     Warrior * p;
     switch (id)
     {
          case Infantryman_ID:
               p = new Infantryman();
               break;
          case Archer_ID:
               p = new Archer();
               break;
          case Horseman_ID:
               p = new Horseman();
               break;
          default:
               assert( false);
     }
     return p;
}
```

Сквозной пример реализации и применения порождающих паттенов -2

- Теперь, скрывая детали, код создания объектов разных типов игровых персонажей сосредоточен в одном месте, а именно, в фабричной функции createWarrior( ). Эта функция получает в качестве аргумента тип объекта, который нужно создать, создает его и возвращает соответствующий указатель на базовый класс.
- Несмотря на очевидные преимущества, у этого варианта фабрики также существуют недостатки. Например, для добавления нового вида боевой единицы необходимо сделать несколько шагов – завести новый идентификатор типа и модифицировать код фабричной функции createWarrior( ).
- Ниже представленный вариант паттерна Factory Method пользуется популярностью благодаря своей простоте. В нем статический фабричный метод createWarrior() определен непосредственно в полиморфном базовом классе Warrior. Этот фабричный метод является параметризированным, то

Реализация паттерна Factory Method на основе обобщенного конструктора

```
// #include <iostream>
#include <vector>

enum Warrior_ID { Infantryman_ID=0, Archer_ID, Horseman_ID };

// Иерархия классов игровых персонажей
class Warrior
{
  public:
    virtual void info() = 0;
    virtual ~Warrior() {}
    // Параметризированный статический фабричный метод
    static Warrior* createWarrior( Warrior_ID id );
};

class Infantryman: public Warrior
{
  public:
    void info() {
      cout << "Infantryman" << endl;
    }
};

class Archer: public Warrior
{
  public:
    void info() {
      cout << "Archer" << endl;
    }
};

class Horseman: public Warrior
{
  public:
    void info() {
      cout << "Horseman" << endl;
    }
};


// Реализация параметризированного фабричного метода
Warrior* Warrior::createWarrior( Warrior_ID id )
{
    Warrior * p;
    switch (id)
    {
        case Infantryman_ID:
            p = new Infantryman();
            break;
        case Archer_ID:
            p = new Archer();
            break;
        case Horseman_ID:
            p = new Horseman();
            break;
        default:
            assert( false);
    }
    return p;
};
```

Сквозной пример реализации и применения порождающих паттенов -3

```
61
62
63  // Создание объектов при помощи параметризированного фабричного метода
64  int main()
65  {
66      vector<Warrior*> v;
67      v.push_back( Warrior::createWarrior( Infantryman_ID));
68      v.push_back( Warrior::createWarrior( Archer_ID));
69      v.push_back( Warrior::createWarrior( Horseman_ID));
70
71      for(int i=0; i<v.size(); i++)
72          v[i]->info();
73      // ...
74  }
```

С точки зрения "чистоты" объектно-ориентированного кода у этого варианта есть следующие недостатки:

Так как код по созданию объектов всех возможных типов сосредоточен в статическом фабричном методе

класса Warrior, то базовый класс Warrior обладает знанием обо всех производных от него классах, что является нетипичным для объектно-ориентированного подхода.

Подобное использование оператора switch (как в коде фабричного метода createWarrior()) в объектно-ориентированном программировании также не

Классическая реализация паттерна Factory Me

```
1   //
2   #include <iostream>
3   #include <vector>
4
5   // Иерархия классов игровых персонажей
6   class Warrior
7   {
8     public:
9       virtual void info() = 0;
10      virtual ~Warrior() {}
11  };
12
13  class Infantryman: public Warrior
14  {
15    public:
16      void info() {
17        cout << "Infantryman" << endl;
18      };
19  };
20
21  class Archer: public Warrior
22  {
23    public:
24      void info() {
25        cout << "Archer" << endl;
26      };
27  };
28
```

```
29  class Horseman: public Warrior
30  {
31    public:
32      void info() {
33        cout << "Horseman" << endl;
34      };
35  };
36
37
38  // Фабрики объектов
39  class Factory
40  {
41    public:
42      virtual Warrior* createWarrior() = 0;
43      virtual ~Factory() {}
44  };
45
46  class InfantryFactory: public Factory
47  {
48    public:
49      Warrior* createWarrior() {
50        return new Infantryman;
51      }
52  };
53
```

Сквозной пример реализации и применения порождающих паттенов - 4

```
54  class ArchersFactory: public Factory
55  {
56    public:
57      Warrior* createWarrior() {
58        return new Archer;
59      }
60  };
61
62  class CavalryFactory: public Factory
63  {
64    public:
65      Warrior* createWarrior() {
66        return new Horseman;
67      }
68  };
69
70
71  // Создание объектов при помощи фабрик объектов
72  int main()
73  {
74      InfantryFactory* infantry_factory = new InfantryFactory;
75      ArchersFactory*  archers_factory  = new ArchersFactory ;
76      CavalryFactory*  cavalry_factory  = new CavalryFactory ;
77
78      vector<Warrior*> v;
79      v.push_back( infantry_factory->createWarrior());
80      v.push_back( archers_factory->createWarrior());
81      v.push_back( cavalry_factory->createWarrior());
82
83      for(int i=0; i<v.size(); i++)
84          v[i]->info();
85      // ...
86  }
```

- Классический вариант паттерна **Factory Method** использует идею полиморфной фабрики. Специально выделенный для создания объектов полиморфный базовый класс **Factory** объявляет интерфейс фабричного метода **createWarrior()**, а производные классы его реализуют.
- Представленный вариант паттерна **Factory Method** является наиболее распространенным, но не единственным. Возможны следующие вариации:
- Класс **Factory** имеет реализацию фабричного метода **createWarrior()** по умолчанию.
- Фабричный метод **createWarrior()** класса Factory параметризирован типом создаваемого объекта (как и у представленного ранее, простого варианта **Factory Method**) и имеет реализацию по умолчанию. В этом случае, производные от **Factory** классы необходимы лишь для того, чтобы определить нестандартное поведение **createWarrior()**.

Сквозной пример реализации и применения порождающих паттенов - 5

Реализация паттерна Abstract Factory

```
#include <iostream>
#include <vector>

// Абстрактные базовые классы всех возможных видов воинов
class Infantryman
{
  public:
    virtual void info() = 0;
    virtual ~Infantryman() {}
};

class Archer
{
  public:
    virtual void info() = 0;
    virtual ~Archer() {}
};

class Horseman
{
  public:
    virtual void info() = 0;
    virtual ~Horseman() {}
};


// Классы всех видов воинов Римской армии
class RomanInfantryman: public Infantryman
{
  public:
    void info() {
      cout << "RomanInfantryman" << endl;
    }
};

class RomanArcher: public Archer
{
  public:
    void info() {
      cout << "RomanArcher" << endl;
    }
};

class RomanHorseman: public Horseman
{
  public:
    void info() {
      cout << "RomanHorseman" << endl;
    }
};


// Классы всех видов воинов армии Карфагена
class CarthaginianInfantryman: public Infantryman
{
  public:
    void info() {
      cout << "CarthaginianInfantryman" << endl;
    }
};

```

Сквозной пример реализации и применения порождающих паттенов - 6

Реализация паттерна Abstract Factory

```
61
62  class CarthaginianArcher: public Archer
63  {
64    public:
65      void info() {
66        cout << "CarthaginianArcher" << endl;
67      }
68  };
69
70  class CarthaginianHorseman: public Horseman
71  {
72    public:
73      void info() {
74        cout << "CarthaginianHorseman" << endl;
75      }
76  };
77
78
79  // Абстрактная фабрика для производства воинов
80  class ArmyFactory
81  {
82    public:
83      virtual Infantryman* createInfantryman() = 0;
84      virtual Archer* createArcher() = 0;
85      virtual Horseman* createHorseman() = 0;
86      virtual ~ArmyFactory() {}
87  };
88
89
```

```
89
90  // Фабрика для создания воинов Римской армии
91  class RomanArmyFactory: public ArmyFactory
92  {
93    public:
94      Infantryman* createInfantryman() {
95        return new RomanInfantryman;
96      }
97      Archer* createArcher() {
98        return new RomanArcher;
99      }
100     Horseman* createHorseman() {
101       return new RomanHorseman;
102     }
103 };
104
105
106 // Фабрика для создания воинов армии Карфагена
107 class CarthaginianArmyFactory: public ArmyFactory
108 {
109   public:
110     Infantryman* createInfantryman() {
111       return new CarthaginianInfantryman;
112     }
113     Archer* createArcher() {
114       return new CarthaginianArcher;
115     }
116     Horseman* createHorseman() {
117       return new CarthaginianHorseman;
118     }
119 };
120
121
```

Сквозной пример реализации и применения порождающих паттернов - 7

Реализация паттерна Abstract Factory

```
122 // Класс, содержащий всех воинов той или иной армии
123 class Army
124 {
125   public:
126    ~Army() {
127       int i;
128       for(i=0; i<vi.size(); ++i)  delete vi[i];
129       for(i=0; i<va.size(); ++i)  delete va[i];
130       for(i=0; i<vh.size(); ++i)  delete vh[i];
131     }
132     void info() {
133       int i;
134       for(i=0; i<vi.size(); ++i)  vi[i]->info();
135       for(i=0; i<va.size(); ++i)  va[i]->info();
136       for(i=0; i<vh.size(); ++i)  vh[i]->info();
137     }
138     vector<Infantryman*> vi;
139     vector<Archer*> va;
140     vector<Horseman*> vh;
141 };
142
143
144 // Здесь создается армия той или иной стороны
145 class Game
146 {
147   public:
148     Army* createArmy( ArmyFactory& factory ) {
149       Army* p = new Army;
150       p->vi.push_back( factory.createInfantryman());
151       p->va.push_back( factory.createArcher());
152       p->vh.push_back( factory.createHorseman());
153       return p;
154     }
155 };
156
157
158 int main()
159 {
160     Game game;
161     RomanArmyFactory ra_factory;
162     CarthaginianArmyFactory ca_factory;
163
164     Army * ra = game.createArmy( ra_factory);
165     Army * ca = game.createArmy( ca_factory);
166     cout << "Roman army:" << endl;
167     ra->info();
168     cout << "\nCarthaginian army:" << endl;
169     ca->info();
170     // ...
171 }
```

Вывод программы будет следующим:

```
1 Roman army:
2 RomanInfantryman
3 RomanArcher
4 RomanHorseman
5
6 Carthaginian army:
7 CarthaginianInfantryman
8 CarthaginianArcher
9 CarthaginianHorseman
```

Сквозной пример реализации и применения порождающих паттенов - 8

Реализация паттерна Prototype

Также как и для паттерна **Factory Method** имеются две возможные реализации паттерна **Prototype**:

- **<u>В виде обобщенного конструктора</u>** на основе прототипов, когда в полиморфном базовом классе **Prototype** определяется статический метод, предназначенный для создания объектов. При этом в качестве параметра в этот метод должен передаваться идентификатор типа создаваемого объекта.
- **<u>На базе специально выделенного класса-фабрики</u>**.

Реализация паттерна Ptototype на основе обобщенного конструктора

```
#include <iostream>
#include <vector>
#include <map>

// Идентификаторы всех родов войск
enum Warrior_ID { Infantryman_ID, Archer_ID, Horseman_ID };

class Warrior;  // Опережающее объявление
typedef map<Warrior_ID, Warrior*> Registry;

// Реестр прототипов определен в виде Singleton Мэйерса
Registry& getRegistry()
{
  static Registry _instance;
  return _instance;
}

// Единственное назначение этого класса - помощь в выборе нужного
// конструктора при создании прототипов
class Dummy { };

// Полиморфный базовый класс. Здесь также определен статический
// обобщенный конструктор для создания боевых единиц всех родов войск
class Warrior
{
  public:
    virtual Warrior* clone() = 0;
    virtual void info() = 0;
    virtual ~Warrior() {}
    // Параметризированный статический метод для создания воинов
    // всех родов войск
    static Warrior* createWarrior( Warrior_ID id ) {
      Registry& r = getRegistry();
      if (r.find(id) != r.end())
        return r[id]->clone();
      return 0;
    }
  protected:
    // Добавление прототипа в множество прототипов
    static void addPrototype( Warrior_ID id, Warrior * prototype ) {
      Registry& r = getRegistry();
      r[id] = prototype;
    }
    // Удаление прототипа из множества прототипов
    static void removePrototype( Warrior_ID id ) {
      Registry& r = getRegistry();
      r.erase( r.find( id));
    }
};


// В производных классах различных родов войск в виде статических
// членов-данных определяются соответствующие прототипы
class Infantryman: public Warrior
{
  public:
    Warrior* clone() {
      return new Infantryman( *this);
    }
    void info() {
      cout << "Infantryman" << endl;
    }
```

Сквозной пример реализации и применения порождающих паттенов - 9

Реализация паттерна Ptototype на основе обобщенного конструктора-2

В виде обобщенного конструктора на основе прототипов, когда в полиморфном базовом классе **Prototype** определяется статический метод, предназначенный для создания объектов. При этом в качестве параметра в этот метод должен передаваться идентификатор типа создаваемого объекта.

```
63      private:
64        Infantryman( Dummy ) {
65          Warrior::addPrototype( Infantryman_ID, this);
66        }
67        Infantryman() {}
68        static Infantryman prototype;
69  };
70
71  class Archer: public Warrior
72  {
73    public:
74      Warrior* clone() {
75        return new Archer( *this);
76      }
77      void info() {
78        cout << "Archer" << endl;
79      }
80    private:
81      Archer(Dummy) {
82        addPrototype( Archer_ID, this);
83      }
84      Archer() {}
85      static Archer prototype;
86  };
87
88  class Horseman: public Warrior
89  {
90    public:
91      Warrior* clone() {
92        return new Horseman( *this);
93      }
```

```
94        void info() {
95          cout << "Horseman" << endl;
96        }
97    private:
98        Horseman(Dummy) {
99          addPrototype( Horseman_ID, this);
100       }
101       Horseman() {}
102       static Horseman prototype;
103 };
104
105
106 Infantryman Infantryman::prototype = Infantryman( Dummy());
107 Archer Archer::prototype = Archer( Dummy());
108 Horseman Horseman::prototype = Horseman( Dummy());
109
110
111 int main()
112 {
113   vector<Warrior*> v;
114   v.push_back( Warrior::createWarrior( Infantryman_ID));
115   v.push_back( Warrior::createWarrior( Archer_ID));
116   v.push_back( Warrior::createWarrior( Horseman_ID));
117
118   for(int i=0; i<v.size(); i++)
119     v[i]->info();
120   // ...
121 }
```

Сквозной пример реализации и применения порождающих паттенов - 10

Реализация паттерна Ptototype на основе обобщенного

```
  private:
    Infantryman( Dummy ) {
      Warrior::addPrototype( Infantryman_ID, this);
    }
    Infantryman() {}
    static Infantryman prototype;
};

class Archer: public Warrior
{
  public:
    Warrior* clone() {
      return new Archer( *this);
    }
    void info() {
      cout << "Archer" << endl;
    }
  private:
    Archer(Dummy) {
      addPrototype( Archer_ID, this);
    }
    Archer() {}
    static Archer prototype;
};

class Horseman: public Warrior
{
  public:
    Warrior* clone() {
      return new Horseman( *this);
    }
    void info() {
      cout << "Horseman" << endl;
    }
  private:
    Horseman(Dummy) {
      addPrototype( Horseman_ID, this);
    }
    Horseman() {}
    static Horseman prototype;
};


Infantryman Infantryman::prototype = Infantryman( Dummy());
Archer Archer::prototype = Archer( Dummy());
Horseman Horseman::prototype = Horseman( Dummy());


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

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

В этой реализации классы всех создаваемых военных единиц, таких как лучники, пехотинцы и конница, являются подклассами абстрактного базового класса **Warrior**, котором определен обобщенный конструктор в виде статического метода **createWarrior(Warrior_ID id)**. Передавая в этот метод в качестве параметра тип боевой
единицы, можно создавать воинов нужных родов войск. Для этого обобщенный конструктор использует реестр прототипов, реализованный в виде ассоциативного массива **std::map**, каждый элемент которого представляет собой пару "идентификатор типа воина" - "его прототип". Добавление прототипов в реестр происходит автоматически. В подклассах **Infantryman**, **Archer**, **Horseman**, прототипы определяются в виде статических членов данных тех же типов. При создании такого прототипа будет вызываться конструктор с параметром типа **Dummy**, который и добавит этот прототип в реестр прототипов с помощью метода **addPrototype()** базового класса Warrior. К этому моменту сам объект реестра должен быть полностью сконструирован, именно поэтому он выполнен в виде **singleton Мэйерса**. Для приведенной реализации паттерна **Prototype** можно отметить следующие особенности: Создавать новых воинов можно только при помощи обобщенного конструктора.
Их непосредственное создание невозможно, так как соответствующие конструкторы объявлены со

Реализация паттерна Prototype с помощью выделенного класса-

```
#include <iostream>
#include <vector>

// Иерархия классов игровых персонажей
// Полиморфный базовый класс
class Warrior
{
  public:
    virtual Warrior* clone() = 0;
    virtual void info() = 0;
    virtual ~Warrior() {}
};


// Производные классы различных родов
class Infantryman: public Warrior
{
    friend class PrototypeFactory;
  public:
    Warrior* clone() {
      return new Infantryman( *this);
    }
    void info() {
      cout << "Infantryman" << endl;
    }
  private:
    Infantryman() {}
};
```

```
class Archer: public Warrior
{
    friend class PrototypeFactory;
  public:
    Warrior* clone() {
      return new Archer( *this);
    }
    void info() {
      cout << "Archer" << endl;
    }
  private:
    Archer() {}
};

class Horseman: public Warrior
{
    friend class PrototypeFactory;
  public:
    Warrior* clone() {
      return new Horseman( *this);
    }
    void info() {
      cout << "Horseman" << endl;
    }
  private:
    Horseman() {}
};


// Фабрика для создания боевых единиц всех родов войск
class PrototypeFactory
```

```
 PrototypeFactory

lic:
Warrior* createInfantrman() {
  static Infantryman prototype;
  return prototype.clone();
}
Warrior* createArcher() {
  static Archer prototype;
  return prototype.clone();
}
Warrior* createHorseman() {
  static Horseman prototype;
  return prototype.clone();
}
```

```
int main()
{
  PrototypeFactory factory;
  vector<Warrior*> v;
  v.push_back( factory.createInfantrman());
  v.push_back( factory.createArcher());
  v.push_back( factory.createHorseman());

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

В приведенной реализации для упрощения кода реестр прототипов не ведется. Воины всех родов войск создаются при помощи соответствующих методов фабричного класса PrototypeFactory, где и определены прототипы в виде статических переменных.

Сквозной пример реализации и применения порождающих паттенов - 12

Реализация паттерна Builder -1

Реализация паттерна **Builder** на примере построения армий для военной стратегии "Пунические войны". Чтобы не нагромождать код лишними подробностями, пусть такие рода войск как пехота, лучники и конница для обеих армий идентичны. А с целью демонстрации возможностей паттерна Builder введем новые виды боевых единиц:

Катапульты для армии Рима.

Боевые слоны для армии Карфагена.

```
#include <iostream>
#include <vector>

// Классы всех возможных родов войск
class Infantryman
{
  public:
    void info() {
      cout << "Infantryman" << endl;
    }
};

class Archer
{
  public:
    void info() {
      cout << "Archer" << endl;
    }
};

class Horseman
{
  public:
    void info() {
      cout << "Horseman" << endl;
    }
};

class Catapult
{
  public:
    void info() {
      cout << "Catapult" << endl;
    }
};

class Elephant
{
  public:
    void info() {
      cout << "Elephant" << endl;
    }
};


// Класс "Армия", содержащий все типы боевых единиц
class Army
{
  public:
    vector<Infantryman> vi;
    vector<Archer>      va;
    vector<Horseman>    vh;
    vector<Catapult>    vc;
    vector<Elephant>    ve;
    void info() {
      int i;
      for(i=0; i<vi.size(); ++i)  vi[i].info();
      for(i=0; i<va.size(); ++i)  va[i].info();
      for(i=0; i<vh.size(); ++i)  vh[i].info();
      for(i=0; i<vc.size(); ++i)  vc[i].info();
      for(i=0; i<ve.size(); ++i)  ve[i].info();
    }
```

Сквозной пример реализации и применения порождающих паттернов - 13

Реализация паттерна Builder -2

```
};


// Базовый класс ArmyBuilder объявляет интерфейс для поэтапного
// построения армии и предусматривает его реализацию по умолчанию

class ArmyBuilder
{
  protected:
    Army* p;
  public:
    ArmyBuilder(): p(0) {}
    virtual ~ArmyBuilder() {}
    virtual void createArmy() {}
    virtual void buildInfantryman() {}
    virtual void buildArcher() {}
    virtual void buildHorseman() {}
    virtual void buildCatapult() {}
    virtual void buildElephant() {}
    virtual Army* getArmy() { return p; }
};


// Римская армия имеет все типы боевых единиц кроме боевы
class RomanArmyBuilder: public ArmyBuilder
{
  public:
    void createArmy() { p = new Army; }
    void buildInfantryman() { p->vi.push_back( Infantryma
    void buildArcher() { p->va.push_back( Archer()); }
    void buildHorseman() { p->vh.push_back( Horseman());
    void buildCatapult() { p->vc.push_back( Catapult()); }
};


// Армия Карфагена имеет все типы боевых единиц кроме катапульт
class CarthaginianArmyBuilder: public ArmyBuilder
{
  public:
    void createArmy() { p = new Army; }
    void buildInfantryman() { p->vi.push_back( Infantryman()); }
    void buildArcher() { p->va.push_back( Archer()); }
    void buildHorseman() { p->vh.push_back( Horseman()); }
    void buildElephant() { p->ve.push_back( Elephant()); }
};


// Класс-распорядитель, поэтапно создающий армию той или иной сторо
// Именно здесь определен алгоритм построения армии.
class Director
{
  public:
    Army* createArmy( ArmyBuilder & builder )
    {
        builder.createArmy();
        builder.buildInfantryman();
        builder.buildArcher();
        builder.buildHorseman();
        builder.buildCatapult();
        builder.buildElephant();
        return( builder.getArmy());
    }
```

Сквозной пример реализации и применения порождающих паттенов - 14

Реализация паттерна Builder -2

```
};


int main()
{
    Director dir;
    RomanArmyBuilder ra_builder;
    CarthaginianArmyBuilder ca_builder;

    Army * ra = dir.createArmy( ra_builder);
    Army * ca = dir.createArmy( ca_builder);
    cout << "Roman army:" << endl;
    ra->info();
    cout << "\nCarthaginian army:" << endl;
    ca->info();
    // ...

    return 0;
}
```

Вывод программы будет следующим:

```
Roman army:
Infantryman
Archer
Horseman
Catapult

Carthaginian army:
Infantryman
Archer
Horseman
Elephant
```

Часто базовый класс строителя (в коде выше это **ArmyBuilder**) не только объявляет интерфейс для построения частей продукта, но и определяет реализацию по умолчанию, которая ничего не делает.

Тогда соответствующие подклассы (**RomanArmyBuilder**, **CarthaginianArmyBuilder**) переопределяют только те методы, которые участвуют в построении текущего объекта.

Так класс **RomanArmyBuilder** не определяет метод **buildElephant**, поэтому Римская армия не может иметь слонов.

А в классе **CarthaginianArmyBuilder** не определен **buildCatapult(),** поэтому армия Карфагена не может иметь катапульты.

Сравнение приведенного кода с кодом создания армии в реализации паттерна **Abstract Factory**.

Если паттерн **Abstract Factory** акцентирует внимание на создании семейств некоторых объектов

паттерн **Builder** подчеркивает поэтапное построение продукта.

При этом класс **Builder** скрывает все подробности построения сложного продукта так, что **Director** ничего не знает о его составных частях.