Chapter 3
Variability Patterns for Object Creation

Prof. Dr. U. Aßmann

Chair for Software Engineering

Department of Computer Science

Technische Universität Dresden

Oct 29, 2018

Lecturer:
Dr.-Ing. Thomas Kühn

3.1 Factory Method (Polymorphic Constructor)

A Restriction of Polymorphism

- Some polymorphic languages (such as Java) do not allow for exchange of the constructor
- Problem: constructors are *concrete,* cannot be varied polymorphically

```
// Creator class abstract
public abstract class Creator {
   public void collect() {
       Set mySet = new Set(10);
       // which set should be allocated?
   }
}
```

```
// Creator class concrete
public class CreatorB extends Creator {
   public void collect() {
       Set mySet = new ListBasedSet(10);
   }
}
```

```
// Product class
public class Set extends Collection {
   public Set(int initial) {
       ....
   }
}
public class ListBasedSet extends Set {
   public ListBasedSet(int initial) {
       …
   }
}
```

So, creator methods, which employ constructors, must be overridden carefully by hand

Factory Method (Polymorphic Constructor)

- Abstract creator classes offer abstract constructors (polymorphic constructors)
  - Concrete subclasses can specialize the constructor
  - Constructor implementation is changed with allocation of concrete Creator

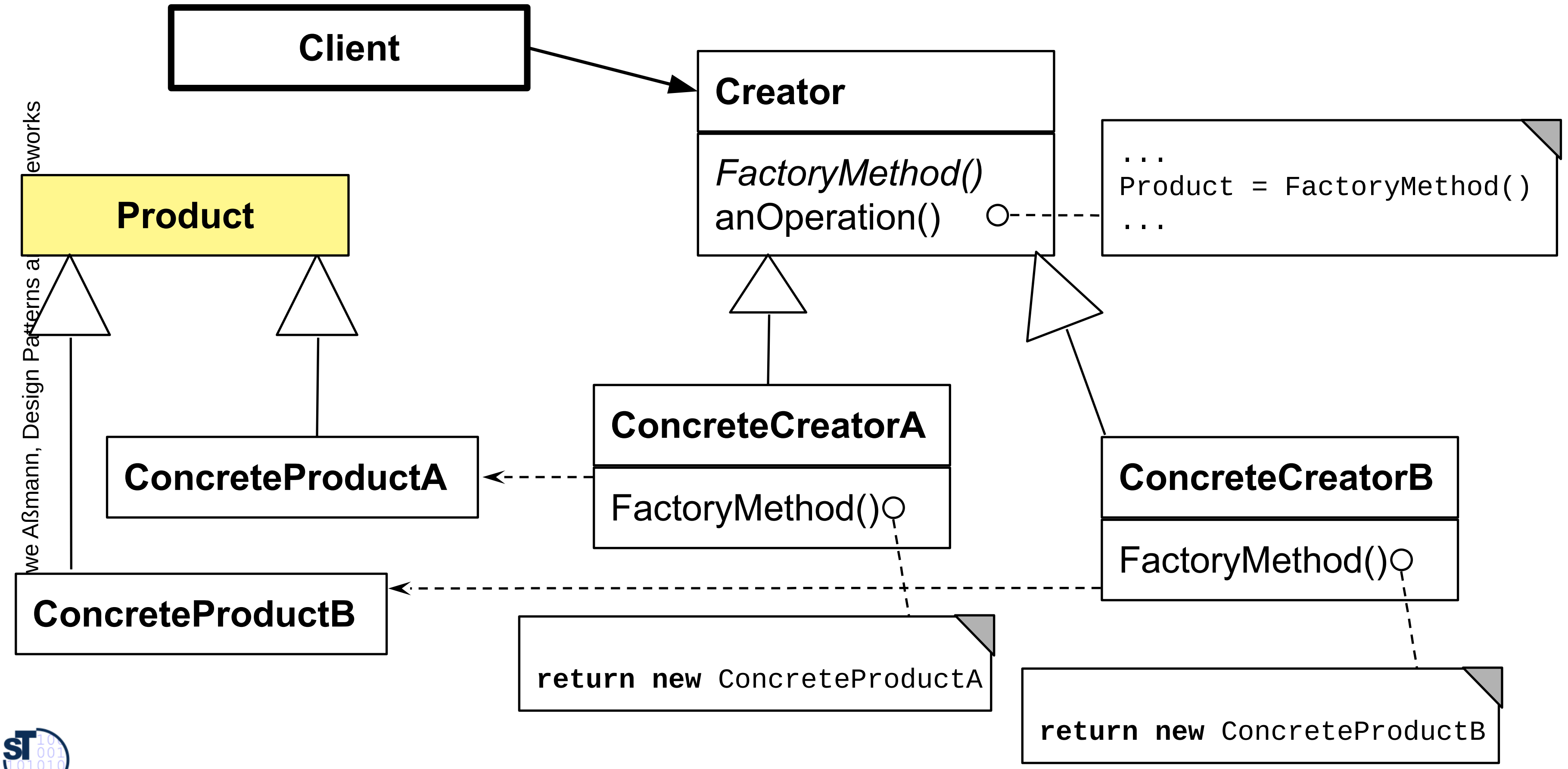
```
// Abstract creator class
public abstract class Creator  {
  // factory method
  public abstract Set createSet(int n);
}
```

```
public class Client {
 … Creator cr = new ConcreteCreator(..)
 public void collect() {
   Set mySet = cr.createSet(10);
   …
 }
}
```

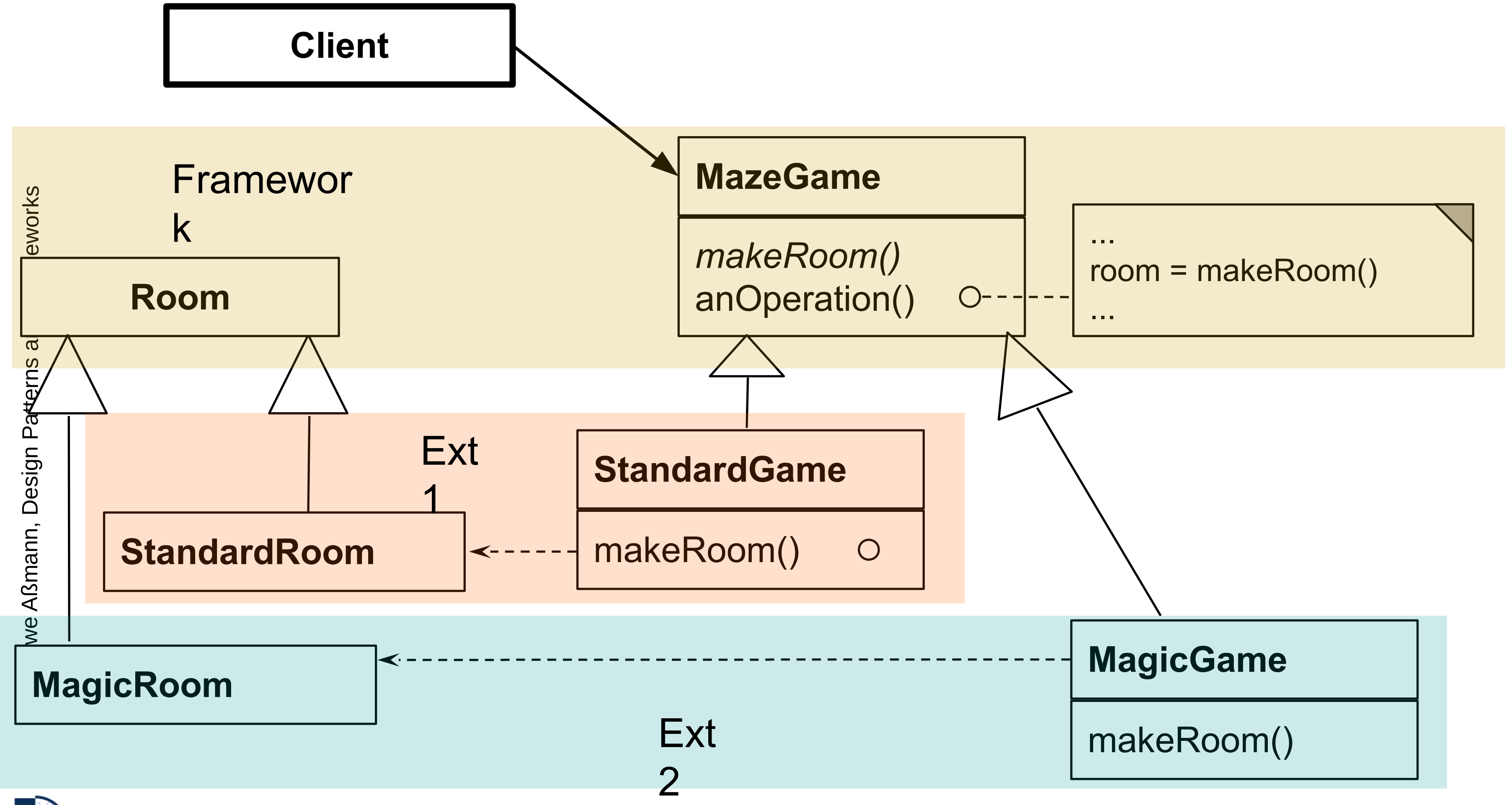
```
// Concrete creator class
public class ConcreteCreator extends Creator {
  public Set createSet(int n) {
    return new ListBasedSet(n);
  }
  …
}
```

Structure for FactoryMethod

- FactoryMethod is a variant of TemplateMethod
- It hides the allocation of a product

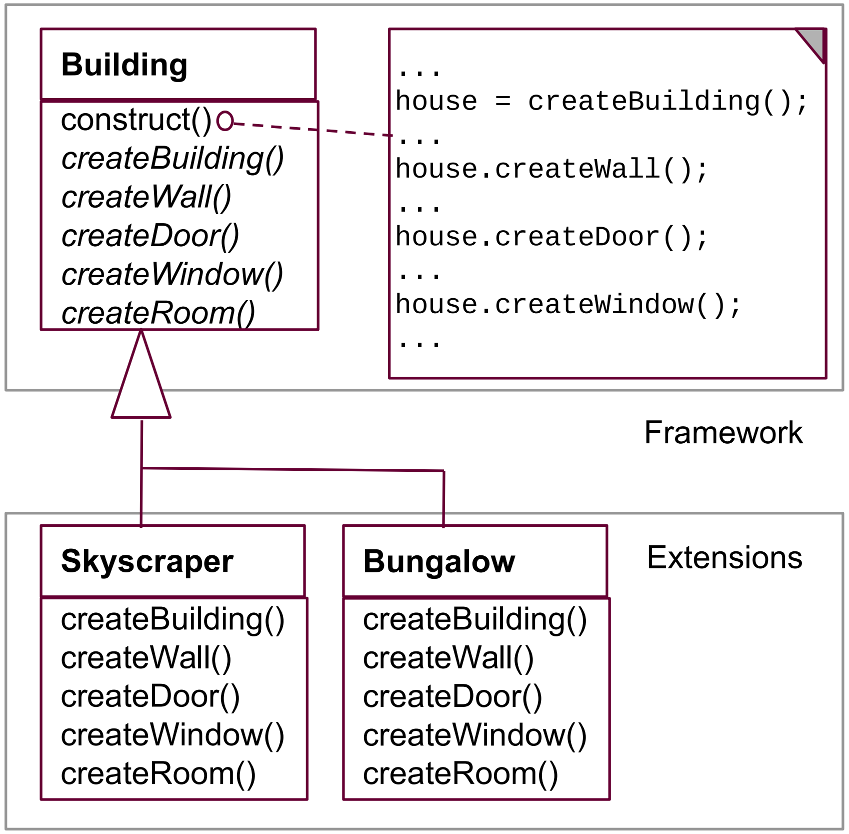


Structure for FactoryMethod



Example FactoryMethod for Buildings

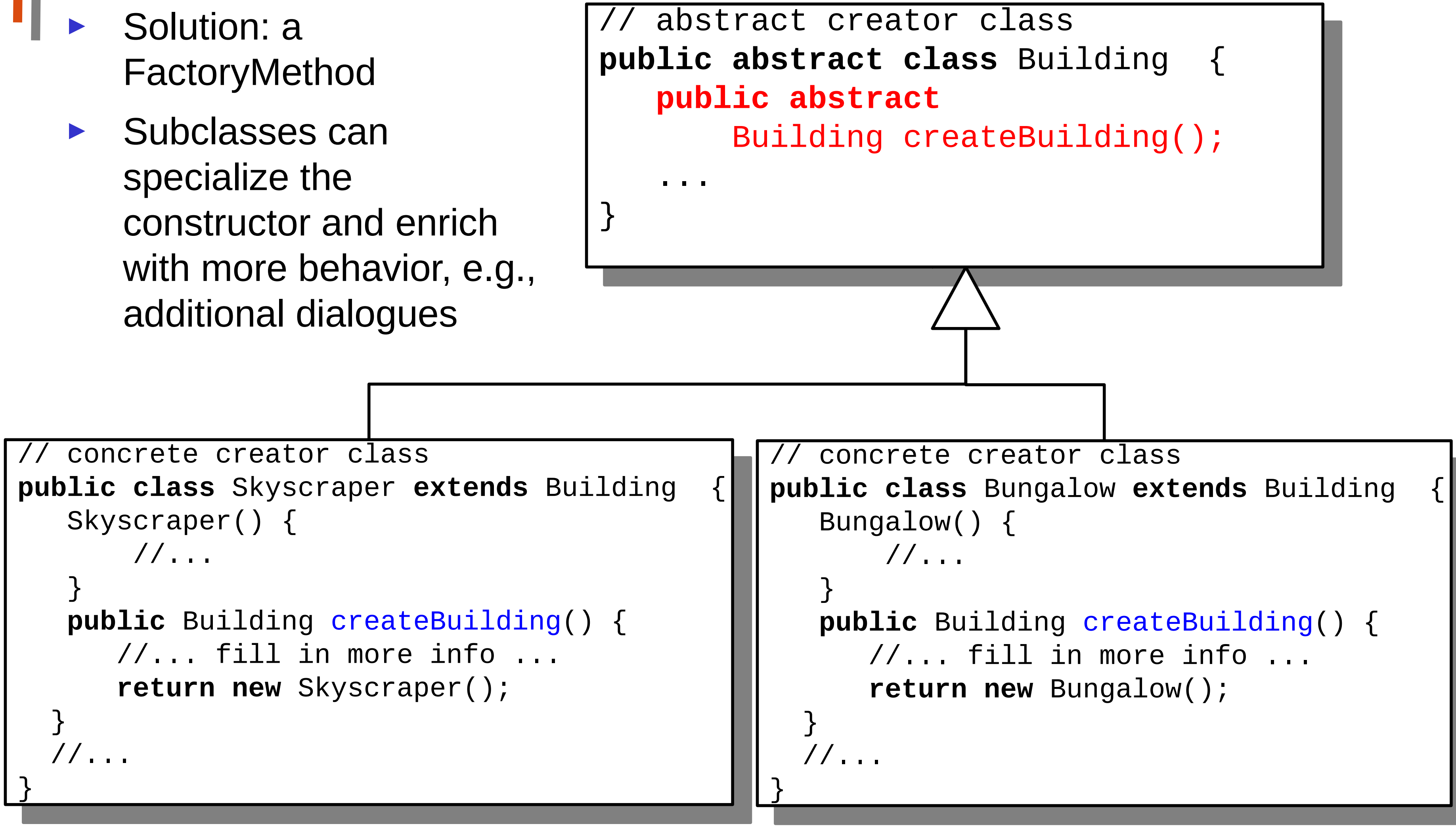
- Consider a framework for planning of buildings
  - Class **Building** with template method **construct** to plan a building interactively
- Users can create new subclasses of buildings
  - All abstract methods must be implemented: `createWall`, `createRoom`, `createDoor`, `createWindow`
- Problem: How can the framework treat new subclasses of Buildings? (unforeseen extension)



```
...
house = createBuilding();
...
house.createWall();
...
house.createDoor();
...
house.createWindow();
...
```

Solution with FactoryMethod

- Solution: a FactoryMethod
- Subclasses can specialize the constructor and enrich with more behavior, e.g., additional dialogues



```
// abstract creator class
public abstract class Building  {
   public abstract
       Building createBuilding();
   ...
}
```

```
// concrete creator class
public class Skyscraper extends Building  {
   Skyscraper() {
       //...
   }
   public Building createBuilding() {
      //... fill in more info ...
      return new Skyscraper();
  }
  //...
}
```

```
// concrete creator class
public class Bungalow extends Building  {
   Bungalow() {
       //...
   }
   public Building createBuilding() {
      //... fill in more info ...
      return new Bungalow();
  }
  //...
}
```

Flexible Construction with Reflection

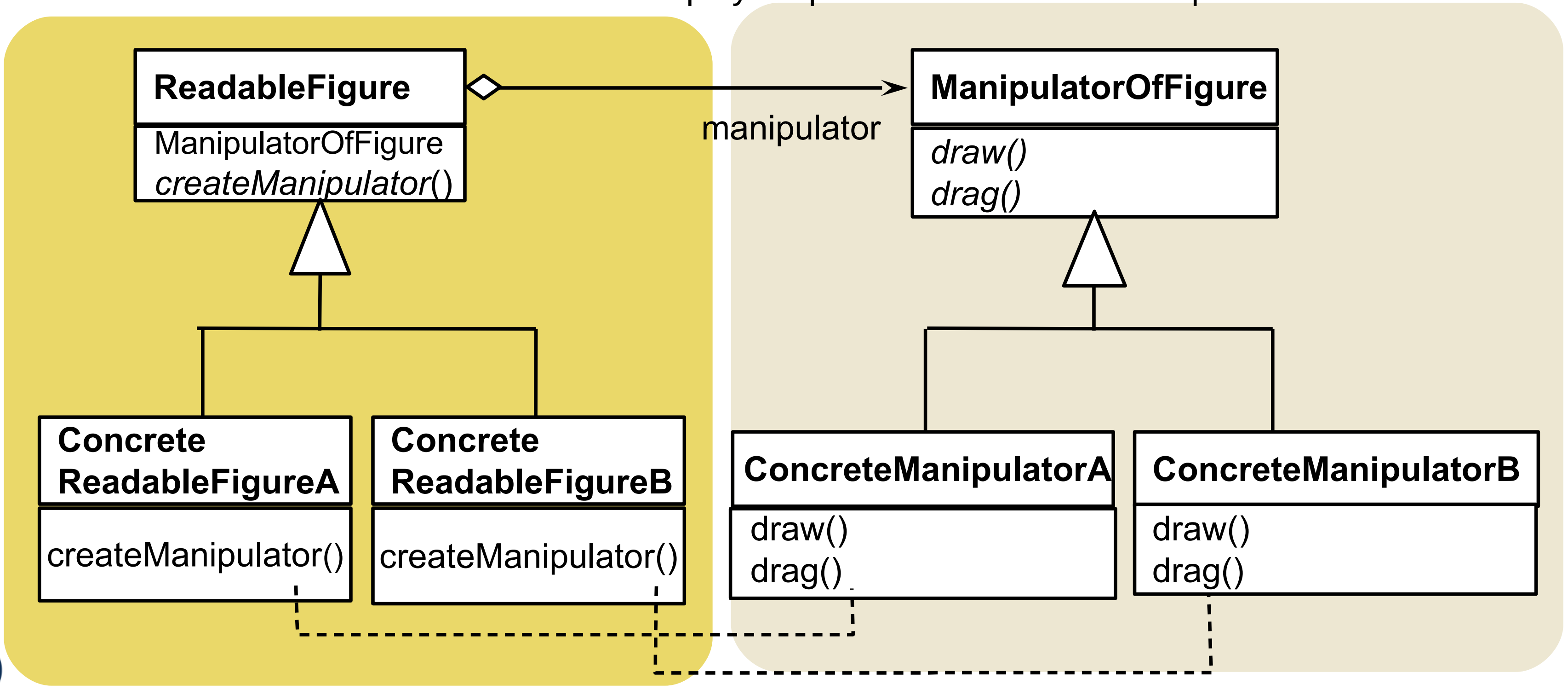
- Constructor can allocate objects of statically unknown classes
- Reflection:
  - Find the class's name and get the class object
  - Then clone the class object

    in Java: **`Class.forName`**`(String name)`
- Attention: reflection is usually slow. It has to lookup bytecode information and must load class code on-the-fly

```
… createProduct() {
    // reflective function for class name, called in subclass
    String className = getClassNameFromSomeWhere();
    // get the class object and allocate from there
    house = (Building) Class.forName(className).newInstance();
    ...
}
```

Factory Methods in Parallel Class Hierarchies

- One class hierarchy offers a factory method to create objects of a second hierarchy
- On every level, the factory method is implemented in a parallel class on exactly the same level and abstraction level
  - E.g, ReadableObject and WritableObject in ReadableFigures and FigureManipulators
- Here, the parallelism constraint is that every readable object must allocate a parallel manipulator.
  - This is a constraint on the polymorphic allocator of the manipulators



Analysis of FactoryMethod: Information Hiding of Abstract Classes

- Abstract classes know *when* an object should be allocated, but do not know which of the subclasses will be filled in at runtime
  - The knowledge which subclass should be used is encapsulated into the client subclasses
- For frameworks this means:
  - The abstract classes of the framework do not know which application class they will work on, but they know when to create an application object
  - The knowledge which application class should be used is encapsulated into the application
- Relatives of FactoryMethod
  - A *FactoryMethod* is a *HookMethod*, used by a *TemplateMethod*, which returns a product, i.e., *FactoryMethods* are called in *TemplateMethods*

3.2 Factory Class (Abstract Factory)

Forces of the Factory Class Pattern

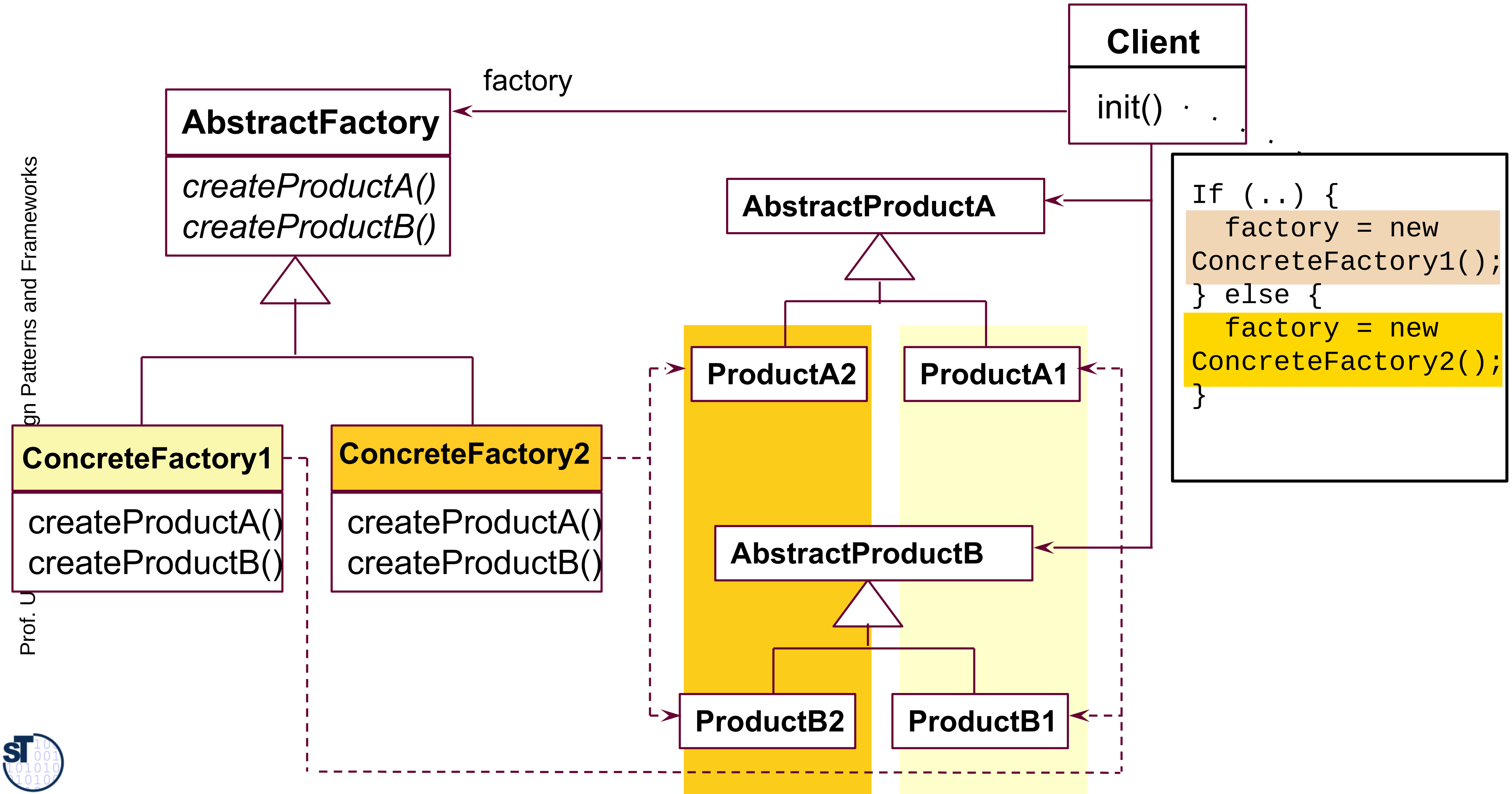
- Given a package with a family of classes (a *product family*). Examples
  - **Widgets** in a window system
  - **Stones** in a Tetris game
  - **Products** of a company
- How can the product family be switched in one go to a variant?
  - Swing widgets to Windows widgets?
  - 2D-stones to 3D-stones in the Tetris game?
  - Cheap variants of the products of the company to expensive variants?

Factory Class Pattern

- A **Factory (FactoryClass)** groups factory methods to a class
  - A Factory is a class that groups a *family of polymorphic constructors* of a family of classes (products)
  - The products can be classes of a layer or a package
  - The products have a strong parallelism constraint (isomorphic hierarchies)
- An **AbstractFactory** contains the interfaces of the constructors
- A **ConcreteFactory** contains the implementation of the constructors
  - The Concrete Factories can be exchanged
  - A Concrete Factory represents one concrete family of objects
- Hence, an **AbstractFactory** offers an interface to create families of related objects
  - That depend on each other
  - Without naming their constructors explicitly

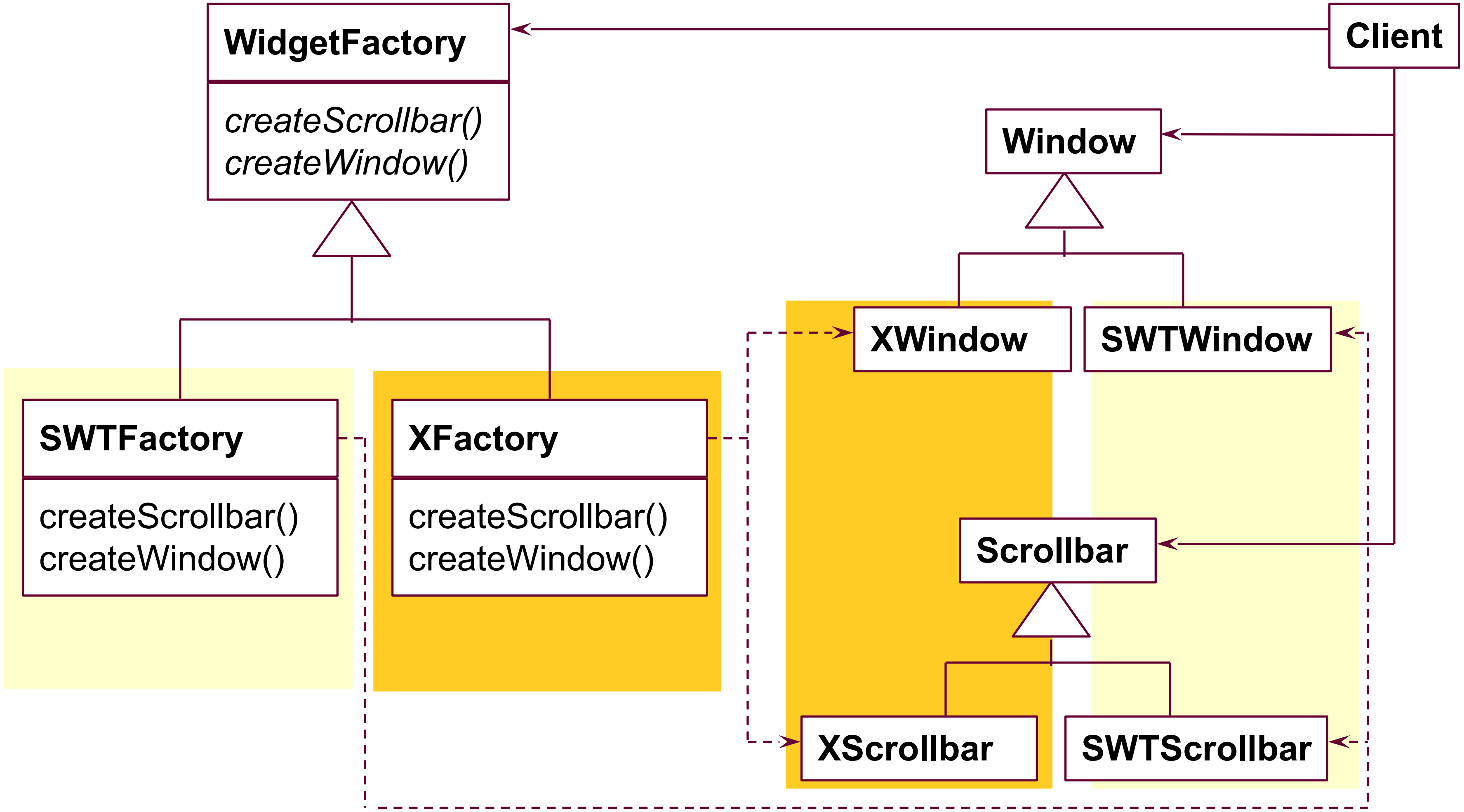
Structure for Factory Class

- By creating the concrete factory, the client determines the entire family of products (here: family 1 or 2)

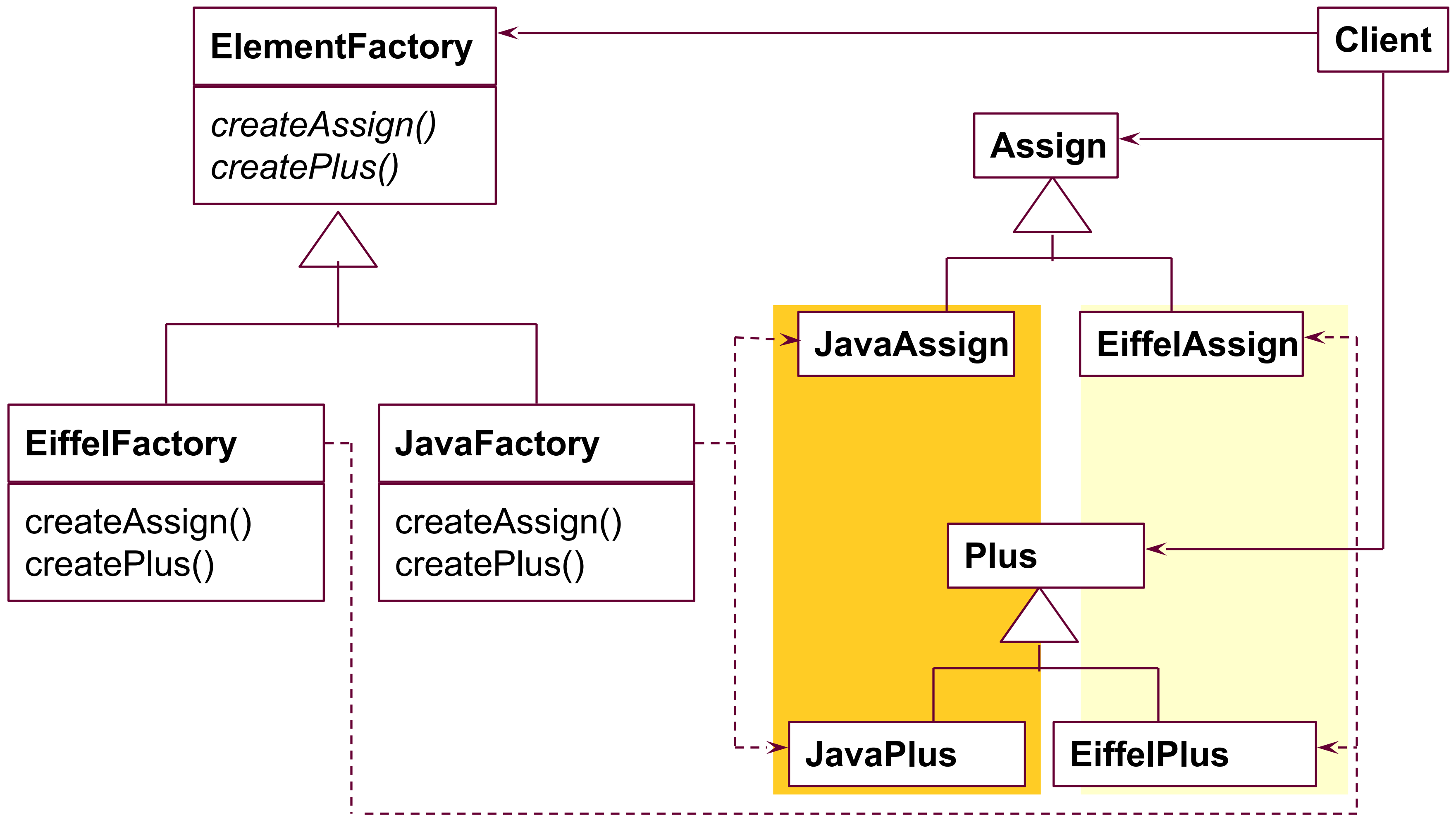


```
If (..) {
   factory = new
ConcreteFactory1();
} else {
   factory = new
ConcreteFactory2();
}
```

Example for Factory Class



Example for Factory Class in Compilers

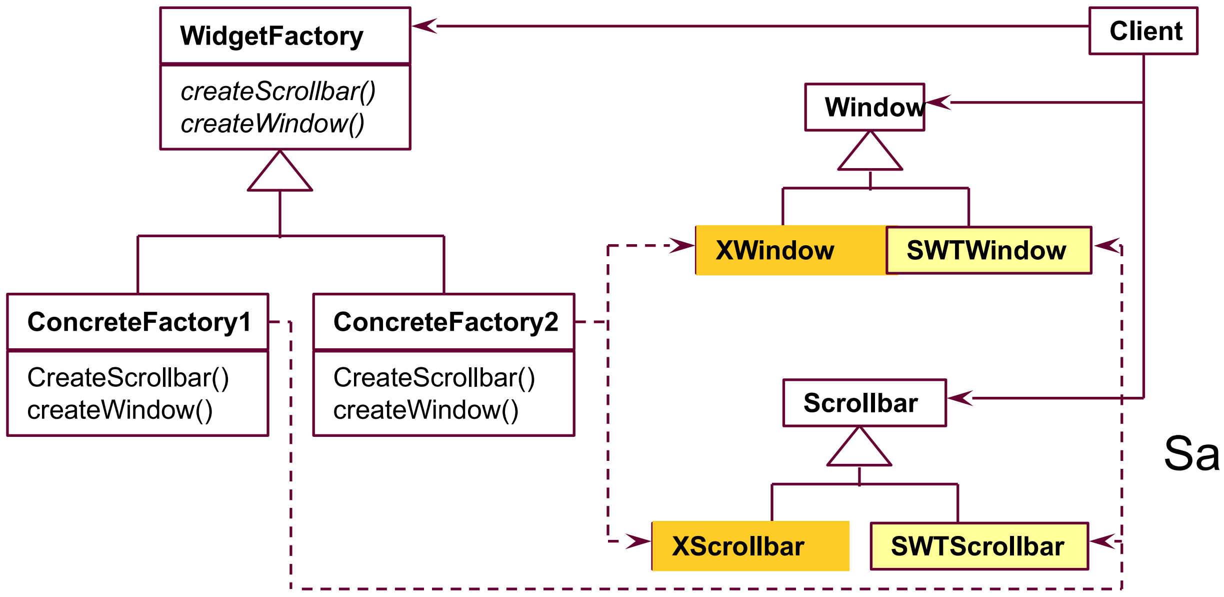


Employment of Factory Class

- For window styles
  - All widgets are used by the framework abstractly
  - The concrete style is determined by a concrete factory class
  - Swing, AWT, ...
- In office systems
  - For families of similar documents
- In business systems
  - For families of similar products
- For tools on several languages
- **Factory Class** is related to *Tools-and-Materials* (TAM), because products are materials (see later)

Pragmatics of Factory Class

- A factory deals with 3+x inheritance hierarchies (factory, product 1, ..., product n)
- The *n* product hierarchies must be maintained *in parallel*, i.e., they form **ParallelHierarchies**
- The factory pattern ensures that all objects are created with the parallelism constraint

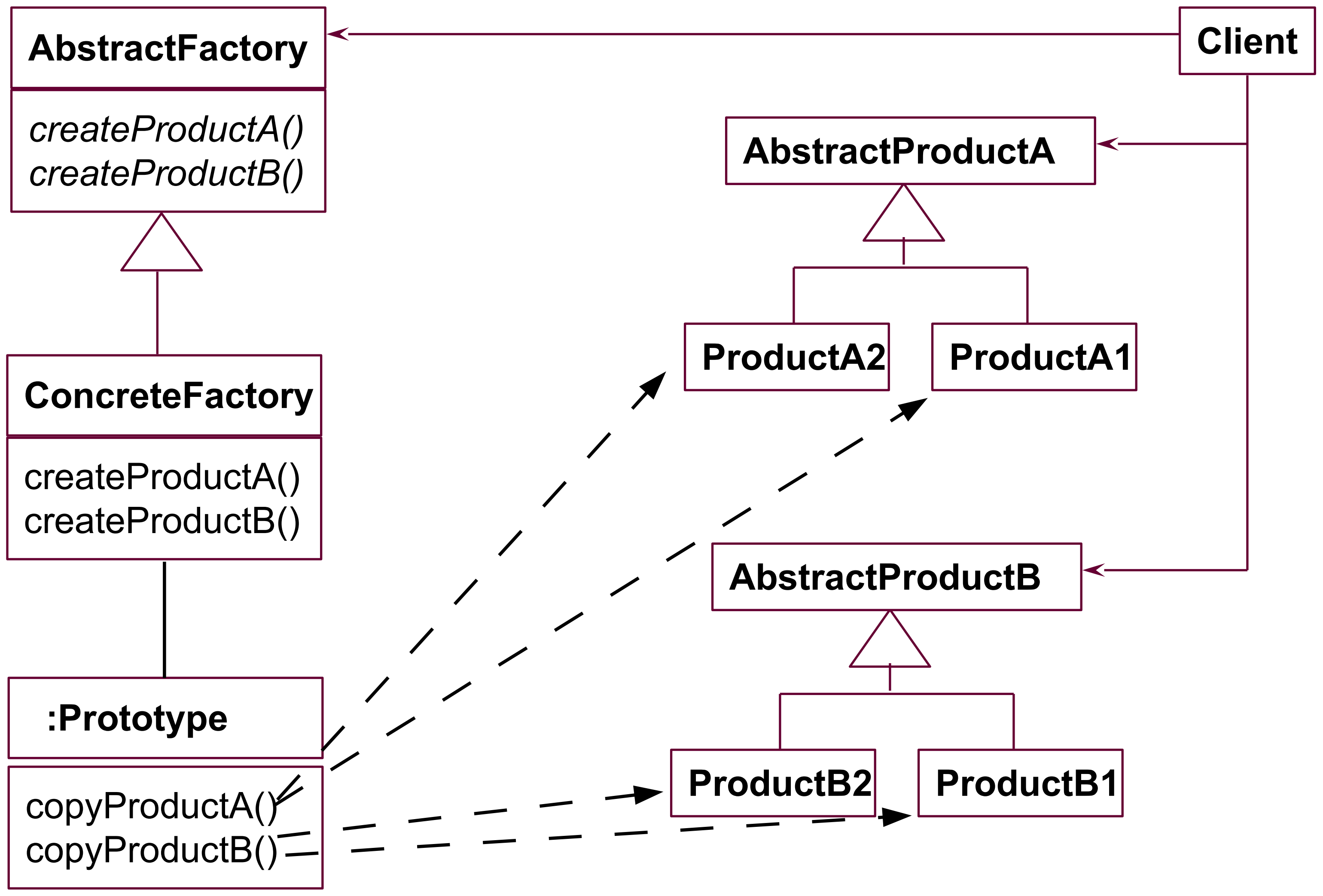


Same height of products

Variant: The Prototyping Factory

- Concrete factories need not be created; one instance is enough, if prototypes of the products exist
- To produce new products, the **ConcreteFactory** clones the set of available products
- The variability of products is handled by the cloning of the prototypes
- Especially useful, if products have complex default state or do not vary much

Structure for Prototyping Factory



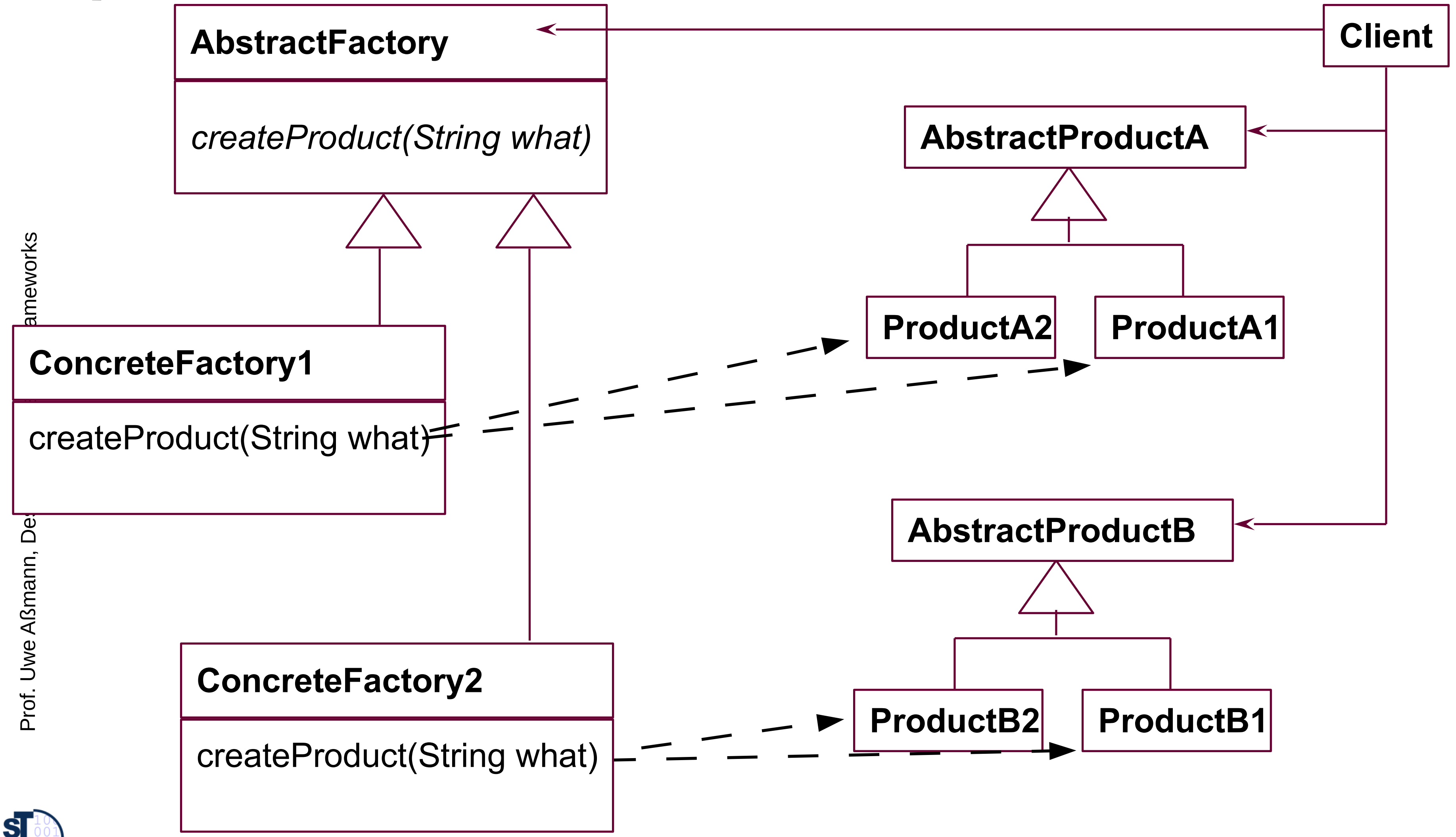
Variant: Factory with Interpretive FactoryMethod

- If more factory methods should be added, this becomes tedious, since the AbstractFactory and all concrete factories must be edited
- Instead: one factory method with parameter string

```
public class abstractFactory {
     abstract Product createProduct(String what);
}
```

```
public class ConcreteFactory extends AbstractFactory {
 Product createProduct(String what) {
       if (what.eq(“p1”)) {
             return new P1();
       Else …
 }
}
```

Structure for Interpretive Factory



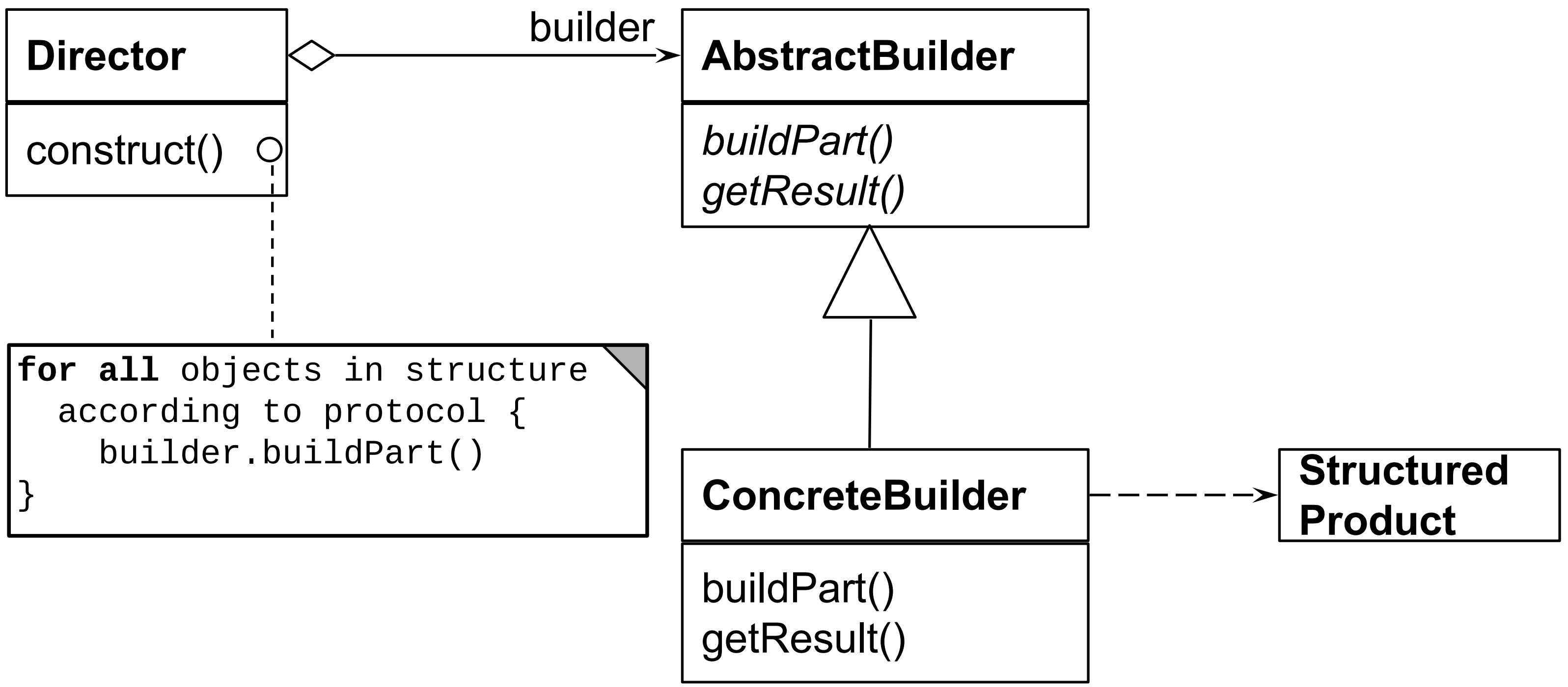
Factory Class - Employment

- Make a system independent of the way how its objects are created
- Hide constructors to make the way of creation exchangable with types
- For product families
  - In which families of objects need to be created together; but the way how is varied
- Related Patterns
  - An *abstract factory* is a special form of *hook class*, to be called by some template classes.
  - Often, a *factory* is a **Singleton** (a Singleton is a class with only one instance)
  - Concrete *factories* can be created by parameterizing the *factory* with **Prototype** objects

3.3 Builder (Factory with Protocol, Structured Factory)

Structure for Builder

- The **Builder** is a Factory Class that produces a *structured* product (a whole with parts)
  - e.g., a business object or product data

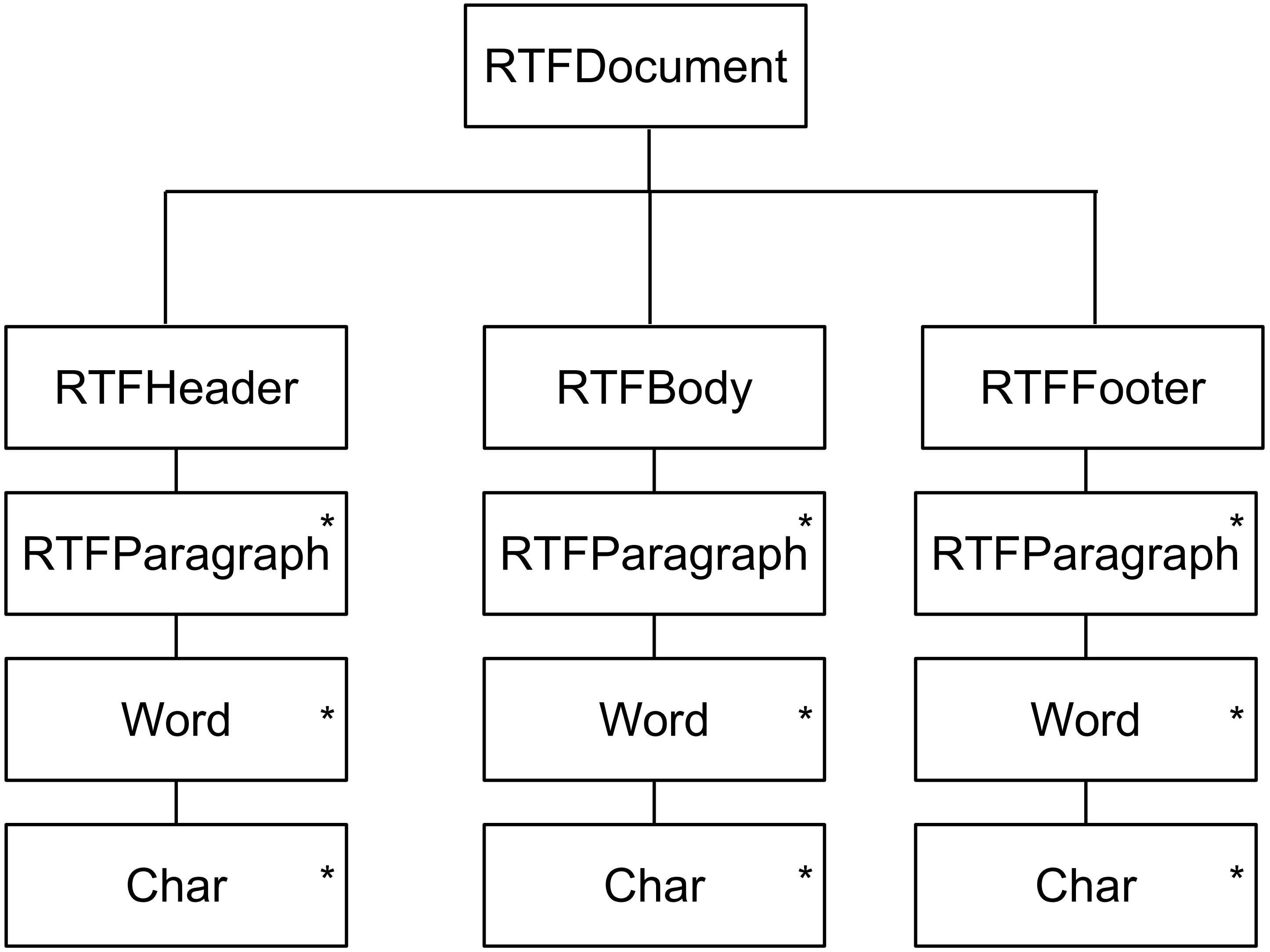


```
for all objects in structure
  according to protocol {
    builder.buildPart()
}
```

Builder Protocol (E.g., Specified by EBNF)

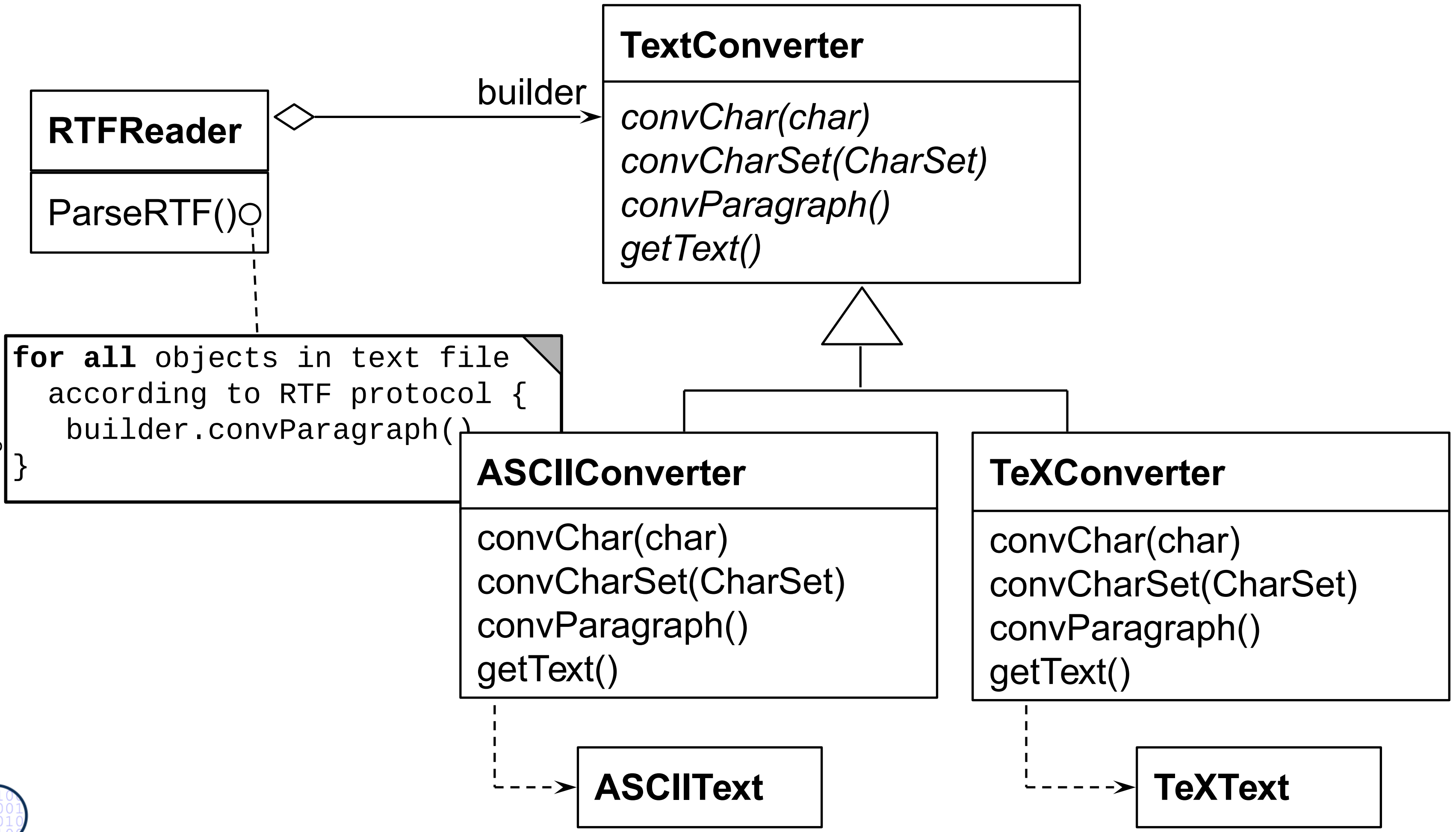
```
-- Grammar in EBNF
RTFDocument ::= RTFHeader RTFBody RTFFooter.
RTFHeader   ::= RTFParagraph*.
RTFParagraph::= Word*.
Word        ::= Char*.
RTFBody     ::= RTFParagraph*.
RTFFooter   ::= RTFParagraph*.
```

RTF Builder Protocol



Example RTF-Document Builder

- RTF grammar defines a protocol for the sequence of text converter functions



```
for all objects in text file
  according to RTF protocol {
   builder.convParagraph()
}
```

The Builder

- Maintains an internal state that memorizes the point of time in construction of the complex data structure
- Data structure defines a protocol for calls to the elementary functions
- Data structure must be defined by a
  - *Grammar*
  - *Regular expression*
  - *Protocol machine* (statechart acceptor)
  - Other mechanisms, such as *Petri nets*
- The other way round: as soon as we have a data structure
  - Defined by a grammar or regular expressions
  - We can build a constructor with the Builder pattern

Builder: Information Hiding

- The builder hides
  - The protocol (the structure of the data)
  - The current status
  - The implementation of the data structure
- Similar to an Iterator, the structure is hidden

Known Uses

- Parsers in compilers are builders that contain the grammar of the concrete syntax of the programming language
- Builders for intermediate representations of all kinds of languages
  - *Programming languages*
  - *Specification languages*
  - *Graphic languages* such as UML
- Builders for all complex data structures
  - Databases with integrity constraints

What have we learned?

- **Factory Method**

  Problem: constructors cannot be varied

  Solution: Application of Template Method for Creation

- **Factory Class**

  Problem: No variability of constructors in dimensional class hierarchies

  Solution: Application of Template Class for Creation

- **Builder**

  Problem: Complex products are build according to a protocol, which is to be varied, too.

  Solution: Application of Template Class with stateful template method