Factory Method Design Pattern

„Definiere eine Klassenschnittstelle zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, welche Klasse instanziiert werden soll. Factory-Methods ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.“

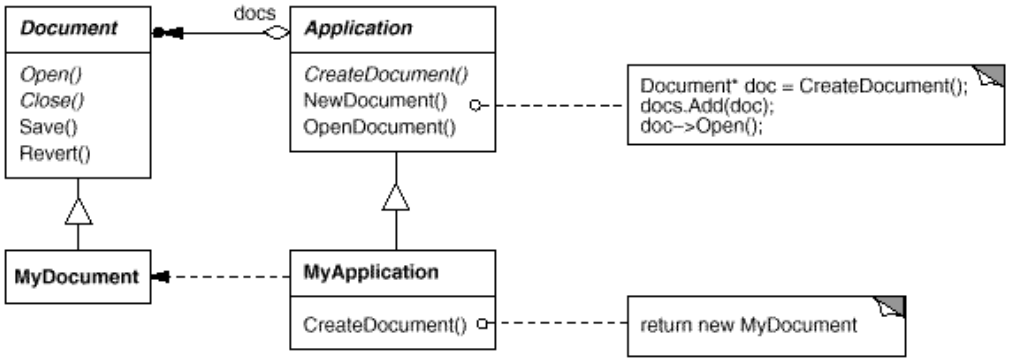
Motivation, Verwendungsbeispiele aus der Praxis

Frameworks verwenden abstrakte Klassen zum Definieren und Verwalten von Beziehungen zwischen Objekten. Außerdem sind diese Frameworks auch oft für die Erstellung von diesen Objekten verantwortlich.

Man nehme ein Framework für Anwendungen, welche verschiedene Dokumente anzeigen können. Zwei wichtige Abstraktionen in diesem Framework sind die Klassen „Application“ und „Document“. Beide Klassen sind abstrakt und die Clients müssen Subklassen dafür erstellen, um die spezifischen Applikationen zu implementieren. Wenn man beispielsweise eine Zeichenapplikation erstellen möchte, muss man eine „DrawingApplication“-Klasse und eine „DrawingDocument“-Klasse erstellen. Die „Application“-Klasse ist für die Verwaltung von Dokumenten verantwortlich und erstellt sie nach Bedarf, wenn der Benutzer beispielsweise in einem Menü die Option „Öffnen“ oder „Neu“ wählt.

Da die bestimmte Dokument-Unterklasse, die instanziiert werden soll, anwendungsspezifisch ist, kann die Anwendungsklasse die Unterklasse des Dokuments nicht für die Instanziierung vorhersagen - die Anwendungsklasse weiß nur, wann ein neues Dokument erstellt werden soll, nicht, welche Art von Dokument erstellt werden soll. Dies führt zu einem Problem: Das Framework muss Klassen instanziieren, kennt aber nur abstrakte Klassen, die es nicht instanziieren kann.

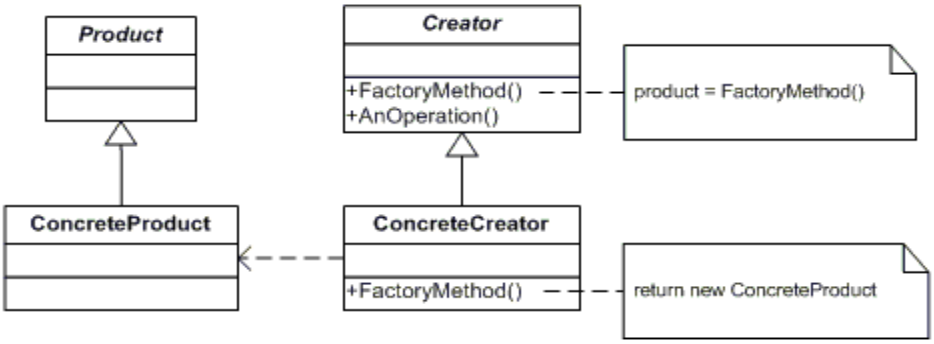
Das Factory-Method-Muster bietet eine Lösung. Es kapselt das Wissen, welche „Document“-Unterklasse erstellt werden soll, und verschiebt dieses Wissen aus dem Framework.



Anwendungsunterklassen definieren einen abstrakten „CreateDocument“-Vorgang in „Application“ neu, um die entsprechende Dokumentunterklasse zurückzugeben. Sobald eine Anwendungsunterklasse instanziiert ist, kann sie anwendungsspezifische Dokumente instanziieren, ohne deren Klasse zu kennen. „CreateDocument“ wird Factory-Methode genannt, weil sie für das "Herstellen" eines Objekts verantwortlich ist.

Zusammenfassend kann gesagt werden, dass das Factory Method Design Pattern zur Entkopplung des Clients von der konkreten Instanziierung einer Klasse dient. Der „Erstellungscode“ eines Objekts wird in eigene Klassen (Factory) ausgelagert. Dadurch wird die Trennung von Objektverwaltung und Objektherstellung realisiert. Dies bedeutet, man muss keine anwendungsspezifischen Klassen in den Code einbauen. Der Code behandelt nur die Schnittstellen.

UML-Diagramm



Komponenten

- Product (Document)
  - Definiert die Schnittstelle der Objekte, die die Factory-Methode erstellt
- ConcreteProduct (MyDocument)
  - implementiert die Produktschnittstelle
- Creator (Application)
  - deklariert die Factory-Methode, die ein Objekt vom Typ „Product“ zurückgibt *(Ersteller kann auch eine Standardimplementierung der Factory-Methode definieren, die ein „ConcreteProduct“-Standardobjekt zurückgibt)*
  - kann die Factory-Methode aufrufen, um ein Produktobjekt zu erstellen
- ConcreteCreator (MyApplication)
  - überschreibt die Factory-Methode, um eine Instanz eines „ConcreteProduct“ zurückzugeben

Anwendungsfälle

- Trennung der Objektverarbeitung (Wie?) von der konkreten Objekterstellung (Instanziierung; Was?). Delegation der Objektinstanziierung an Unterklasse.
  - Authentifizierungssysteme: Für jeden User wird nach dem Login ein Ticket erstellt, das seine Rechte im System repräsentiert. Statt eine universelle Ticketklasse mit duzenden, je nach Userrechten gesetzten Attributen zu nutzen, werden spezielle Tickets erstellt. Dies geschieht mittels einer TicketFactory und einer createTicket()-Methode, die mit den nötigen Informationen parametrisiert wird. Anhand dieser Parameter wird entschieden, welches Ticket erstellt wird. Verschiedene Erzeugungsprozesse werden durch mehrfaches Ableiten dieser Factory ermöglicht (InternetTicketFactory, IntranetTicketFactory). ([PK], Seite 27f.)
- Fälle, in denen mit einer wachsenden Anzahl und Ausformung von Produkten zu rechnen ist. Sowie Szenarien, in denen alle Produkte einen allgemeinen Herstellungsprozess durchlaufen müssen, egal was für ein Produkt sie konkret sind.
  - Eine Pizzeria erstellt je nach Parameter verschiedene Pizzen, die immer gleich zubereitet werden (backen, schneiden, einpacken). Um den regionalen Wünschen der Menschen in Wien, Salzburg und Linz zu genügen, müssen spezielle Pizzen für eben diese Städte erstellt werden. Diese Erstellung geschieht in den speziellen Unterklassen der Pizzeria (SalzburgPizzeria, WienPizzeria, LinzPizzeria), die die speziellen Pizzen erstellen (WienSalami, LinzCalzone etc.). Die Verarbeitung ist in allen Fällen gleich und erfolgt in der abstrakten Pizzeriaklasse, die nur die abstrahierte Pizzaschnittstelle kennt. Schnell können somit neue Pizzasorten und Pizzerias ins System integriert werden.
- Wenn die konkret zu erstellenden Produkte nicht bekannt sind oder nicht von vorne herein festgelegt werden sollen.
  - Frameworks und Klassenbibliotheken.

Vorteile und Nachteile

Vorteile:

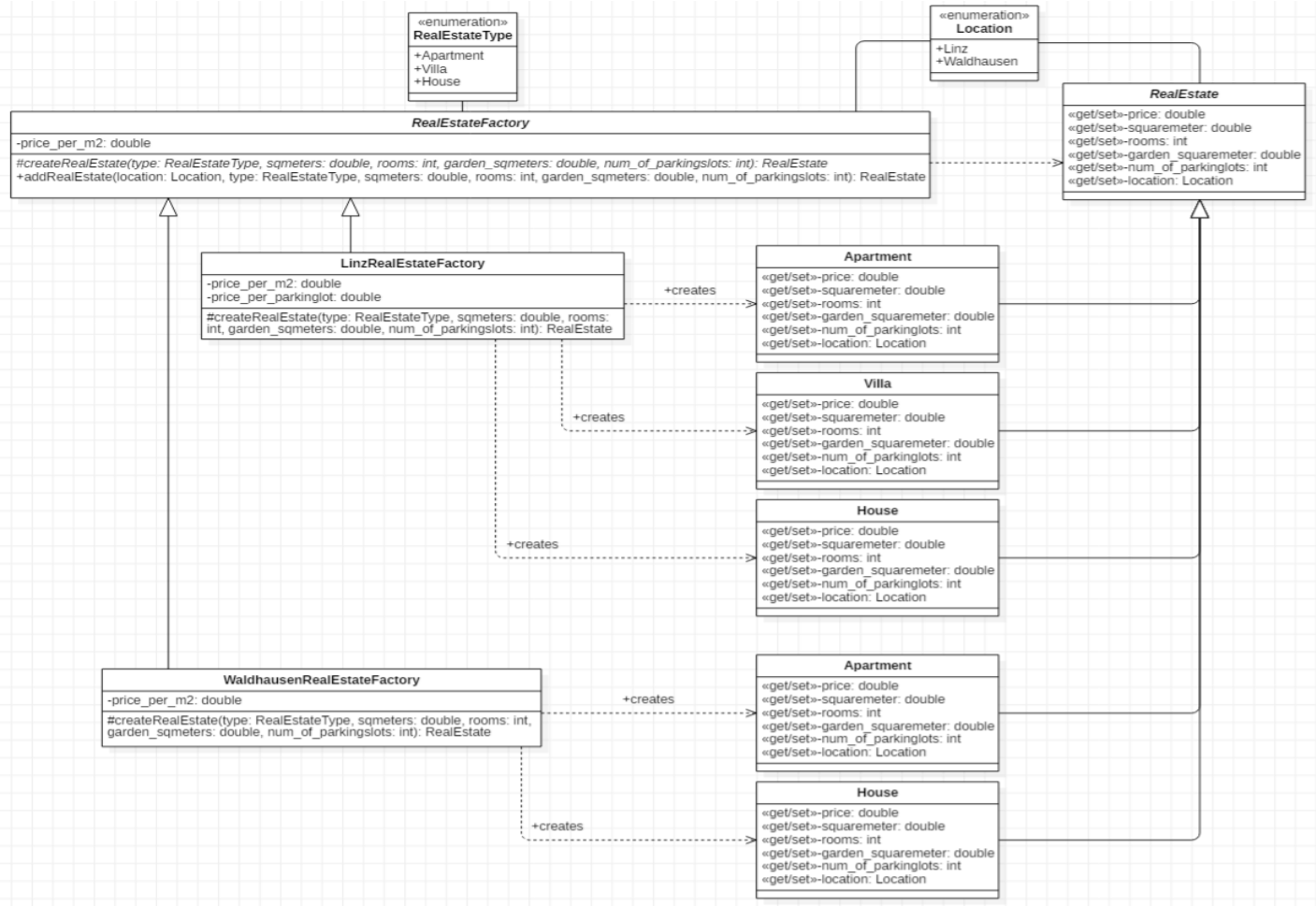
- Herstellungsprozess ist von einer konkreten Implementierung getrennt und unterschiedliche Produktimplementierungen können denselben Produktionsvorgang durchlaufen.
- Wiederverwendbarkeit und Konsistenz durch
  - Kapselung des Objekterstellungscodes in eigener Klasse. Dadurch entsteht eine einheitliche und zentrale Schnittstelle für den Client. Die Produktimplementierung ist von seiner Verwendung entkoppelt. Außerdem entsteht ein zentraler Punkt der Wartung (geringerer Änderungsaufwand).
  - Kapselung des allgemeinen Herstellungscodes in die Superklasse Creator, die auf jedem Produkt vor Auslieferung an den Client durchgeführt wird.
- Erweiterbarkeit, Austauschbarkeit und Flexibilität durch problemlose Einführung neuer Products und ConcreteCreators (bzw. Modifizierung oder Auswechseln bestehender) ohne Brechen des Clientcodes, dank konsistenter Schnittstelle. So kann ein neuer ConcreteCreator in das System integriert werden und dabei bestehenden Creatorcode wiederverwenden ohne Änderungen am bestehenden Client nötig zu machen.

Nachteile:

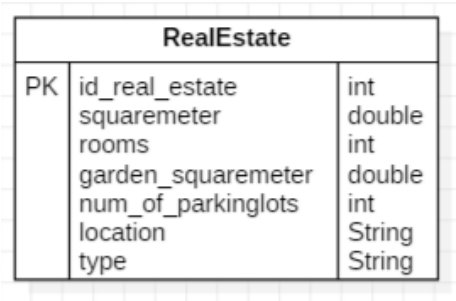
- Enge Kopplung eines konkreten Creators an ein konkretes Produkt. Für jedes neue Produkt muss ein neuer ConcreteCreator geschrieben werden und den abstrakte Creator ableiten. Unser Gesamtsystem hat damit nur wegen der Fabrikmethode einen weiteren Evolutionsast. Bei der parametrisierten Variante des Patterns ist dies weniger problematisch, da oft nur ein bestehender konkreter Creator angepasst werden muss.

Beispiel: Screenshots und Erklärung

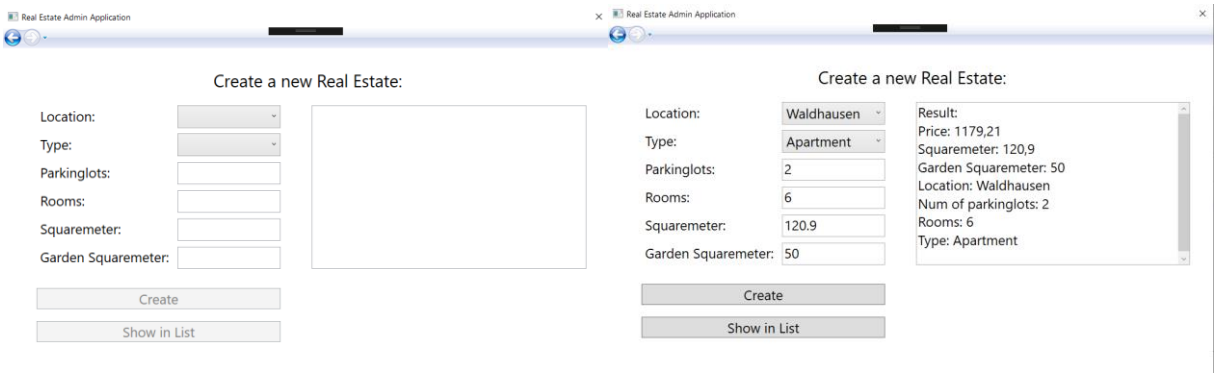
Dieses Beispiel dient zur Verwaltung von Immobilien. Es gibt drei Arten von Immobilien: Apartment, House und Villa. Jeder Ort (Factory) erstellt die Immobilien nach den aktuellen Preisen. Neue Orte können einfach hinzugefügt werden, in dem die Factory-Klasse abgeleitet wird.



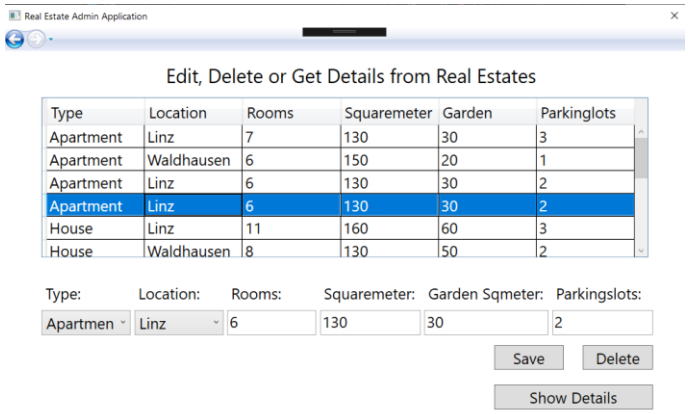
Das Backend in Form einer REST-Schnittstelle wurde mit Java realisiert. Diese verwaltet die Properties, die zur Erstellung der RealEstates notwendig sind. In der folgenden Grafik ist das Datenmodell ersichtlich:



Die Admin-Anwendung wurde mit WPF erstellt. Es bietet die Möglichkeit, dynamisch RealEstate Objekte zu erstellen und diese durch die REST-Schnittstelle in der Datenbank zu speichern.



Die erstellten Objekte können in einer Listenansicht kontrolliert werden. Dabei besteht auch die Möglichkeit, die Datensätze zu ändern bzw. zu löschen. Beim Klick auf einen Eintrag können Details angezeigt werden. Beim Aufruf der Detail-Seite wird das RealEstate-Objekt wieder anhand der Factory Methode erstellt und der Preis somit berechnet.



Type: Apartment

Location: Linz

Rooms: 6

Squaremeter: 130 m2

Garden Squaremeter: 30 m2

Parkingslots: 2

Price: 1784 €

Die Client-Anwendung basiert auf einer Angular Applikation, welche die Datensätze aus der Datenbank ladet und die RealEstate-Objekte auf einer Website anzeigt. Dabei weiß die Anwendung nicht, mit welchen konkreten Produkten sie es zu tun hat.

Quelle:

Design Patterns: https://bit.ly/2LJEUCl

UML-Diagramm: https://bit.ly/2SDJQvD

Beispiele, Vor-/Nachteile: https://bit.ly/2QRgw7X