Типовые решения проектирования. Порождающие паттерны

Основная цель: независимость процесса инициализации объектов. Система становится независимой от способа создания, композиции и представления объектов.

Особенности:

- Система зависит от композиции классов, а не от иерархии наследования.
- Основной упор делается не на определении и кодировании фиксированного набора поведений, а на определении некоторого базового множества поведений, комбинируя которые можно получить все, что нужно.
- Для создания объектов с конкретным поведением требуется нечто большее, чем простая инициализация.

Порождающие паттерны.

Две характерные особенности:

1. Инкапсуляция знаний о конкретных классах, которые применяются в системе.
2. Скрываются детали того, каким образом эти классы создаются и стыкуются.

Единственная информация, которая доступна системе – это интерфейсы объектов, которые определяются с помощью абстрактных классов. Таким образом получаем ответы на следующие вопросы:

- что создается,
- кто создает,
- и когда.

В результате можно собрать систему из готовых объектов с самой различной структурой либо статически, либо динамически.

Порождающие паттерны
Список и краткие характеристики:

- Фабричный метод [Factory Method] – определяет интерфейс для создания объектов, при этом непосредственное создание объектов выполняется подклассами.
- Абстрактная фабрика [Abstract Factory] – представляет интерфейс для создания семейств связанных между собой или независимых объектов, конкретные классы которых неизвестны.
- Строитель [Builder] – отделяет сборку сложного объекта от его представления, позволяет использовать один и тот же процесс конструирования для создания различных представлений.
- Прототип [Prototype] – описывает виды создаваемых объектов путем прототипа, инициализирует новый объект путем копирования прототипа.
- Одиночка [Singleton] – гарантирует, что некоторый класс будет представлен в виде единственного экземпляра и предоставляет точку доступа к этому экземпляру.

Фабричный метод [Factory Method]

Проблематика:

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

Рассмотрим каркас для приложений, способных представлять пользователю сразу несколько документов. Две основных абстракции в таком каркасе - это классы Application и Document.

Оба класса абстрактные, поэтому клиенты должны порождать от них подклассы для создания специфичных для приложения реализаций. Например, чтобы создать приложение для рисования, мы определим классы DrawingApplication и DrawingDocument.

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

Фабричный метод [Factory Method]

Проблематика:

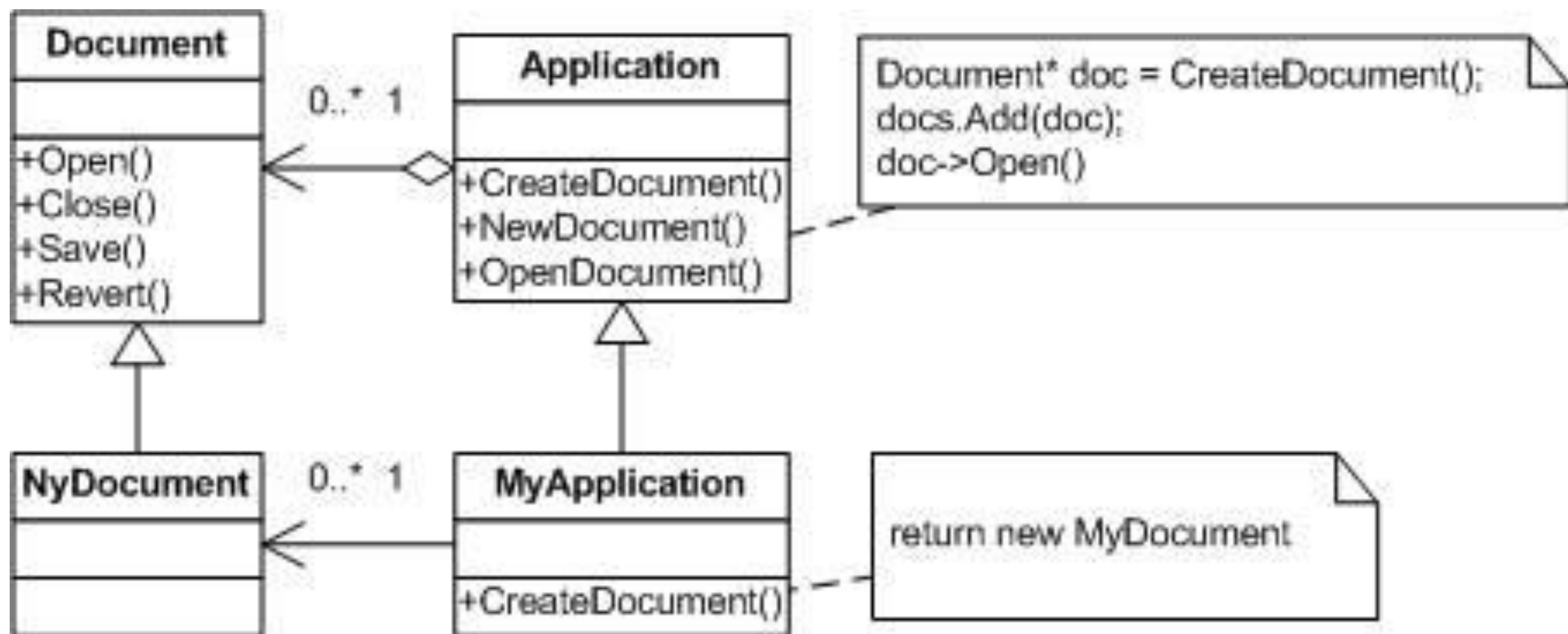
Поскольку решение о том, какой подкласс класса Document инициализировать, зависит от приложения, то Application не может "предсказать", что именно понадобится. Этому классу известно лишь, когда нужно инициализировать новый документ, а не какой документ создать. Возникает дилемма: каркас должен инициализировать классы, но "знает" он лишь об абстрактных классах, которые создавать нельзя.

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

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

Фабричный метод [Factory Method]

Применимость:



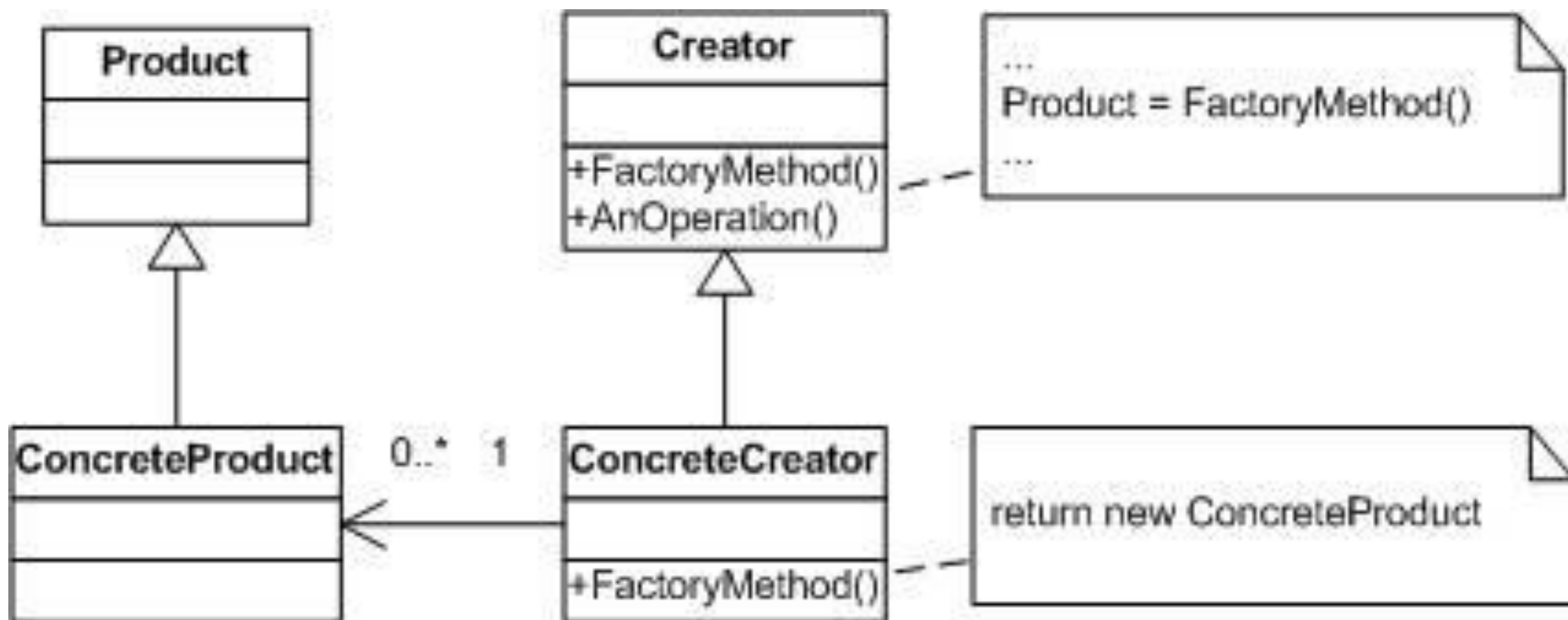
Фабричный метод [Factory Method]

Применимость:

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

Фабричный метод [Factory Method]

Применимость:



Фабричный метод [Factory Method]

Участники:

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

Отношения

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

Фабричный метод [Factory Method]

Результаты:

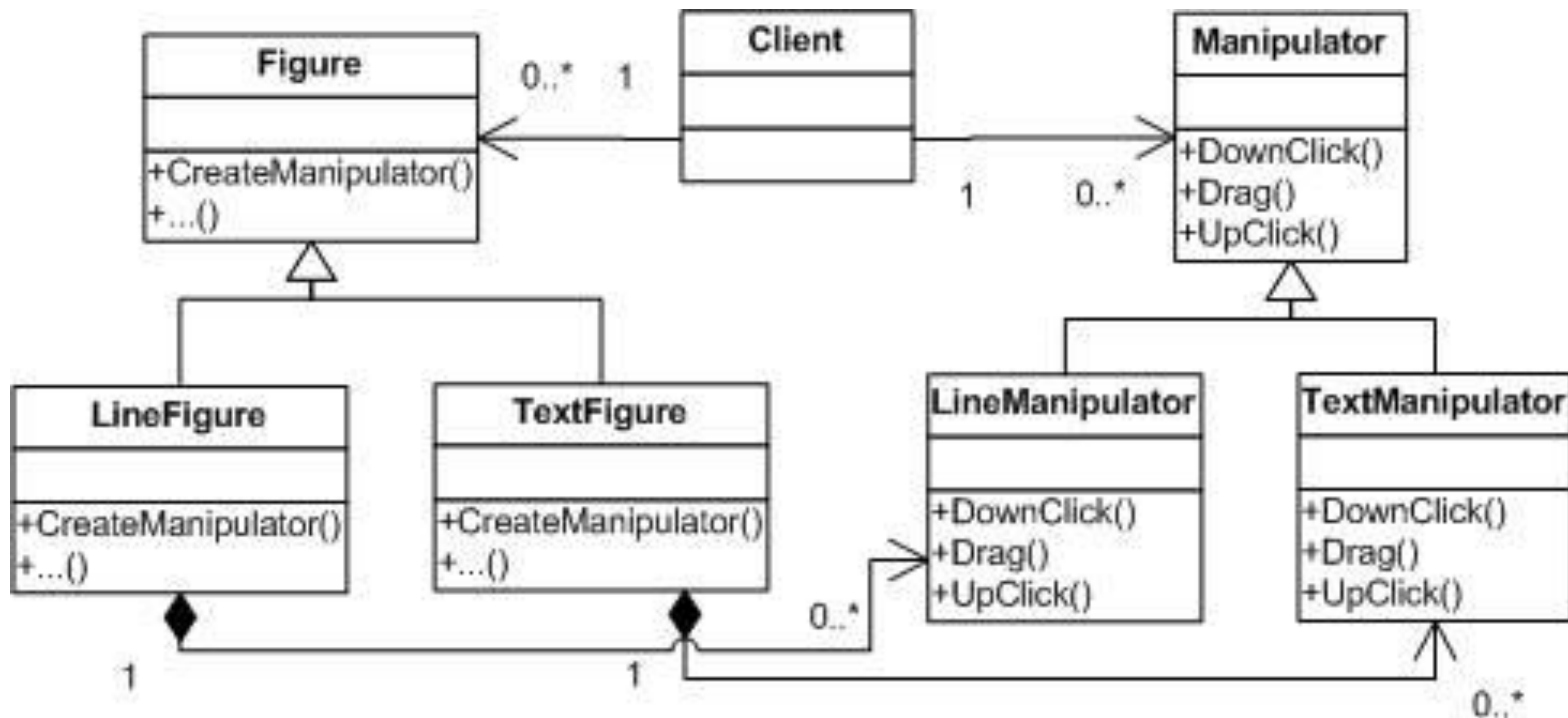
- Фабричные методы избавляют проектировщика от необходимости встраивать в код зависящие от приложения классы. Код имеет дело только с интерфейсом класса Product, поэтому он может работать с любыми определенными пользователями классами конкретных продуктов. Потенциальный недостаток фабричного метода состоит в том, что клиентам, возможно, придется создавать подкласс класса Creator для создания лишь одного объекта ConcreteProduct. Порождение подклассов оправдано, если клиенту так или иначе приходится создавать подклассы Creator, в противном случае клиенту придется иметь дело с дополнительным уровнем подклассов.
- Фабричный метод предоставляет подклассам операции-зацепки (hooks). Создание объектов внутри класса с помощью фабричного метода всегда оказывается более гибким решением, чем непосредственное создание. Фабричный метод создает в подклассах операции-зацепки для предоставления расширенной версии объекта. В примере с документом класс Document мог бы определить фабричный метод CreateFileDialog, который создает диалоговое окно для выбора файла существующего документа. Подкласс этого класса мог бы определить специализированное для приложения диалоговое окно, заместив этот фабричный метод. В данном случае фабричный метод не является абстрактным, . а содержит разумную реализацию по умолчанию.

- Фабричный метод соединяет параллельные иерархии. В примерах, которые мы рассматривали до сих пор, фабричные методы вызывались только создателем. Но это совершенно необязательно: клиенты тоже могут применять фабричные методы, особенно при наличии параллельных иерархий классов. Параллельные иерархии возникают в случае, когда класс делегирует часть своих обязанностей другому классу, не являющемуся производным от него. Рассмотрим, например, графические фигуры, которыми можно манипулировать интерактивно: растягивать, двигать или вращать с помощью мыши. Реализация таких взаимодействий с пользователем - не всегда простое дело. Часто приходится сохранять и обновлять информацию о текущем состоянии манипуляции. Но это состояние нужно только во время самой манипуляции, поэтому помещать его в объект, представляющий-фигуру, не следует. К тому же фигуры ведут себя по-разному, когда пользователь манипулирует ими. Например, растягивание отрезка может сводиться к изменению положения концевой точки, а растягивание текста - к изменению междустрочных интервалов.

При таких ограничениях лучше использовать отдельный объект-манипулятор Manipulator, который реализует взаимодействие и контролирует его текущее состояние. У разных фигур будут разные манипуляторы, являющиеся подклассом Manipulator. Получающаяся иерархия класса Manipulator параллельна (по крайней мере, частично) иерархии класса Figure. Класс Figure предоставляет фабричный метод CreateManipulator, который позволяет клиентам создавать соответствующий фигуре манипулятор. Подклассы Figure замещают этот метод так, чтобы он возвращал подходящий для них подкласс Manipulator. Вместо этого класс Figure может реализовать CreateManipulator так, что он будет возвращать экземпляр класса Manipulator по умолчанию, а подклассы Figure могут наследовать это умолчание. Те классы фигур, которые функционируют по описанному принципу, не нуждаются в специальном манипуляторе, поэтому иерархии параллельны только отчасти.

Фабричный метод [Factory Method]

Двойная иерархия:



Реализация:

две основных разновидности паттерна.

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

Абстрактная фабрика [Abstract Factory]
Проблематика:

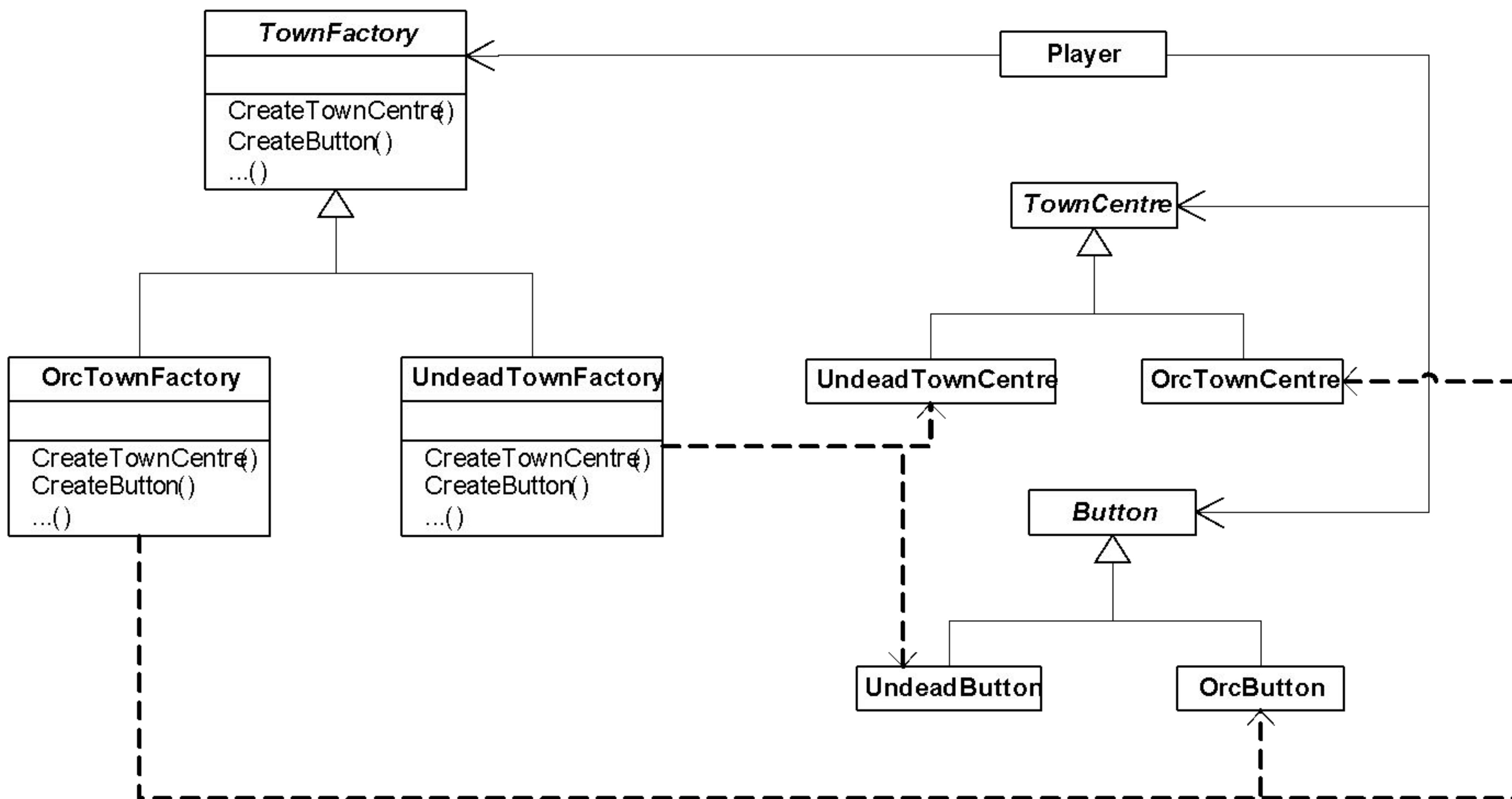
Вернемся теперь к нашей мега крутой игре. Предположим, что наша игра должна поддерживать разные стандарты внешнего вида для городов, заселенных различными расами (например, города орков должны быть городами орков, а города андедов – андедскими). Внешний вид при этом определяет визуальное представление и поведение элементов интерфейса пользователя – здания, окна, кнопки и т.д. и т.п.

Чтобы в наше приложение можно было легко добавлять новые расы (например, белых пушистых кроликов) мы должны отказать от жесткой привязки к соответствующим элементам интерфейса.

Тем более мы должны избегать ситуации, когда инициализация элементов города будет разбросана по приложению (например, орочьи города строятся в орочьих модулях, а андедовские в андедовских).

Давайте разберем один из способов решения этой проблемы.

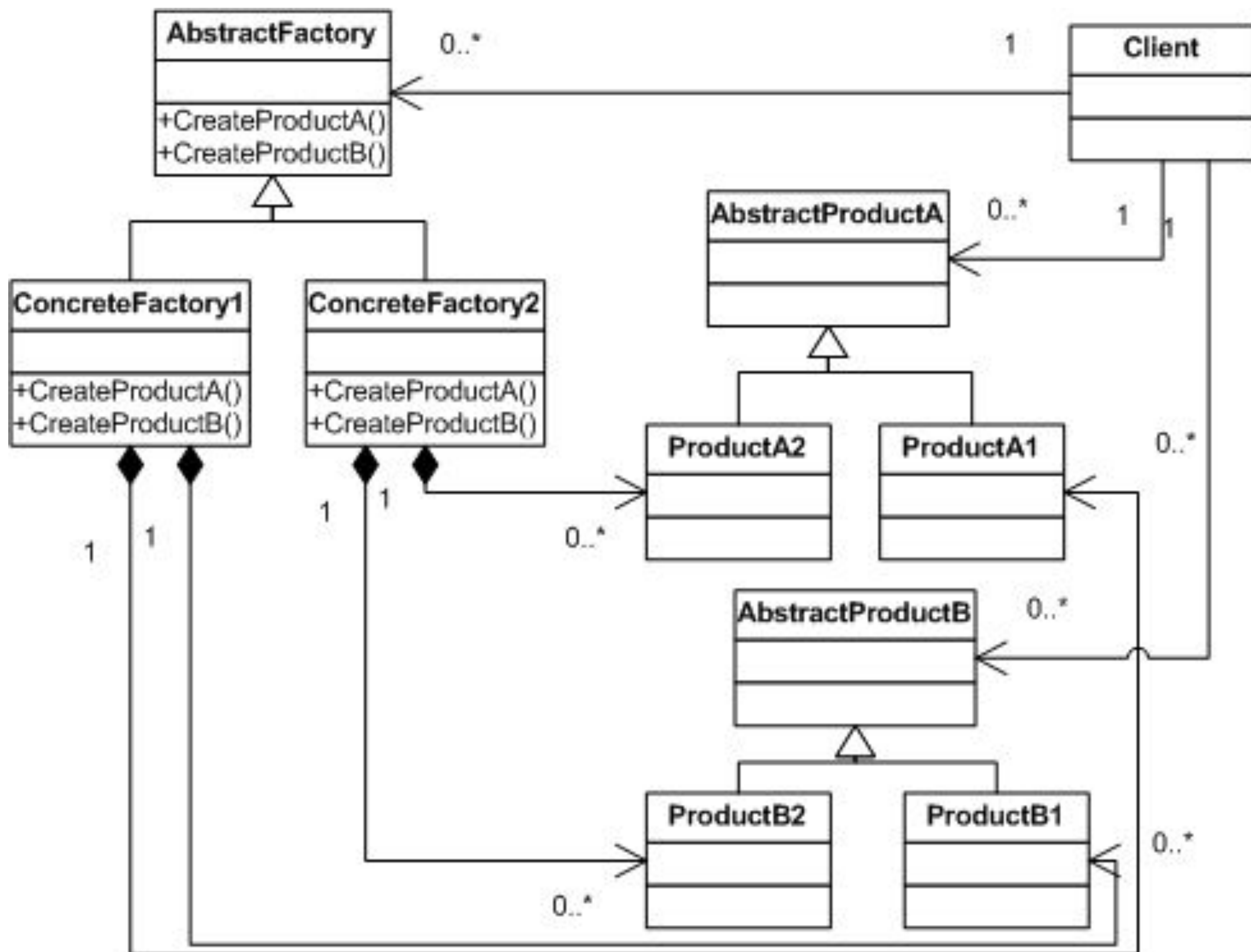
Абстрактная фабрика [Abstract Factory]
Применимость:



Абстрактная фабрика [Abstract Factory]
Применимость:

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

Абстрактная фабрика [Abstract Factory]
Структура:



Участники:

- AbstractFactory (TownFactory) - абстрактная фабрика: объявляет интерфейс для операций, создающих абстрактные объекты-продукты;
- CocreteFactory (OrcTownFactory, UndeadTownFactory) - конкретная фабрика - реализует операции, создающие конкретные объекты-продукты;
- AbstractProduct (TownCentre, Button) - абстрактный продукт, объявляет интерфейс для типа объекта-продукта;
- ConcreteProduct (OrcTownCentre, OrcButton, UndeadTownCentre, UndeadButton) - конкретный продукт: - определяет объект-продукт, создаваемый соответствующей конкретной фабрикой;
  - реализует интерфейс AbstractProduct;
- Client – клиент, пользуется исключительно интерфейсами, которые объявлены в классах AbstractFactory и AbstractProduct.

Отношения

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

Абстрактная фабрика (Abstract Factory)
Результаты:

Обладает следующими плюсами и минусами:

- изолирует конкретные классы. Помогает контролировать классы объектов, создаваемых приложением. Поскольку фабрика инкапсулирует ответственность за создание классов и сам процесс их создания, то она изолирует клиента от деталей реализации классов. Клиенты манипулируют экземплярами через их абстрактные интерфейсы. Имена изготавливаемых классов известны только конкретной фабрике, в коде клиента они не упоминаются;
- упрощает замену семейств продуктов. Класс конкретной фабрики появляется в приложении только один раз: при инициализации. Это облегчает замену используемой приложением конкретной фабрики. Приложение может изменить конфигурацию продуктов, просто подставив новую конкретную фабрику. Поскольку абстрактная фабрика создает все семейство продуктов, то и заменяется сразу все семейство. В нашем примере перейти от зданий Орков к зданиям Андедов можно, просто переключившись на продукты соответствующей фабрики и заново создав интерфейс;

Абстрактная фабрика (Abstract Factory) Результаты:

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

Абстрактная фабрика (Abstract Factory)

Некоторые полезные приемы реализации:

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

Абстрактная фабрика (Abstract Factory)

Некоторые полезные приемы реализации:

- Если семейств продуктов может быть много, то конкретную фабрику удастся реализовать с помощью паттерна прототип. В этом случае она инициализируется экземпляром-прототипом каждого продукта в семействе и создает новый продукт путем клонирования этого прототипа. Подход на основе прототипов устраняет необходимость создавать новый класс конкретной фабрики для каждого нового семейства продуктов.
- В языках, где сами классы являются настоящими объектами (например, Smalltalk и Objective C), возможны некие вариации подхода на базе прототипов. В таких языках класс можно представлять себе как вырожденный случай фабрики, умеющей создавать только один вид продуктов. Можно хранить классы внутри конкретной фабрики, которая создает разные конкретные продукты в переменных. Это очень похоже на прототипы. Такие классы создают новые экземпляры от имени конкретной фабрики. Новая фабрика инициализируется экземпляром конкретной фабрики с классами продуктов, а не путем порождения подкласса. Подобный подход задействует некоторые специфические свойства языка, тогда как подход, основанный на прототипах, от языка не зависит.

Прототип [Prototype]

Проблематика:

Мы продолжаем разрабатывать нашу супер-пупер игру. Реализуем функционал, позволяющий пользователю создавать армию. Предположим, что для этих целей мы используем некоторую библиотеку, которая позволяет работать с графическими объектами. Нам необходимо добавить в эту библиотеку новые объекты, представляющие воинов, героев, всадников, монстров и т.д.

Предположим, что у нас в библиотеке имеется возможность группировать объекты на некоторых панелях. Эта панель содержит инструменты для выбора, перемещения и иных манипуляции с объектами. Так, игрок, щелкнув, например, по пиктограмме героя помещает его в армию. А используя инструмент перемещения формирует боевой порядок войск, перемещая их вверх или вниз.

Предположим, что в библиотеке имеется:

- абстрактный класс Graphic для графических компонентов вроде героев, солдат и т.д.
- также абстрактный класс Tool для определения инструментов на панели
- кроме того, имеется предопределенный подкласс GraphicTool для инструментов, которые создают графические объекты и добавляют их в документ.

Прототип [Prototype]

Проблематика:

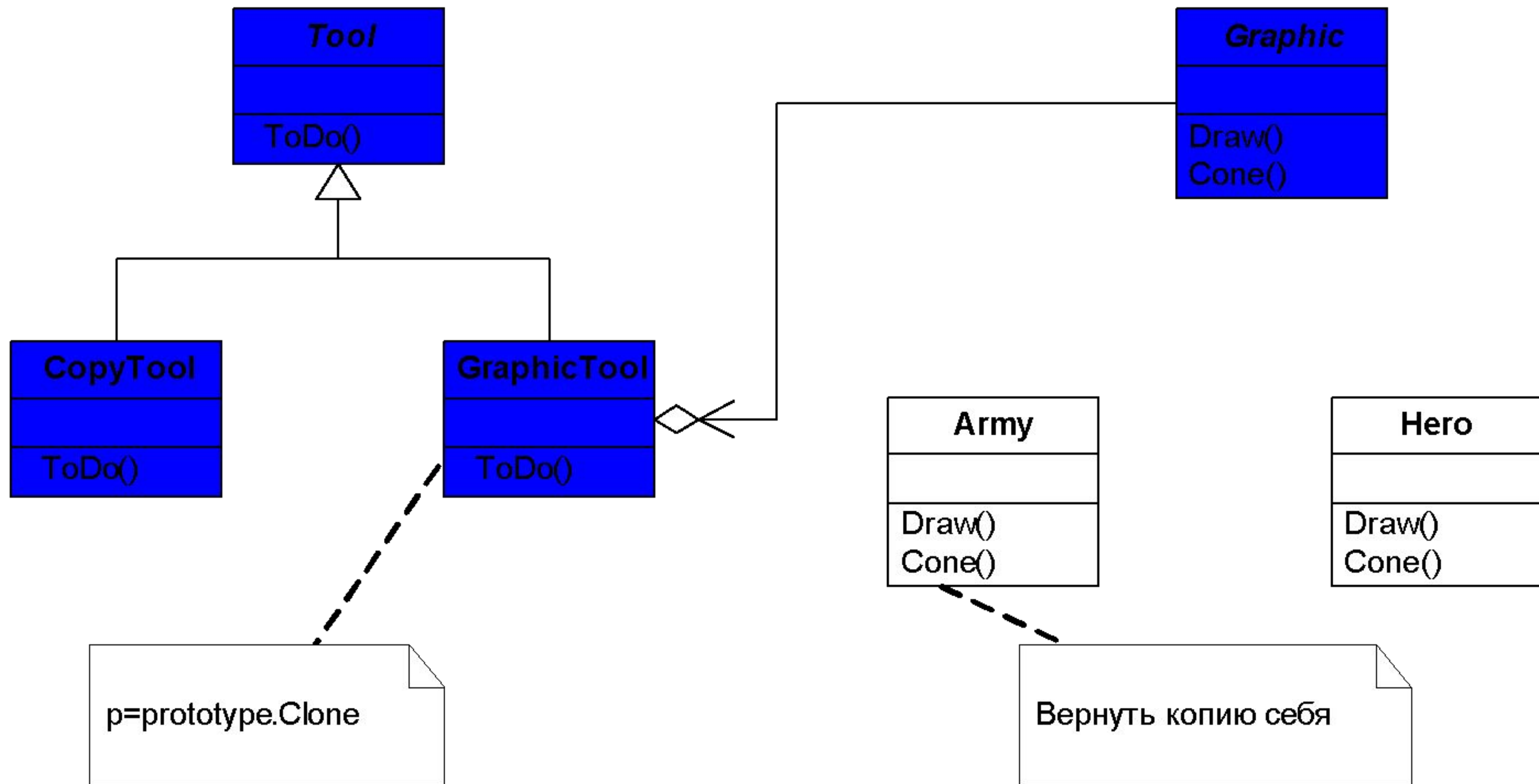
Однако класс GraphicTool создаст некую проблему для проектировщика библиотеки. Классы героев, солдат и монстров для нашего приложения, а класс GraphicTool принадлежит библиотеке. Для того, чтобы его можно было использовать повторно, он ничего не должен знать о том, как создавать экземпляры наших игровых классов и добавлять их в армию.

Можно было бы породить от GraphicTool подклассы для каждого вида наших объектов, по тогда оказалось бы слишком много классов, отличающихся только тем, какой объект армии они инсталлируют.

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

Итак, в нашем редакторе армий каждый инструмент для создания элемента армии - это экземпляр класса GraphicTool, инициализированный тем или иным прототипом. Любой экземпляр GraphicTool будет создавать армейский объект, клонируя его прототип и добавляя клон в армейский строй.

Прототип [Prototype]
Проблематика:

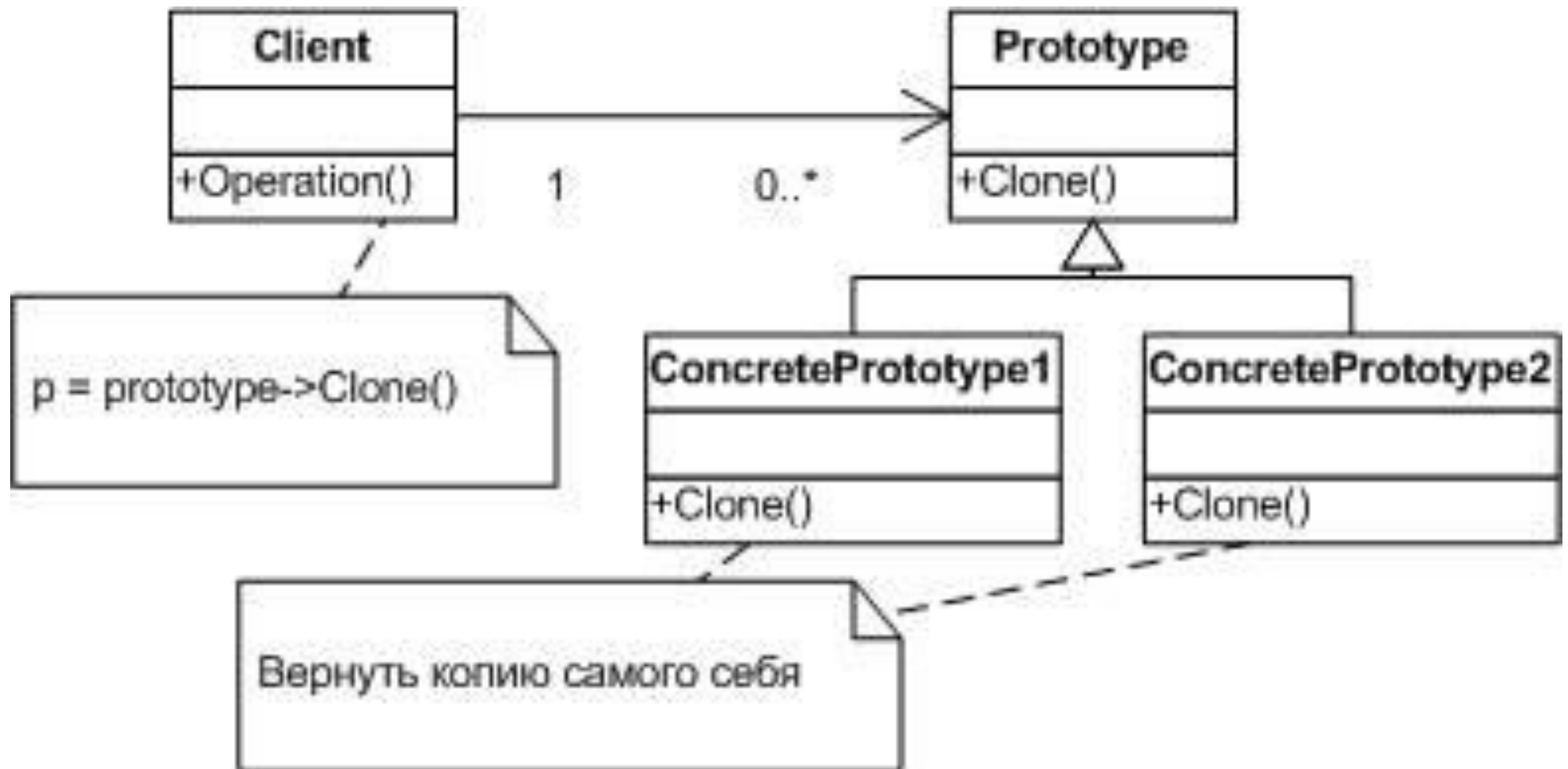


Прототип [Prototype]

Применимость:

- система не должна зависеть от того, как в ней создаются, компонуются и представляются продукты;
- инициализируемые классы определяются во время выполнения, например с помощью динамической загрузки;
- для того чтобы избежать построения иерархий классов или фабрик, параллельных иерархии классов продуктов;
- экземпляры класса могут находиться в одном из не очень большого числа различных состояний. Может оказаться удобнее установить соответствующее число прототипов и клонировать их, а не инициализировать каждый раз класс вручную в подходящем состоянии.

Прототип [Prototype]
Проблематика:



Прототип [Prototype]
Участники и отношения:

Участники:

- Prototype (Graphic) - Прототип: - объявляет интерфейс для клонирования самого себя;
- ConcretePrototype (Army- армия, Hero - герой) - конкретный прототип: - реализует операцию клонирования себя;
- Client (GraphicTool) - клиент: - создает новый объект, обращаясь к прототипу с запросом клонировать себя.

Отношения:

- Клиент обращается к прототипу, чтобы тот создал свою копию.

Прототип [Prototype]
Результаты:

У прототипа те же самые результаты, что у абстрактной фабрики и строителя: он скрывает от клиента конкретные классы продуктов, уменьшая тем самым число известных клиенту имен. Кроме того, все эти паттерны позволяют клиентам работать со специфичными для приложения классами без модификаций.

<u>Дополнительные преимущества паттерна прототип:</u>

- Основной недостаток паттерна прототип заключается в том, что каждый подкласс класса Prototype должен реализовывать операцию Clone, а это далеко не всегда просто. Например, сложно добавить операцию Clone, когда рассматриваемые классы уже существуют. Проблемы возникают и в случае, если во внутреннем представлении объекта есть другие объекты или наличествуют круговые ссылки.
- добавление и удаление продуктов во время выполнения. Прототип позволяет включать новый конкретный класс продуктов в систему, просто сообщив клиенту о новом экземпляре-прототипе. Это несколько более гибкое решение по сравнению с тем, что удастся сделать с помощью других порождающих паттернов, ибо клиент может устанавливать и удалять прототипы во время выполнения;

Прототип [Prototype]
Результаты:

- спецификация новых объектов путем изменения значений. Динамичные системы позволяют определять поведение за счет композиции объектов - например, путем задания значений переменных объекта, - а не с помощью определения новых классов. По сути дела, вы определяете новые виды объектов, инициализируя уже существующие классы и регистрируя их экземпляры как прототипы клиентских объектов. Клиент может изменить поведение, делегируя свои обязанности прототипу. Такой дизайн позволяет пользователям определять новые классы без программирования. Фактически клонирование объекта аналогично инициализации класса. Паттерн прототип может резко уменьшить число необходимых системе классов, В нашем редакторе с помощью одного только класса GraphicTool удастся создать бесконечное разнообразие армейских объектов;

Прототип [Prototype]
Результаты:

- специфицирование новых объектов путем изменения структуры. Многие приложения строят объекты из крупных и мелких составляющих. Например, редакторы для проектирования печатных плат создают электрические схемы из подсхем (для таких приложений характерны паттерны компоновщик и декоратор). Такие приложения часто позволяют инициализировать сложные, определенные пользователем структуры, скажем, для многократного использования некоторой подсхемы. Паттерн прототип поддерживает и такую возможность. Мы просто добавляем подсхему как прототип в палитру доступных элементов схем. При условии, что объект, представляющий составную схему, реализует операцию Clone как глубокое копирование, схемы с разными структурами могут выступать в качестве прототипов;

Прототип [Prototype]
Результаты:

- уменьшение числа подклассов. Паттерн фабричный метод часто порождает иерархию классов Creator, параллельную иерархии классов продуктов. Прототип позволяет клонировать прототип, а не запрашивать фабричный метод создать новый объект. Поэтому иерархия класса Creator становится вообще ненужной;
- динамическое конфигурирование приложения классами. Некоторые среды позволяют динамически загружать классы в приложение во время его выполнения. Паттерн прототип - это ключ к применению таких возможностей в языке типа С++. Приложение, которое создает экземпляры динамически загружаемого класса, не может обращаться к его конструктору статически. Вместо этого исполняющая среда автоматически создает экземпляр каждого класса в момент его загрузки и регистрирует экземпляр в диспетчере прототипов (см. раздел «Реализация»). Затем приложение может запросить у диспетчера прототипов экземпляры вновь загруженных классов, которые изначально не были связаны с программой.

Прототип [Prototype]
Некоторые полезные приемы реализации:

- использование диспетчера прототипов. Если число прототипов в системе не фиксировано (то есть они могут создаваться и уничтожаться динамически), ведите реестр доступных прототипов. Клиенты должны не управлять прототипами самостоятельно, а сохранять и извлекать их из реестра. Клиент запрашивает прототип из реестра перед его клонированием. Такой реестр называют диспетчером прототипов. Диспетчер прототипов - это ассоциативное хранилище, которое возвращает прототип, соответствующий заданному ключу. В нем есть операции для регистрации прототипа с указанным ключом и отмены регистрации. Клиенты могут изменять и даже «просматривать реестр во время выполнения, а значит, расширять систему и вести контроль над ее состоянием без написания кода;

Прототип [Prototype]
Некоторые полезные приемы реализации:

- реализация операции Clone. Самая трудная часть паттерна прототип - правильная реализация операции clone. Особенно сложно это в случае, когда в структуре объекта есть круговые ссылки. В большинстве языков имеется некоторая поддержка для клонирования объектов. Но эти средства не решают проблему "глубокого и поверхностного копирования". Суть ее в следующем: должны ли при клонировании объекта клонироваться также и его переменные экземпляра или клон просто разделяет с оригиналом эти переменные? Поверхностное копирование просто, и часто его бывает достаточно. Но для клонирования прототипов со сложной структурой обычно необходимо глубокое копирование, поскольку клон должен быть независим от оригинала. Поэтому нужно гарантировать, что компоненты клона являются клонами компонентов прототипа. При клонировании вам приходится решать, что именно может разделяться и может ли вообще. Если объекты в системе предоставляют операции Save (сохранить) и Load (загрузить), то разрешается воспользоваться ими для реализации операции Clone по умолчанию, просто сохранив и сразу же загрузив объект. Операция Save сохраняет объект в буфере памяти, а Load создает дубликат, реконструируя объект из буфера;

Прототип [Prototype]
Некоторые полезные приемы реализации:

- инициализация клонов. Хотя некоторым клиентам вполне достаточно клона как такового, другим нужно инициализировать его внутреннее состояние полностью или частично. Обычно передать начальные значения операции Clone невозможно, поскольку их число различно для разных классов прототипов. Для некоторых прототипов нужно много параметров инициализации, другие вообще ничего не требуют. Передача Clone параметров мешает построению единообразного интерфейса клонирования. Может оказаться, что к ваших классах прототипов уже определяются операции для установки и очистки некоторых важных элементов состояния. Если так, то этими операциями можно воспользоваться сразу после клонирования. В противном случае, возможно, понадобится ввести операцию Initialize, которая принимает начальные значения в качестве аргументов и соответственно устанавливает внутреннее состояние клона. Будьте осторожны, если операция clone реализует глубокое копирование: копии может понадобиться удалять (явно или внутри Initialize) перед повторной инициализацией.

Строитель [Builder]

Проблематика:

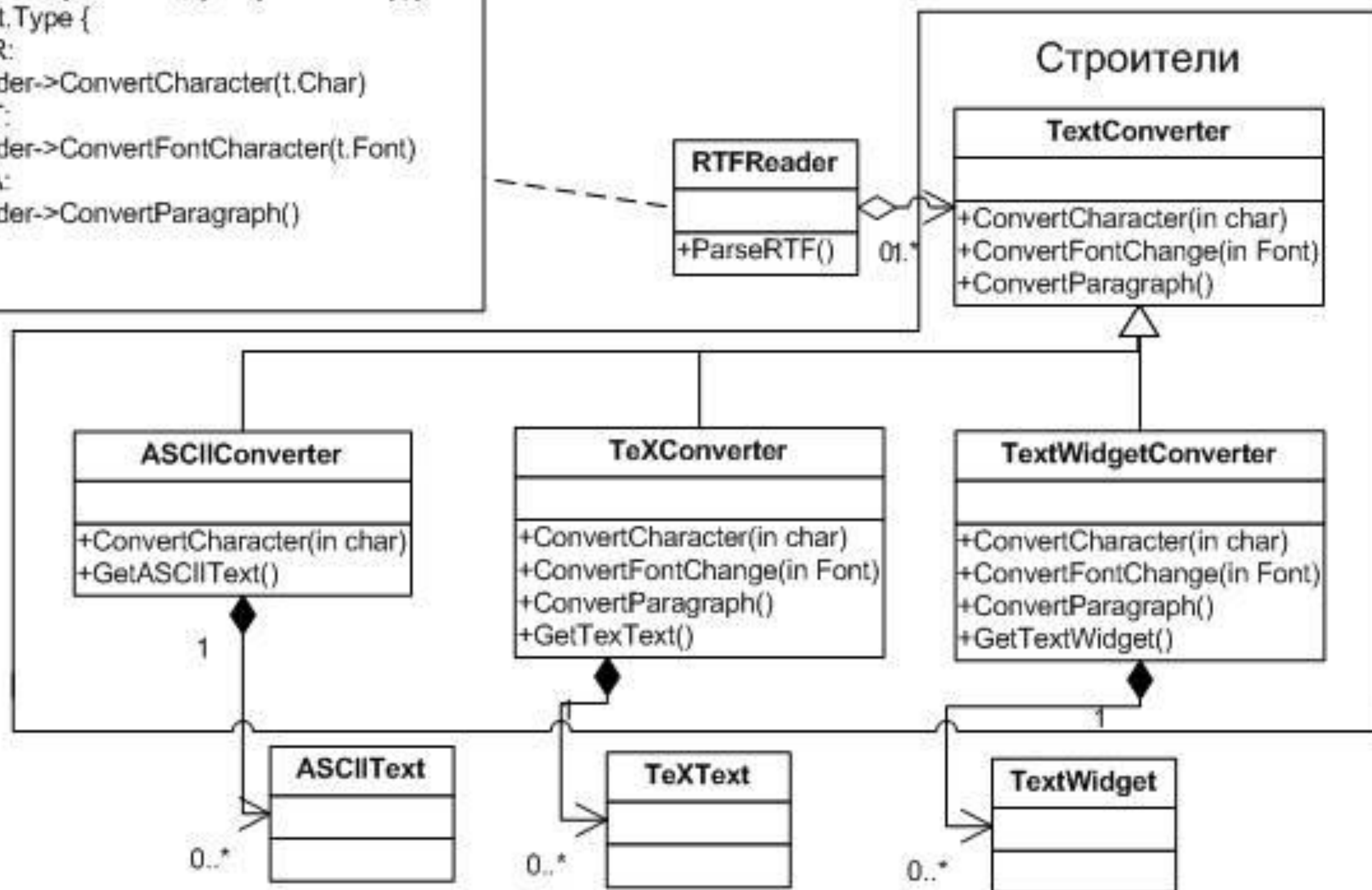
Программа, в которую заложена возможность распознавания и чтения документа в формате RTF (Rich Text Format), должна также "уметь" преобразовывать его во многие другие форматы. Однако число вероятных преобразований заранее неизвестно. Поэтому должна быть обеспечена возможность без труда добавлять новый конвертор.

Таким образом, нужно сконфигурировать класс RTFReader с помощью объекта TextConverter, который мог бы преобразовывать RTF в другой текстовый формат. При разборе документа в формате RTF класс RTFReader вызывает TextConverter для выполнения преобразования. Всякий раз, как RTFReader распознает лексему RTF (простой текст или управляющее слово), для ее преобразования объекту TextConverter посылается запрос. Объекты TextConverter отвечают как за преобразование данных, так и за представление лексемы в конкретном формате. Подклассы TextConverter специализируются на различных преобразованиях и форматах.

Класс каждого конвертора принимает механизм создания и сборки сложного объекта и скрывает его за абстрактным интерфейсом. Конвертор отделен от загрузчика, который отвечает за синтаксический разбор RTF-документа. В паттерне строитель абстрагированы все эти отношения. В нем любой класс конвертора называется строителем, а загрузчик - распорядителем..

Строитель [Builder]

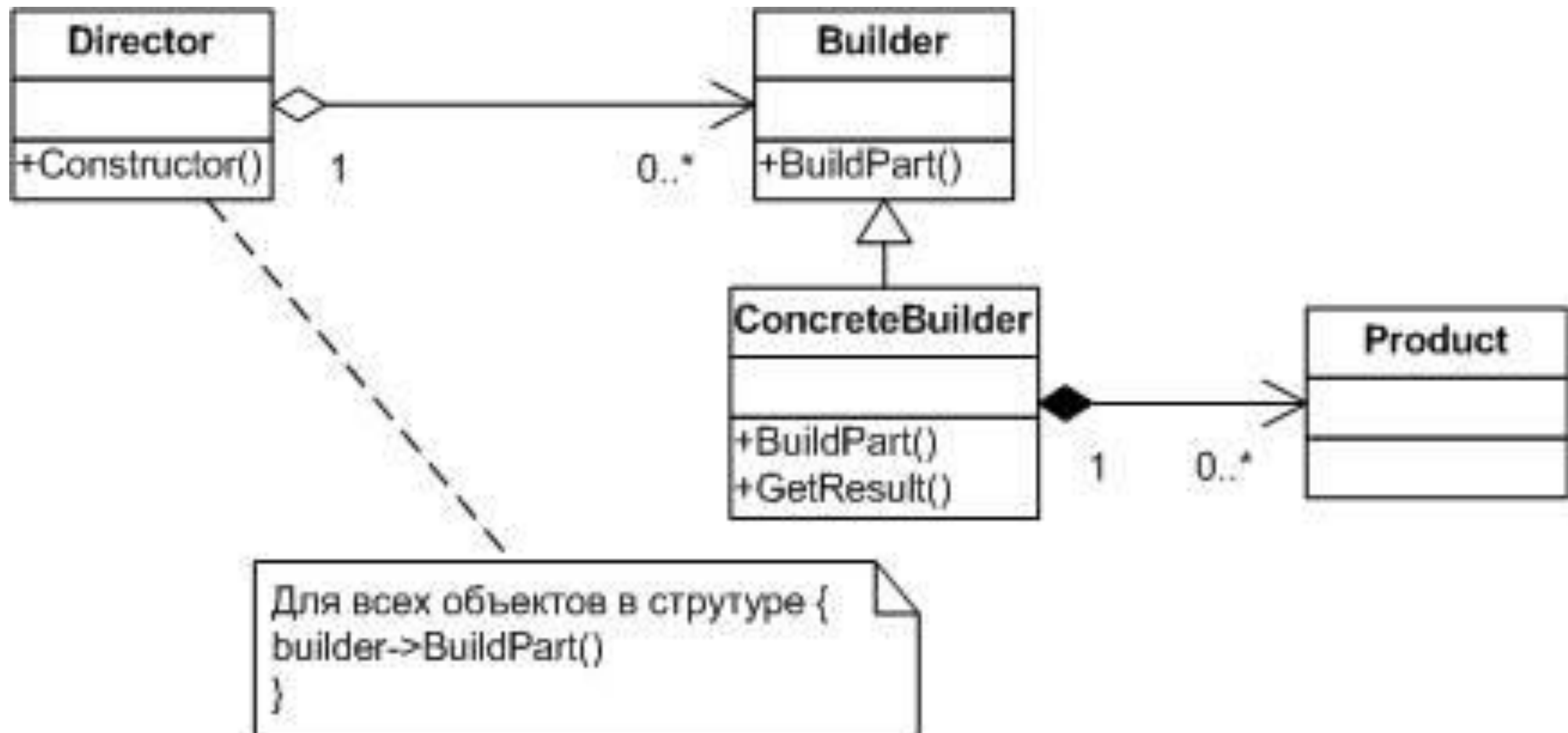
Проблематика:



Строитель [Builder]

Применимость:

- алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой;
- процесс конструирования должен обеспечивать различные представления конструируемого объекта.



Строитель [Builder]
Участники и отношения:

Участники:

- Builder (TextConverter) -строитель: - задает абстрактный интерфейс для создания частей объекта Product;
- ConcreteBuilder (ASCIIConverter,TeXConverter,TextWidgetConverter)- конкретный строитель: - конструирует и собирает вместе части продукта посредством реализации интерфейса Builder;

  - определяет создаваемое представление и следит за ним;

  - предоставляет интерфейс для доступа к продукту (например, GetASCIIText,GetTextWidget);
- Director (RTFReader) - распорядитель: - конструирует объект, пользуясь интерфейсом Builder;
- Product (ASCIIText, TeXText, TextWidget) - продукт: - представляет сложный конструируемый объект. ConcreteBuilder строит внутреннее представление продукта и определяет процесс его сборки;

  - включает классы, которые определяют составные части, в том числе интерфейсы для сборки конечного результата из частей.

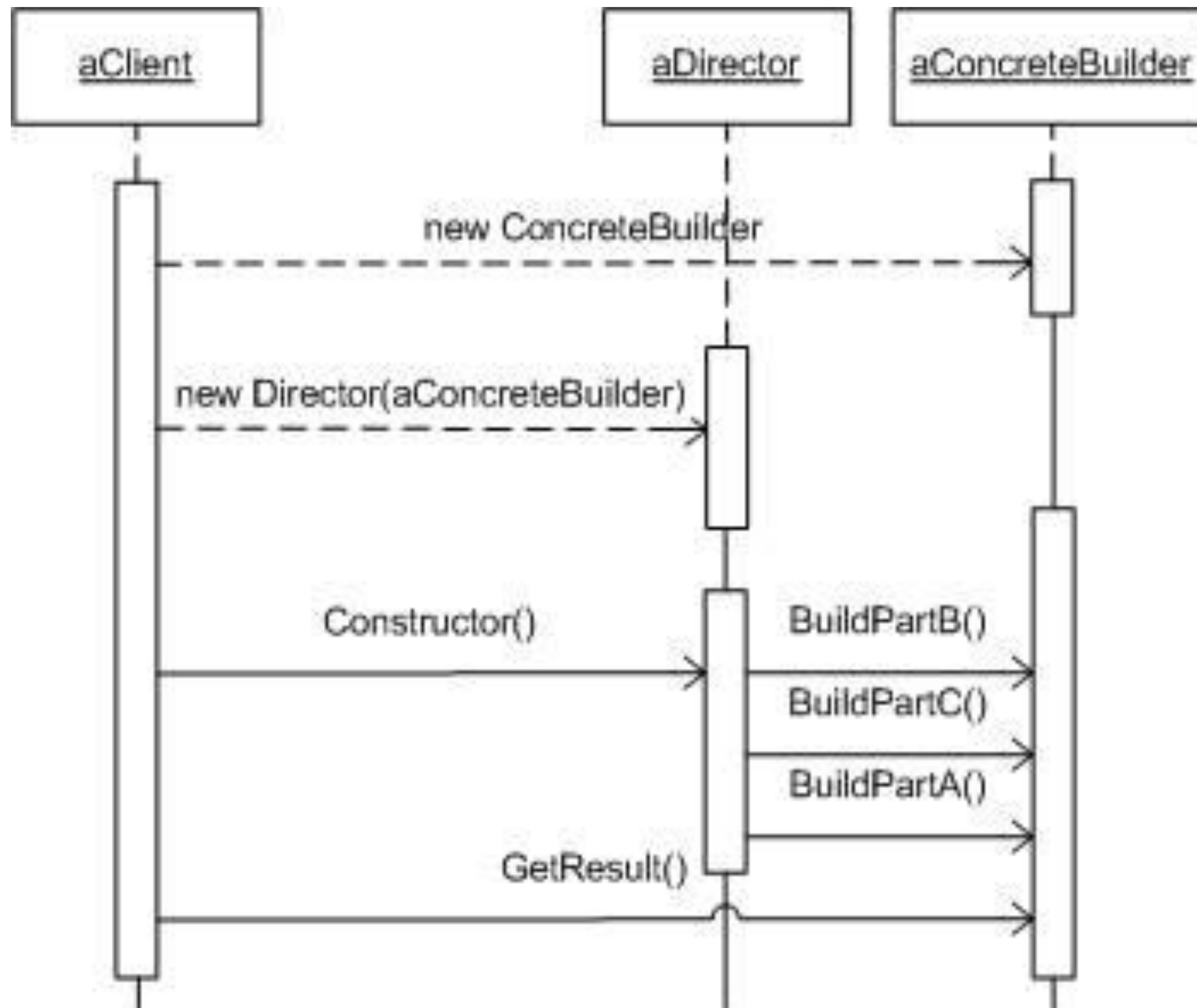
Строитель [Builder]
Участники и отношения:

Отношения:

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

Строитель [Builder]

Отношения:



Строитель [Builder]

Результаты:

- позволяет изменять внутреннее представление продукта. Объект Builder предоставляет распорядителю абстрактный интерфейс для конструирования продукта, за которым он может скрыть представление и внутреннюю структуру продукта, а также процесс его сборки. Поскольку продукт конструируется через абстрактный интерфейс, то для изменения внутреннего представления достаточно всего лишь определить новый вид строителя;
- изолирует код, реализующий конструирование и представление. Паттерн строитель улучшает модульность, инкапсулируя способ конструирования и представления сложного объекта. Клиентам ничего не надо знать о классах, определяющих внутреннюю структуру продукта, они отсутствуют в интерфейсе строителя. Каждый конкретный строитель ConcreteBuilder содержит весь код, необходимый для создания и сборки конкретного вида продукта. Код пишется только один раз, после чего разные распорядители могут использовать его повторно для построения вариантов продукта из одних и тех же частей. В примере с RTF-документом мы могли бы определить загрузчик для формата, отличного от RTF, скажем, SGMLReader, и воспользоваться теми же самыми классами TextConverters для генерирования представлений SGML-документов в виде ASCII-текста, TeX-текста или текстового виджета;

Строитель [Builder]

Результаты:

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

Реализация:

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

Строитель [Builder]

Реализация:

Вот еще некоторые достойные внимания вопросы реализации:

- интерфейс сборки и конструирования. Строители конструируют свои продукты шаг за шагом. Поэтому интерфейс класса Builder должен быть достаточно общим, чтобы обеспечить конструирование при любом виде конкретного строителя. Ключевой вопрос проектирования связан с выбором модели процесса конструирования и сборки. Обычно бывает достаточно модели, в которой результаты выполнения запросов на конструирование просто добавляются к продукту. В примере с RTF-документом строитель преобразует и добавляет очередную лексему к уже конвертированному тексту. Но иногда может потребоваться доступ к частям сконструированного к данному моменту продукта. Другим примером являются древовидные структуры, скажем, деревья синтаксического разбора, которые строятся снизу вверх. В этом случае строитель должен был бы вернуть узлы-потомки распорядителю, который затем передал бы их назад строителю, чтобы тот мог построить родительские узлы.

Строитель [Builder]

Реализация:

- почему нет абстрактного класса для продуктов. В типичном случае продукты, изготавливаемые различными строителями, имеют настолько разные представления, что изобретение для них общего родительского класса ничего не дает. В примере с RTF-документами трудно представить себе общий интерфейс у объектов ASCIIText и TextWidget, да он и не нужен. Поскольку клиент обычно конфигурирует распорядителя подходящим конкретным строителем, то, надо полагать, ему известно, какой именно подкласс класса Builder используется и как нужно обращаться с произведенными продуктами;
- пустые методы класса Builder по умолчанию. В С++ методы строителя намеренно не объявлены чисто виртуальными функциями-членами. Вместо этого они определены как пустые функции, что позволяет подклассу замешать только те операции, в которых он заинтересован.

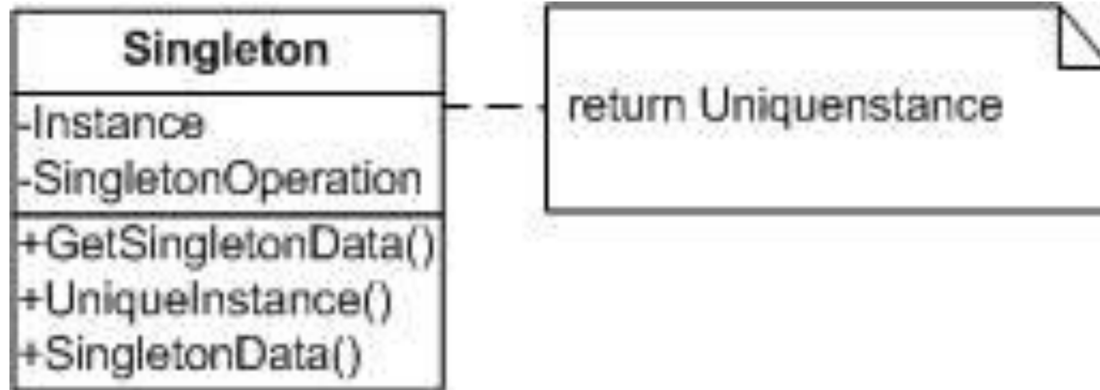
Одиночка [Singleton]
Проблематика:

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

Как гарантировать, что у класса есть единственный экземпляр и что этот экземпляр легко доступен? Глобальная переменная дает доступ к объекту, но не запрещает инициализировать класс в нескольких экземплярах.

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

Одиночка [Singleton]
Проблематика:



- должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам;

- единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

Одиночка [Singleton]
Участники и отношения:

Участники:

- Singleton - одиночка: - определяет операцию Instance, которая позволяет клиентам получать доступ к единственному экземпляру. Instance - это операция класса, то есть метод класса в терминологии Smalltalk и статическая функция-член в C++;

  - может нести ответственность за создание собственного уникального экземпляра.

Отношения:

- Клиенты получают доступ к экземпляру класса singleton только через его операцию

Одиночка [Singleton]
Участники и отношения:

<u>Отношения:</u>

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

Одиночка [Singleton]

Результаты:

У паттерна одиночка есть определенные достоинства:

- контролируемый доступ к единственному экземпляру. Поскольку класс Singleton инкапсулирует свой единственный экземпляр, он полностью контролирует то, как и когда клиенты получают доступ к нему;
- уменьшение числа имен. Паттерн одиночка - шаг вперед по сравнению с глобальными переменными. Он позволяет избежать засорения пространства имен глобальными переменными, в которых хранятся уникальные экземпляры;
- допускает уточнение операций и представления. От класса Singleton можно порождать подклассы, а приложение легко сконфигурировать экземпляром расширенного класса. Можно конкретизировать приложение экземпляром того класса, который необходим во время выполнения;

Одиночка [Singleton]
Результаты:

- допускает переменное число экземпляров. Паттерн позволяет нам легко изменить свое решение и разрешить появление более одного экземпляра класса Singleton. Вы можете применять один и тот же подход для управления числом экземпляров, используемых в приложении. Изменить нужно будет лишь операцию, дающую доступ к экземпляру класса singleton;
- большая гибкость, чем у операций класса. Еще один способ реализовать функциональность одиночки - использовать операции класса, то есть статические функции-члены в C++ и методы класса в Smalltalk. Но оба этих приема препятствуют изменению дизайна, если потребуется разрешить наличие нескольких экземпляров класса. Кроме того, статические функции-члены в C++ не могут быть виртуальными, так что их нельзя полиморфно заместить в подклассах.

Одиночка [Singleton]

Реализация:

- гарантирование единственного экземпляра. Паттерн одиночка устроен так, что тот единственный экземпляр, который имеется у класса, - самый обычный, но больше одного экземпляра создать не удастся. Чаще всего для этого прячут операцию, создающую экземпляры, за операцией класса (то есть за статической функцией-членом или методом класса), которая гарантирует создание не более одного экземпляра. Данная операция имеет доступ к переменной, где хранится уникальный экземпляр, и гарантирует инициализацию неременной этим экземпляром перед возвратом ее клиенту. При таком подходе можно не сомневаться, что одиночка будет создан и инициализирован перед первым использованием.
- порождение подклассов Singleton. Основной вопрос не столько в том, как определить подкласс, а в том, как сделать, чтобы клиенты могли использовать его единственный экземпляр. По существу, переменная, ссылающаяся на экземпляр одиночки, должна инициализироваться вместе с экземпляром подкласса. Простейший способ добиться этого - определить одиночку, которого нужно применять в операции Instance класса Singleton.

Обсуждение порождающих паттернов
Задача: параметризация системы классами объектов.

- Первый способ: порождение подклассов от класса, создающего объекты. Это, по сути, фабричный метод. Недостаток – требуется порождать новый подкласс только для того, чтобы сменить класс продукта.
- Второй способ: композиция объектов. Определяется объект, которому известно о классах объектах-продуктах и этот объект делается параметром системы. Это основная идея трех паттернов: абстрактная фабрика, строитель и прототип.