

OBJECT-ORIENTED PROGRAMMING

Creational Design Patterns

Lecture #10

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2023/2024

Creational design patterns

- Abstract the instantiation process
- Plan for today
 - Factory Method
 - Abstract Factory
 - Singleton
 - Builder
 - Prototype

Creational design patterns

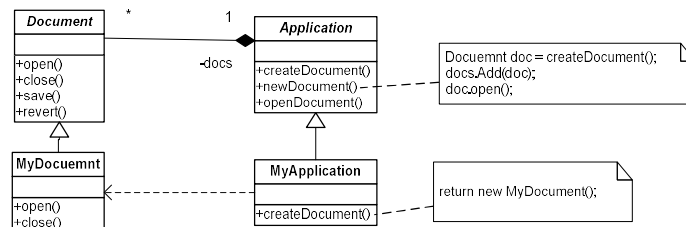
- **Creational patterns** abstract the instantiation process
 - They control (hide) creation of objects and helps build systems independent from creation and building of objects
- **Creational patterns of classes** use inheritance to decide what objects should be created
 - **Factory Method, Prototype**
- **Creational patterns of objects** delegates creation of objects to other objects
 - **Abstract Factory, Builder**

Creational design patterns

- All OO languages include **new** command for an object instantiation
- Creational patters allows us to write methods for objects instantiation without direct usage of **new**
- Methods can create various objects and can be overridden/overloaded to create other new objects without modification of code

Factory Method

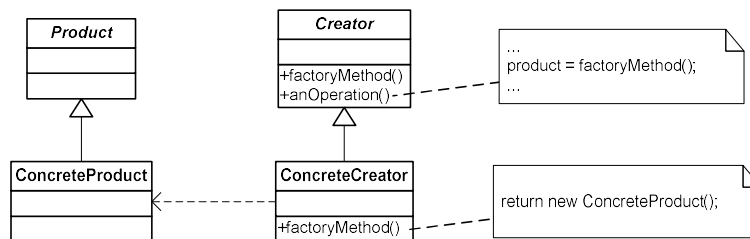
- Purpose
 - Defines interface for creating objects and allows subclasses to control the type of created objects
- Motivation



- Method `createDocument()` represents **Factory Method**

Factory Method

- Usage
 - In case the class cannot assume what type of creating objects is used
 - Subclasses must specify the creation of objects
- Structure



Lecture #10: Creational Design Patterns

7

Factory Method

- Participating parts
 - Product – defines interface for object types that are created
 - ConcreteProduct – implements Product interface
 - Creator – declares factory method, which returns object of type Product
 - ConcreteCreator – overrides factory method and returns instance of ConcreteProduct
- Collaboration of parts
 - Creator depends on its subclasses that implements factory method returning adequate instance of ConcreteProduct

Lecture #10: Creational Design Patterns

8

Factory Method

- Class `Creator` is created without knowledge what `ConcreteProduct` is instantiated. What `ConcreteProduct` is instantiated is given by corresponding `ConcreteCreator` subclass used in application
- It does not mean the subclass decides during runtime what concrete type is used!

Factory Method

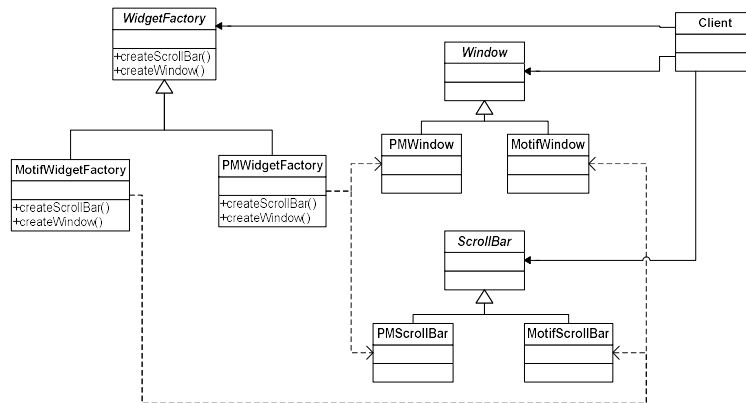
- Advantages
 - Code is much more flexible and reusable thanks to avoidance of direct instantiating of application specific objects
 - Code works only with `Product` interface and can use any classes `ConcreteProduct`, which implement this interface
- Implementation
 - `Creator` can be either abstract or concrete class
 - When factory method creates more types of objects, for example input parameter distinct among them by `if-else` construction

Abstract Factory

- Purpose
 - An interface of creating a group of related or dependent objects without specifying their concrete classes
 - Creation of objects is delegated to other objects using composition and Factory Method is used for creation of concrete objects

Abstract Factory

- Motivation

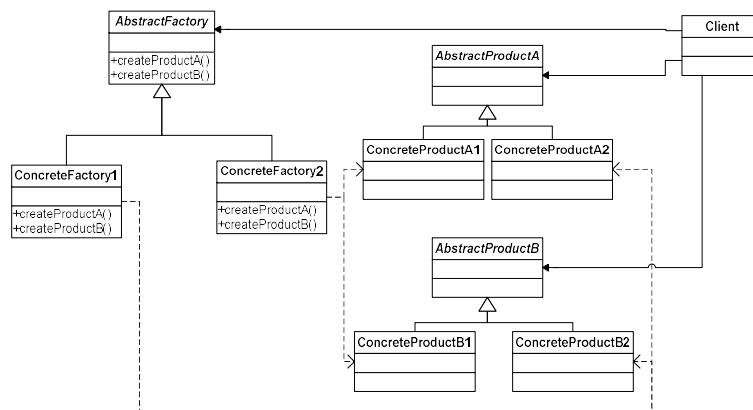


Abstract Factory

- Usage
 - System should be independent of the way how its objects are created
 - Class cannot assume what classes are used for creating objects
 - System must use only one group of related objects
 - Objects of one group must be used together

Abstract Factory

- Structure



Abstract Factory

- Participating parts
 - **AbstractFactory** – interface declaring operations for creating objects of abstract products
 - **ConcreteFactory** – implements operations for creating concrete objects
 - **AbstractProduct** – interface of a product
 - **ConcreteProduct** – defines concrete product that is created by concrete factory method; implements interface **AbstractProduct**
 - **Client** – uses only interfaces declared as abstract (**AbstractFactory**, **AbstractProduct**)

Abstract Factory

- Collaboration of parts
 - Only single instance of **ConcreteFactory** is created (see pattern Singleton) and it creates objects according concrete implementation. For creting other types of products there must be another instance of other **ConcreteFactory**
 - Class **AbstractFactory** depends on implementation of factory methods by its subclasses

Abstract Factory

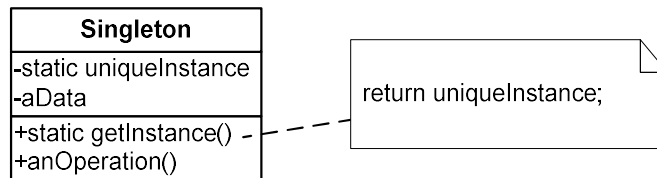
- Advantages
 - Isolates client from concrete implementation of classes
 - Allows simple switch of product groups, because concrete factory class implements creation of whole product group
 - Enforces usage of one product group
- Implementation
 - Typical implementation needs only single instance of factory class
 - In this case pattern Singleton is used
 - In case concrete factory class must create new types of products (not included in abstract declaration), only one factory method (with parameter) is implemented

Singleton

- Purpose
 - Allows creation single instance of a given class and allows global access to it
- Motivation
 - Sometimes we need just one single instance of the class
 - For example, I want a single object to control and manage objects of windows in the system
 - We need an easy access to the object
 - To ensure it cannot be created more than one objects of the class

Singleton

- Structure



- Advantages

- Controlled access to an explicit object
- Does not allow to create other instances

Builder

- Purpose

- Separates construction (composition) process of a complex object from its representation (structure)

- Motivation

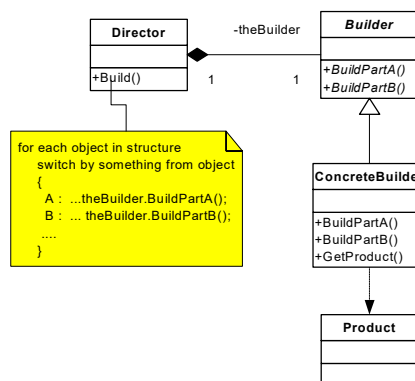
- Situations in which we need to compose object from other objects
- We want to hide before client the implementation of object composition
- We want to support client by variability of composition of object structure

Builder

- Usage
 - Separate problem of object composition (from what to build) from its construction (how to build) and from object representation (what to build)
 - Client accesses object construction by interface that is implemented by various concrete objects

Builder

- Structure

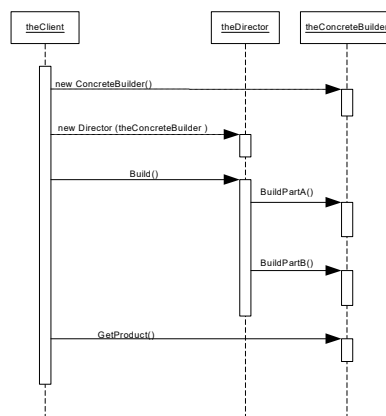


Builder

- Participating parts
 - **Builder** – abstract interface for creating parts of an object
 - **ConcreteBuilder** – implements **Builder** and constructs concrete type of **Product** (executes whole construction)
 - **Director** – directs construction of structured object by builder object implementing **Builder** (i.e. **ConcreteBuilder**); it does not know the structure
 - **Product** – represents building structured objects

Builder

- Collaboration of parts



Prototype

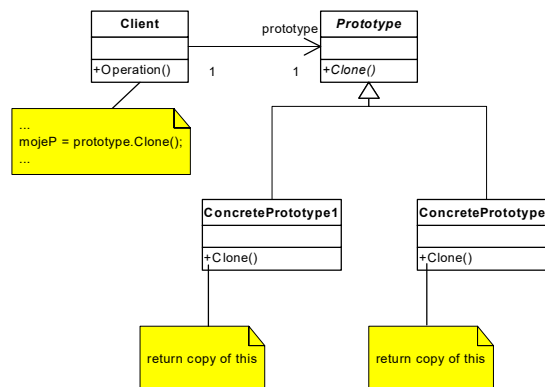
- Purpose
 - Objects are created by cloning prototype instances defining a single interface
- Motivation
 - When creating objects, we often cannot choose the type of the instance
 - Usage of Factory patterns requires rewriting whole factory method after inheritance
 - Creation of objects as copies of other objects without knowing their actual classes

Prototype

- Usage
 - Structure of products has common interface that supports cloning objects
 - Client contains list of prototype objects used for cloning new objects
 - Client access objects (to clone them) only using common interface without knowing actual classes of them

Prototype

- Structure



Prototype

- Participating parts

- **Prototype** – abstract interface for creating objects, declares abstract method `Clone()` of object cloning
- **ConcretePrototype** – implements **Prototype** and creates copy of this concrete instance by rewriting `Clone()`
- **Client** – uses **Prototype** for creating new objects by method `Clone()`, it does not need to know what **ConcretePrototype** is used (sometimes called **Prototype Manager** and often implemented by various **Factory** patterns)