Software Engineering

Lesson Design Pattern 01
Factory Method, Abstract Factory, Builder
v1.0b

Uwe Gühl

Winter 2007/ 2008

Contents

- Factory Method
- Abstract Factory
- Builder
- Difference between Abstract Factory and Builder

Factory Method

- Intent: Define an interface for creating an object, but let subclasses decide which class to instantiate.
  Factory Method lets a class defer instantiation to subclasses.
  - known as “Virtual Constructor” as well
- Motivation
  - Frameworks use abstract classes to define and manage relationships between objects
  - A framework is also responsible to create such objects

Factory Method

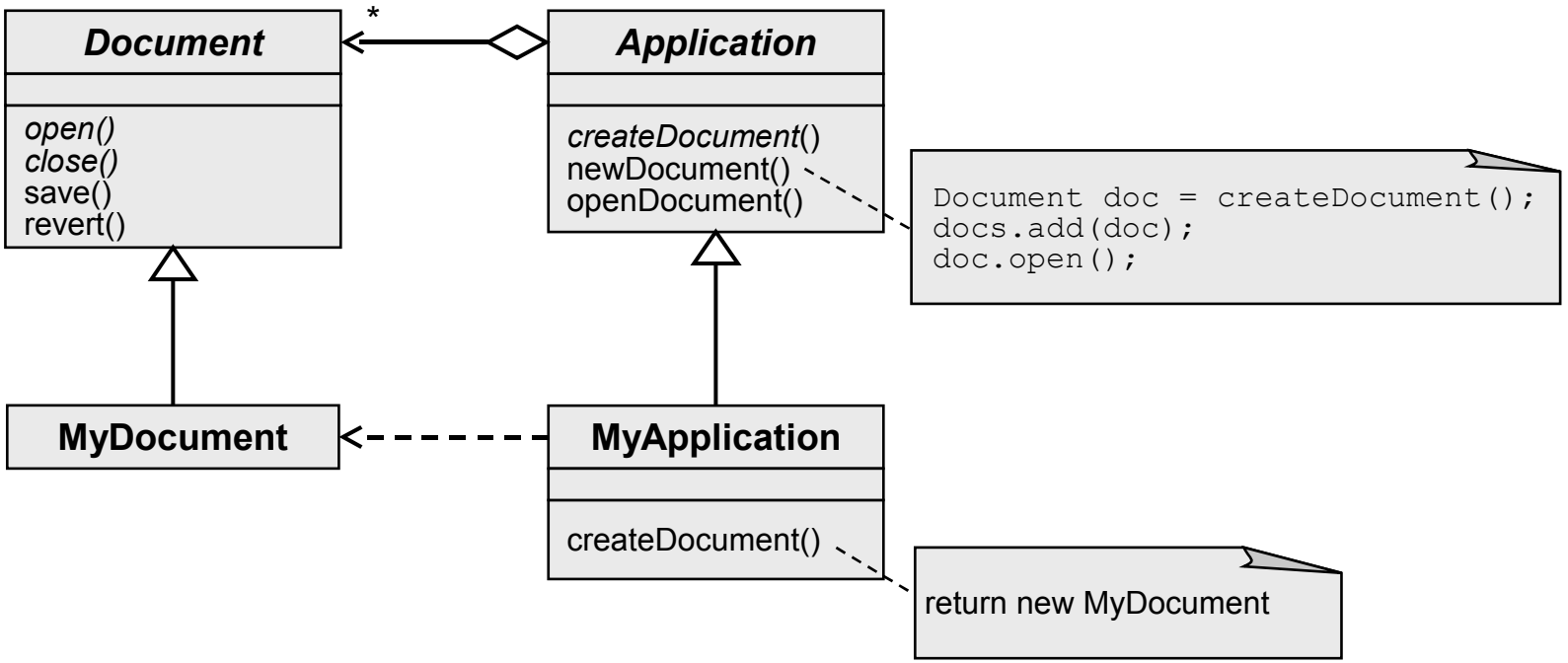
- Introduction
  - Application framework with two kinds of documents resulting in two key abstractions
    - ***Application***
    - ***Document***
  - Clients work with concrete subclasses to realize client specific implementations
    - A Drawing application would need
      - DrawingApplication
      - DrawingDocument

Factory Method

- Introduction
  - The application is responsible for the documents and creates them on demand (e. g. with calling „Open“ or „New“)
  - Problem: Application knows when, but not which kind of subclass of ***Document*** to instantiate
    - Framework has to instantiate classes but knows only abstract classed
  - Idea: Encapsulation of the Know-how, which Document subclass to instantiate, out of the framework

Factory Method

- Introduction

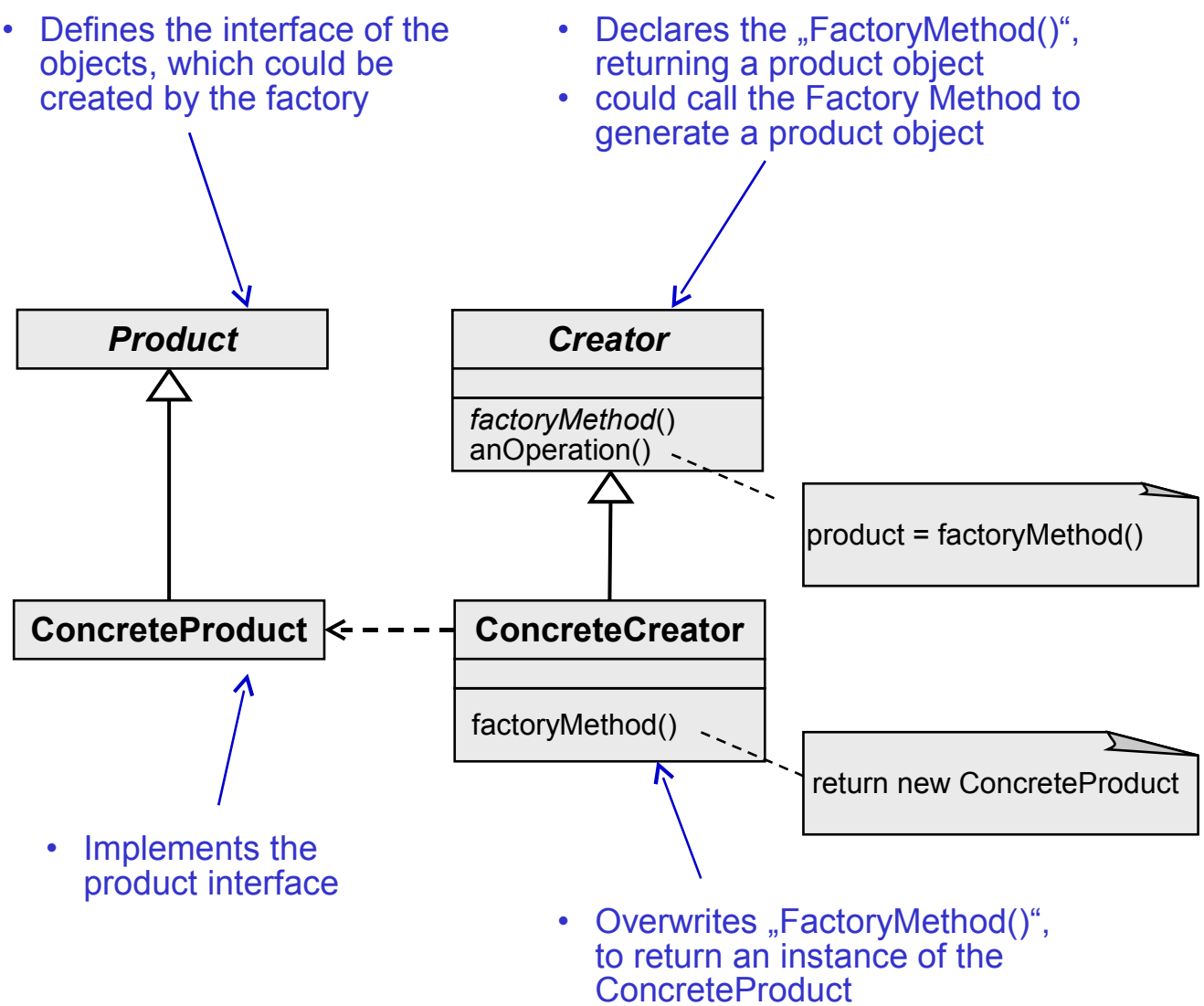


Factory Method

- Introduction
  - Subclasses of the ***Application*** overwrite the method CreateDocument, to return the fitting ***Document***-Subclass
  - As soon the subclass of the ***Application*** is instantiated, this subclass could instantiate the application specific documents, and it has nothing to know about this class.
  - „CreateDocument“ is a „**FactoryMethod**“, because it is responsible for the creation of an object

Factory Method

- Structure



Factory Method

- Collaboration
  - ***Creator*** relies on its subclasses to define the factory method so that it returns an instance of the appropriate **ConcreteProduct**

Factory Method

- Consequences

+ Application specific classes don't have to bind in code, communication is only with the ***Product*** interface necessary

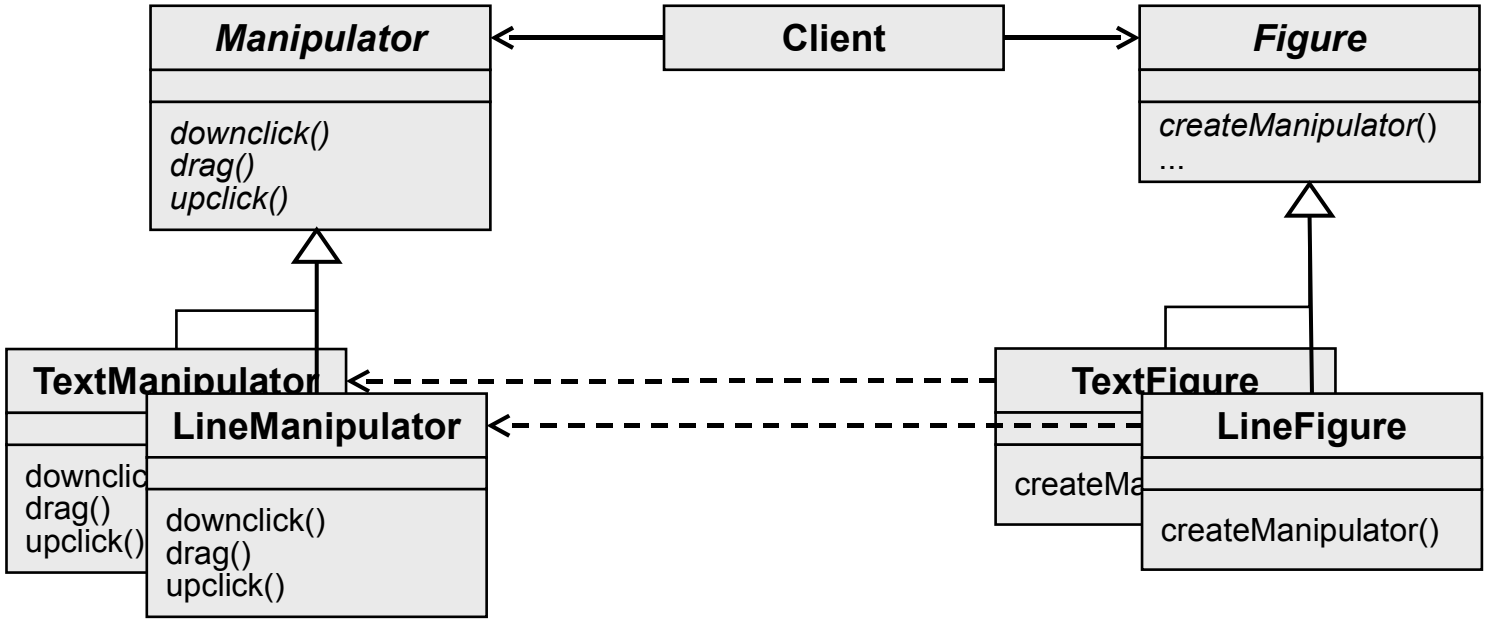
– Maybe clients need subclasses of the ***Creator*** class only to be able to create a specific **ConcreteProduct** object

Factory Method

- Consequences
  - Offers a „hook“ for subclasses
    Factory method offers for subclasses a connection to make extended versions of an object possible
    Example:
    - ***Document*** defines a factory method „CreateFileDialog“
    - **MyDocument** defines an application specific „CreateFileDialog“

Factory Method

- Consequences
  - Could connect parallel class hierarchies
    Example: Delegation of object manipulation of graphical figures



Factory Method

- Implementation
  - Two philosophies in the use of the factory method pattern:
    - ***Creator*** offers as abstract class no implementation of the factory method, that it declares
      → Subclasses are necessary to define an implementation
    - **Creator** is a concrete class and offers a standard implementation for the factory method (Abstract classes with standard implementation are unusual)
      → Use of the factory method because of flexibility reasons

Factory Method

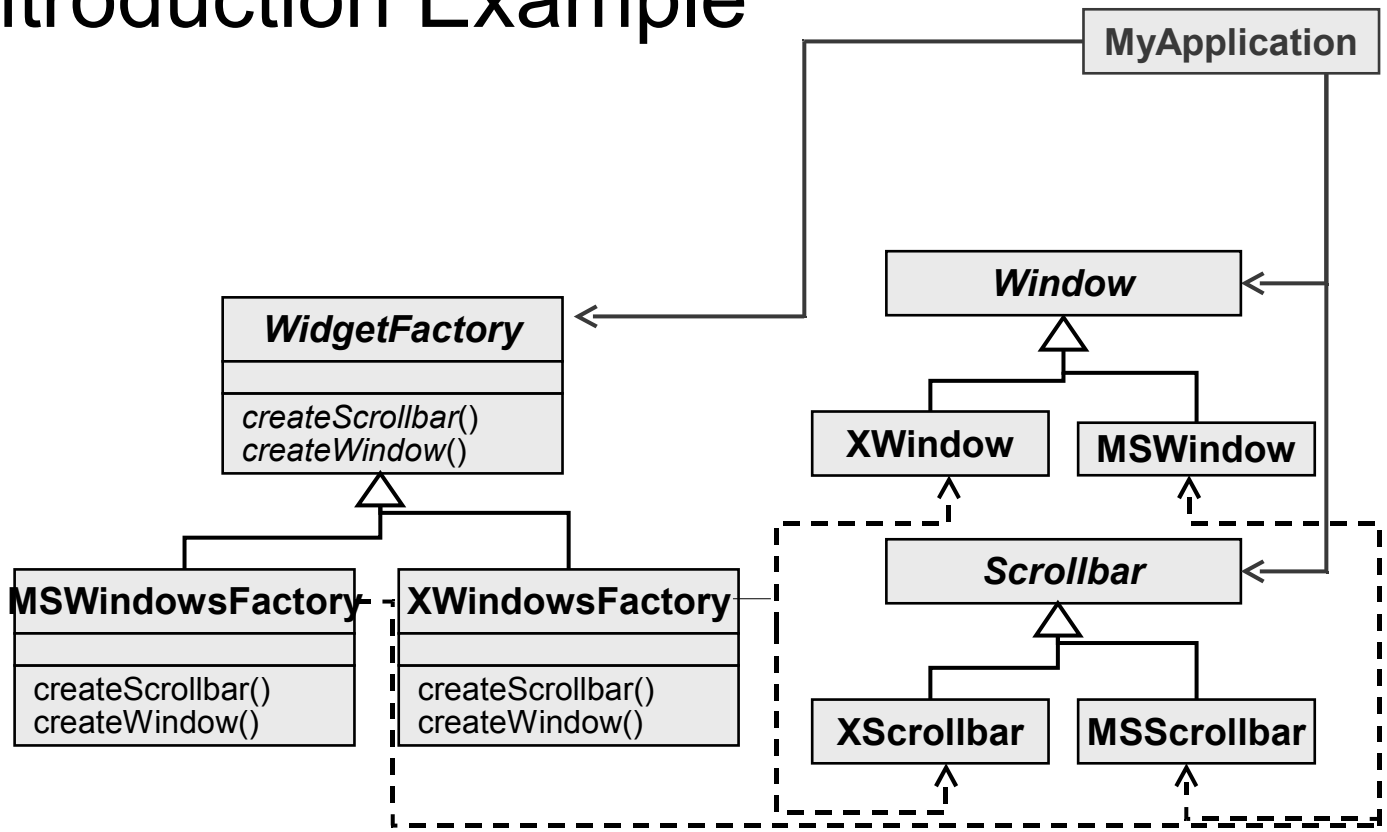
- Implementation
  - Parameterized factory methods
    The factory method generates depending on delivered parameter different kinds of ***Product***s.
    Example out of the Unidraw graphical editing framework:
    - ***Creator*** with factory method Create(productId)
    - ProductId specifies the class to create
    - Save: (1) Write productId
      (2) Write instance variables
    - Read: (1) Read productId
      (2) Framework calls create(productId) → constructor
      (3) Call of a read method for instance variables

Abstract Factory

- Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- Motivation
  - GUI for different Look and Feel standards
  - Idea
    - abstract WidgetFactory class, offering an interface for every kind of widget
    - Subclasses implement widgets
    - Return values of corresponding operations are widget objects

Abstract Factory

- Introduction Example



Abstract Factory

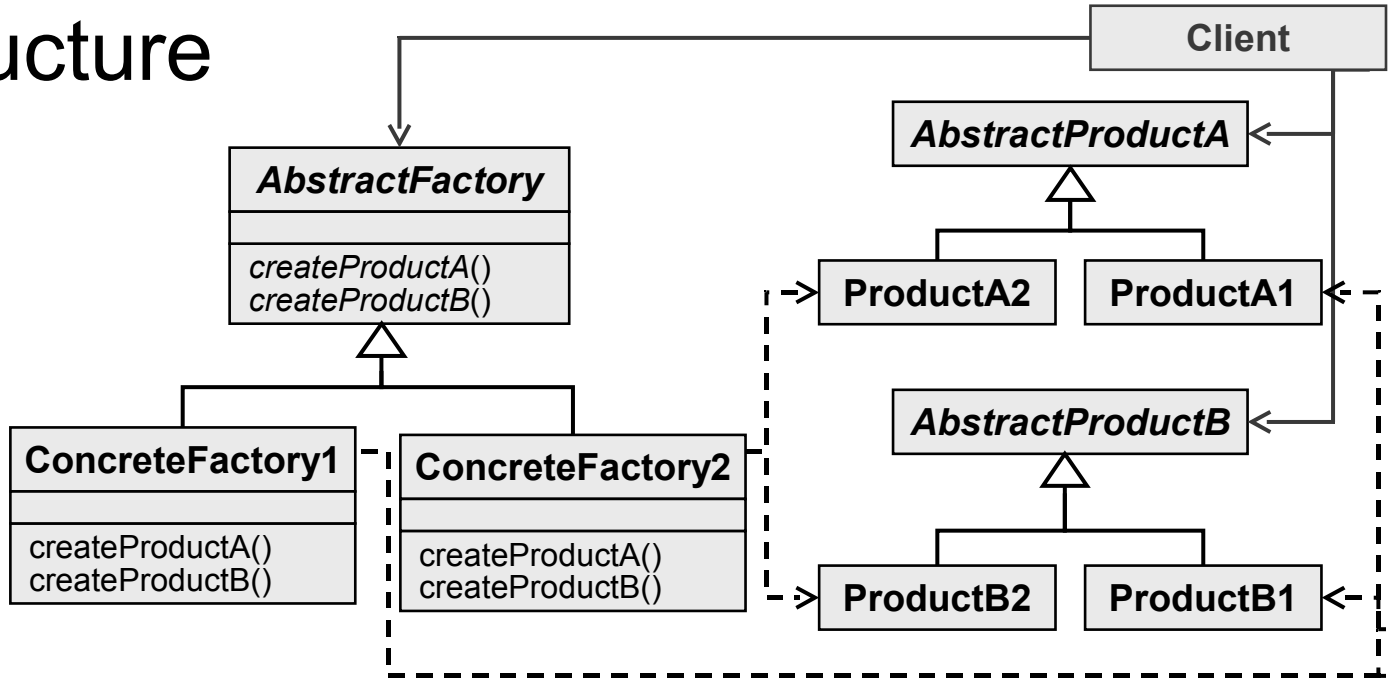
- Introduction Example Code

The **only** place, where the concrete implementation class is named. Replacing ‚X‘ with ‚MS‘ would change all window elements automatically to MS-Windows Look&Feel.

```
public class MyApplication {
  public Window buildWindow() {
    Window myWindow;
    WidgetFactory factory = new XWindowsFactory();
    window = factory.createWindow();
    window.addScrollbar(factory. createScrollbar());
    return window;
  }
}
```

Abstract Factory

- Structure



AbstractFactory declares the interface to create abstract product objects.

ConcreteFactory implements the operations to generate concrete products

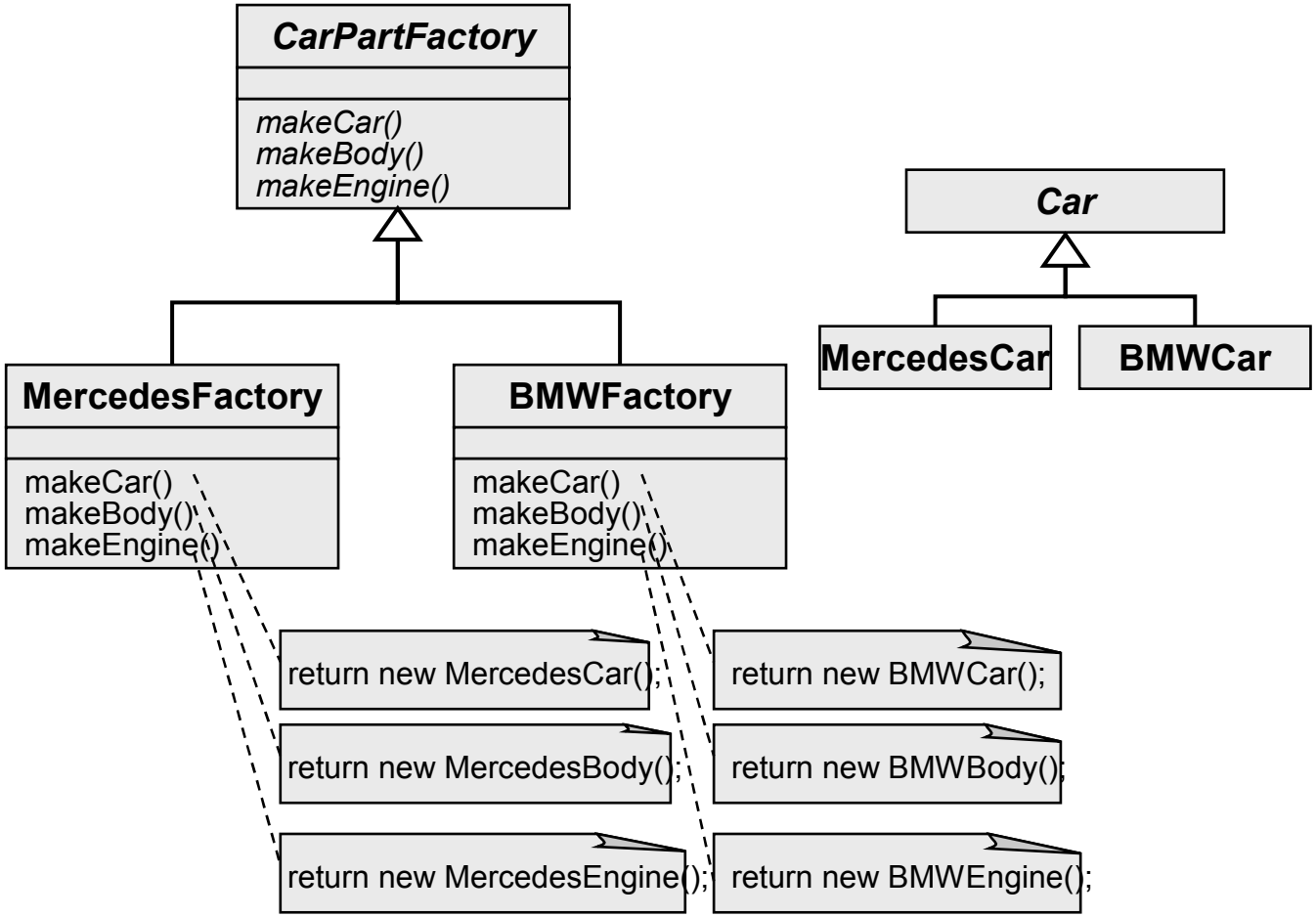
AbstractProduct declares the common interface for a product

ConcreteProduct implements the ***AbstractProduct*** interface for a concrete product

Client uses only the interfaces declared by ***AbstractFactory*** and ***AbstractProduct***(s)

Abstract Factory

- Example



Abstract Factory

- Example – Code

```
public class CarAssembler {
   public Car assembleCar() {
      Car car;
      Factory factory = new MercedesFactory();
      car = factory.makeCar();
      car.addEngine(factory.makeEngine());
      car.addBody(factory.makeBody());
      . . .
      return car;
   }
}
```

Abstract Factory

- Consequences

+ Concrete classes are isolated
  The names of the product classes do not appear in client code
+ Exchanging of product families easy
  The name of the concrete factory appears only once in the application – where it' instantiated

Abstract Factory

- Consequences

+ Consistency of products gets supported:
  The creation of products with the factory avoids, that a client creates products from different families at the same time by accident
  (Example: XWindow with MSScrollbar or MercedesCar with BMWEngine).
– Support of new products (Car respectively Window subclasses) is complex
  The interface of the abstract factory has to be adapted
  → The abstract factory fixes the set of products which could be generated

Abstract Factory

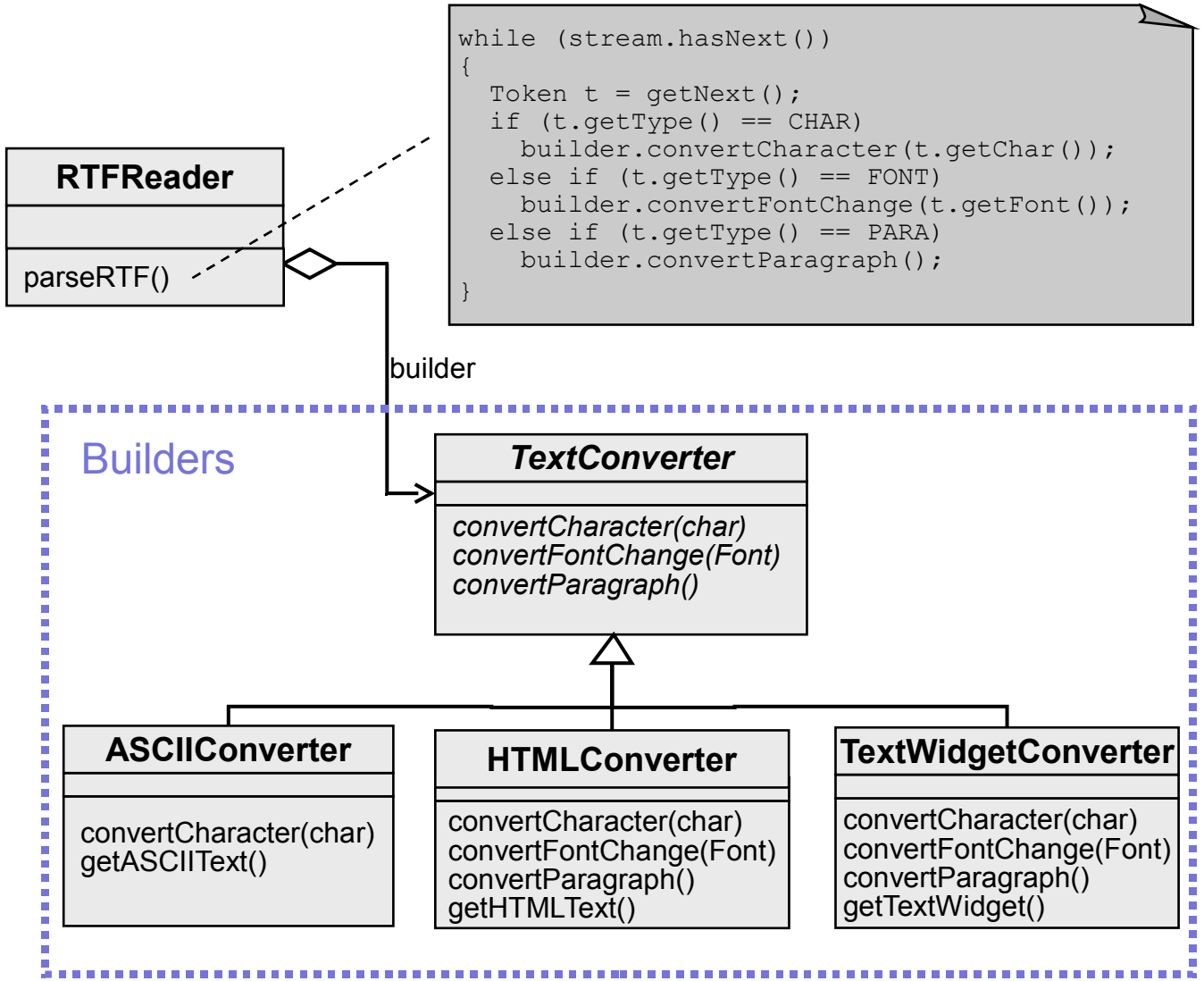
- Implementation
  - Factories as Singletons
    - If an application needs only one instance of a ConcreteFactory
  - Generation of products
    - ConcreteFactory uses therefore often patterns
      - FactoryMethod
        with product depending overriding
      - Prototype
        Instead of using (many) subclasses, ConcreteFactory gets initialized with a prototypical instance

Builder

- Intent: Separate the construction of a complex product from its representation so that the same construction process can create different representations
- Motivation
  - Example: Parsing of a document in RTF (Rich Text Format) and converting in another format (ASCII, HTML or a GUI-Text-Widget) – the number of possible conversions is not limited.
  - Idea
    - Parser uses a textConverter object to perform conversion

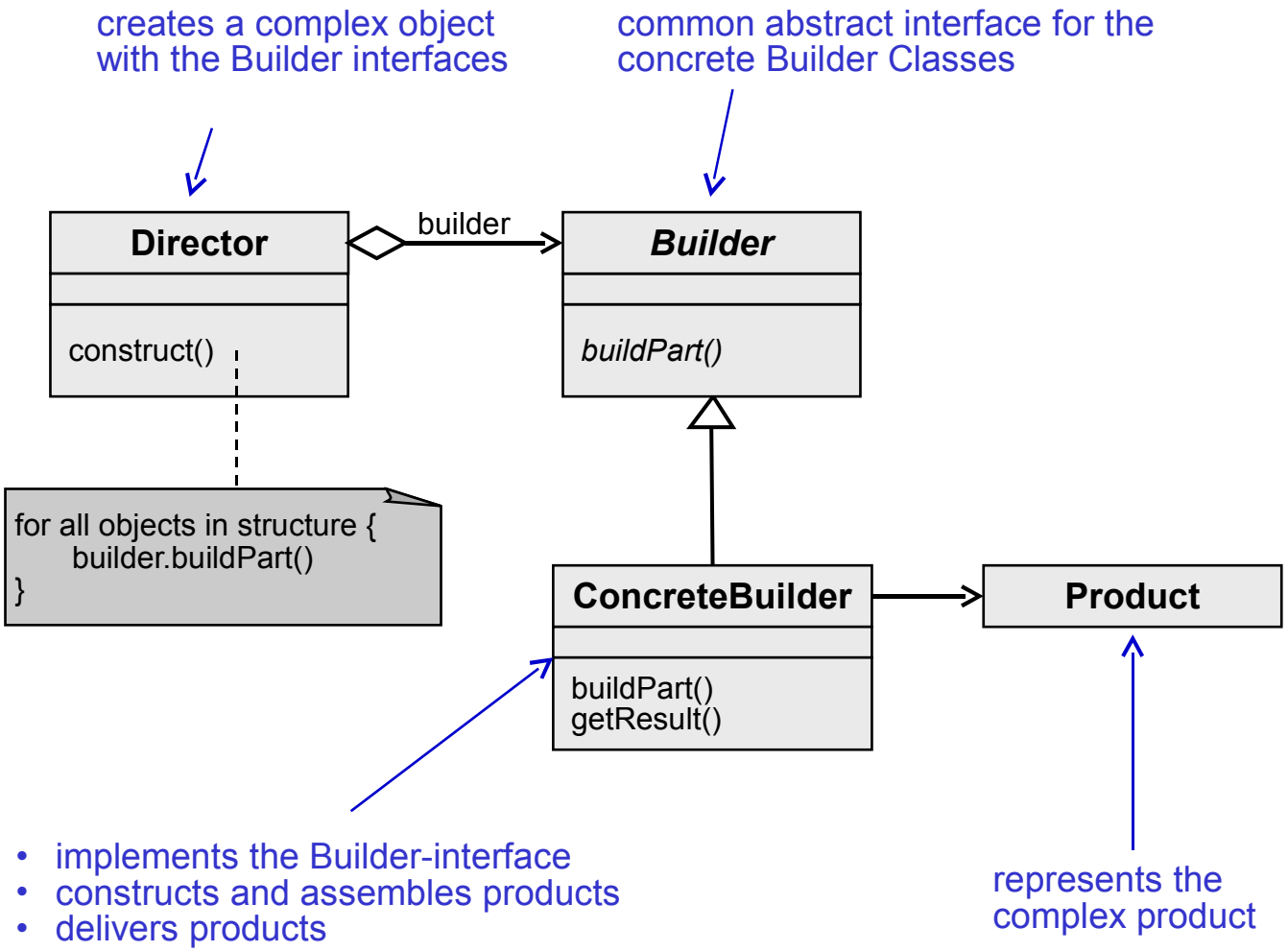
Builder

- Example



Builder

- Structure

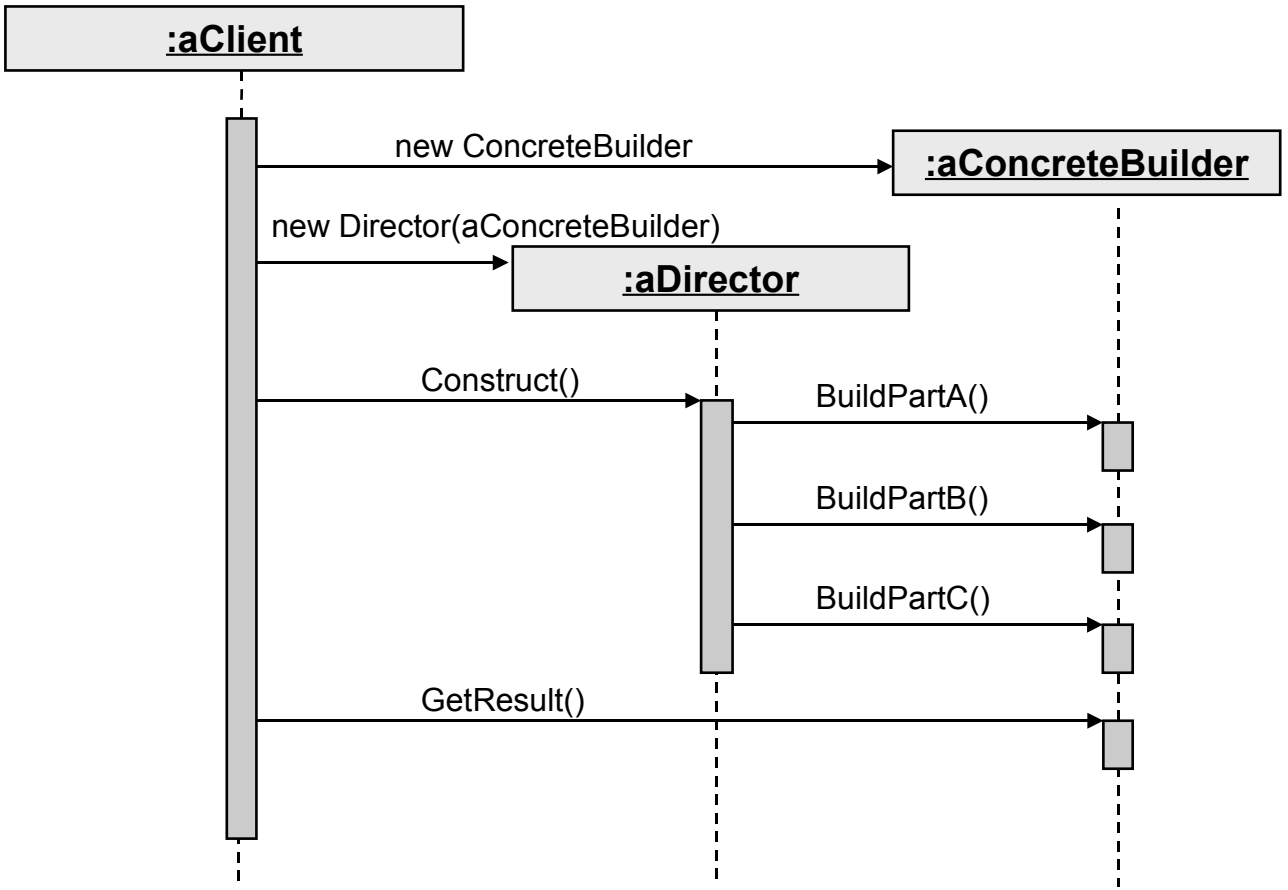


Builder

- Collaboration – Description
  - The **Client** generates the **Director** object, which gets configured with the desired ***Builder*** object
  - **Director** informs the ***Builder***, if a part of the **Product** should be assembled
  - ***Builder*** handles the requests of the **Directors** and adds parts to the **Product**
  - The **Client** gets finally the **Product** from the ***Builder***

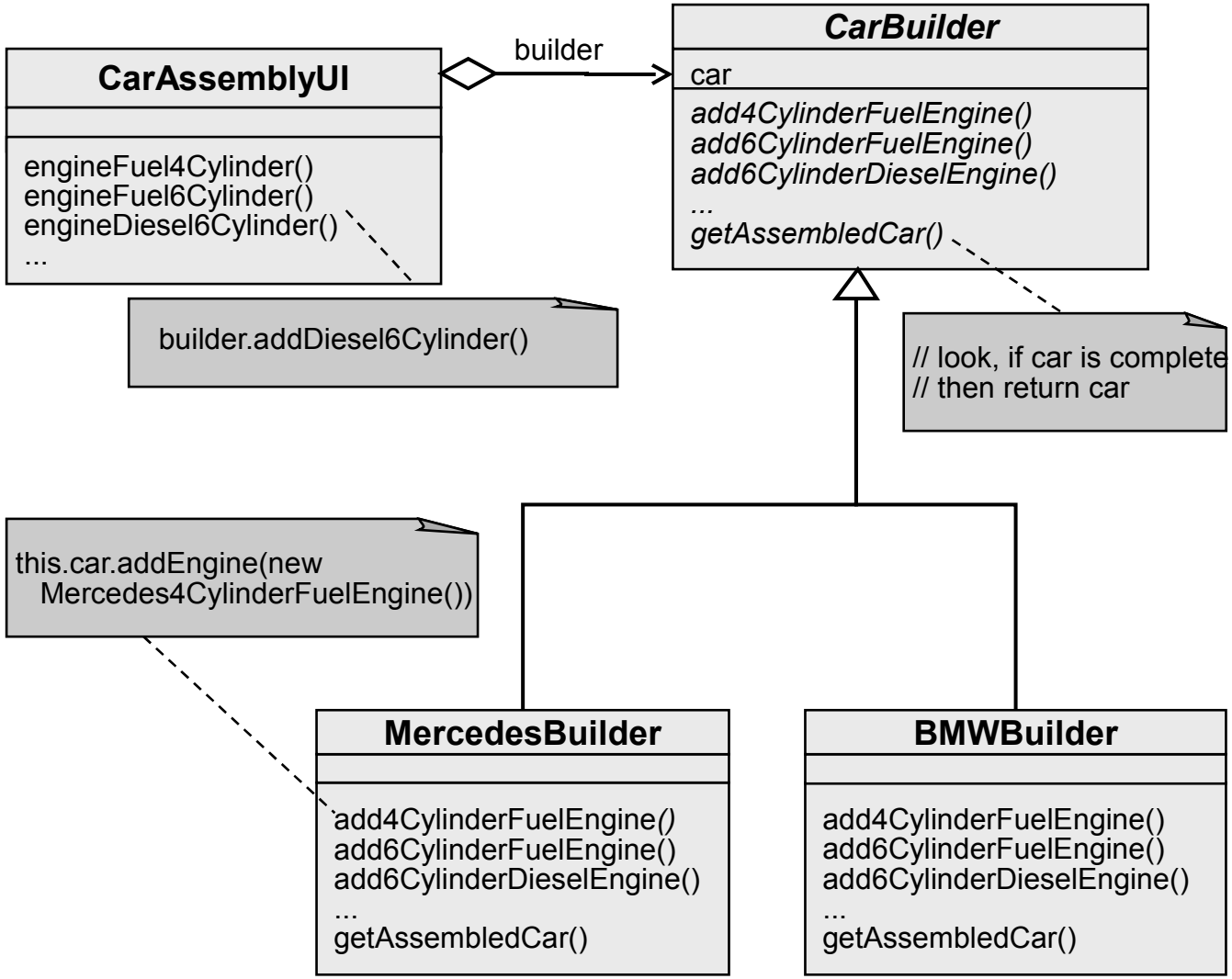
Builder

- Collaboration – Sequence Diagram



Builder

- Example



Builder

- Consequences
  - Internal representation of the **Product** is variable
    - The ***Builder*** offers the **Director** an abstract interface for the construction of a **Product**
    - The representation and internal structure of the **Product** is hidden
    - The internal representation of the **Product** may change, but it does not influence the **Director**

Builder

- Consequences
  - Partitioning of the code in construction process and internal representation
    - Clients do not have to know about the classes representing the internal **Product** structure
    - Reuse is possible
  - Detailed control on the **Product** structure by stepwise construction

Builder

- Implementation
  - ***Builder*** contains methods for all producible components, **ConcreteBuilder** overwrite them
  - Assembly and construction interface
    - ***Builder*** interface has to be flexible enough, to enable the **Product** construction for all **ConcreteBuilder**
    - Is **Product** as return value reasonable (RTF-Reader) or is a return of part nodes better?
      Example: Creation of a maze: Door between two rooms

Abstract Factory and Builder Comparison

- Abstract Factory
  - Goal: Choice out of different product families, independent if the products are complex or simple
  - The factory gets the request, to create and return a complete individual component
  - The client may add components to a complex product, but the factory does not know (and not care)

Abstract Factory and Builder Comparison

- Builder
  - Goal: Creation of a complex object step-by-step
  - If the **Builder** should create an individual component, he does not return, but adds it to an internal encapsulated product
  - Not until the end, if all parts are added, the Builder might be asked for the complete product
- Abstract Factory and Builder could work together for a family of multiple complex products