Chapter 3
Variability Patterns for Object Creation

1) FactoryMethod
2) AbstractFactory
3) Builder

Prof. Dr. U. Aßmann

Chair for Software Engineering

Faculty of Informatics

Dresden University of Technology

Version 11-0.3, 10/21/11

3.1 Factory Method (Polymorphic Constructor)

A Restriction of Polymorphism

- Often, polymorphic language do not allow for exchange of the constructor
- Problem: constructors are *concrete*, cannot be varied polymorphically

```
// Product class
public class Set extends Collection {
    public Set(int initialLength) {
        ...
    }
}

public class ListBasedSet extends Set {
    public ListBasedSet(int initialLength) {
        ...
    }
}
```

```
// 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() {
        mySet = new ListBasedSet(10);
    }
}
```

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

3

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);
}
```
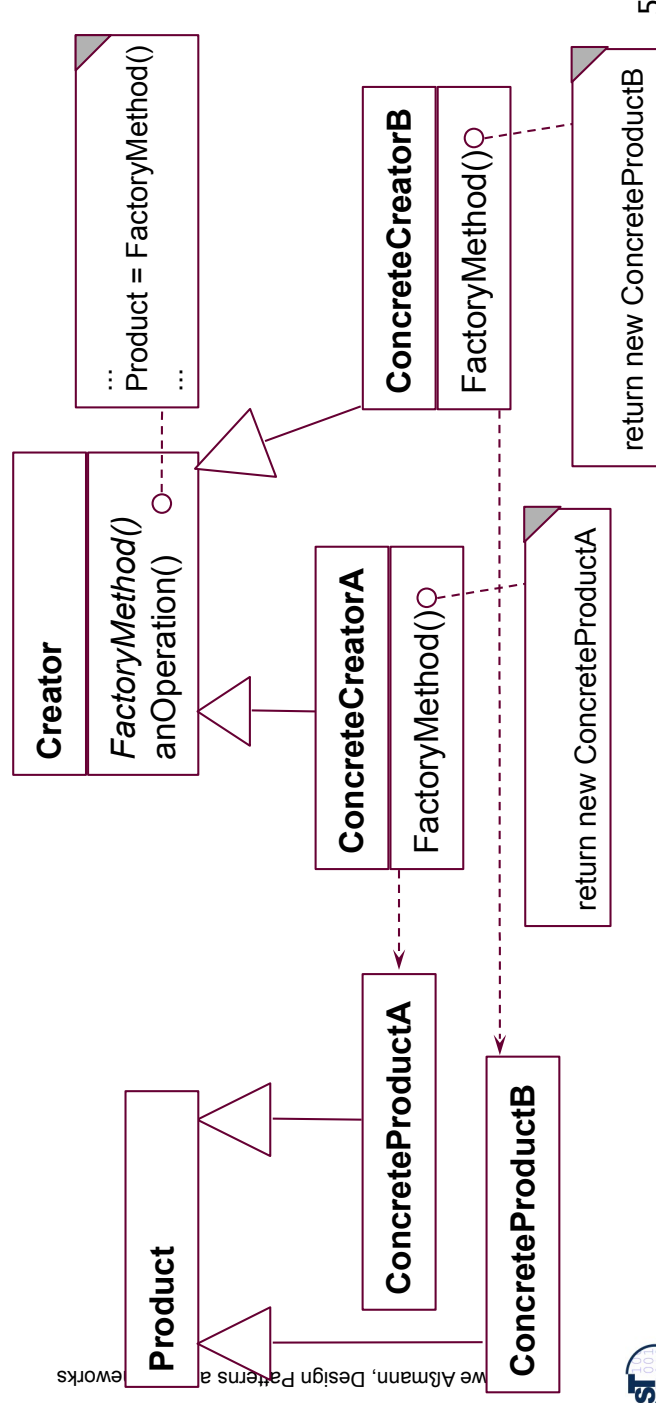
```
// Concrete creator class
public class ConcreteCreator extends Creator {
    public Set createSet(int n) {
        return new ListBasedSet(n);
    }
    ...
}
```
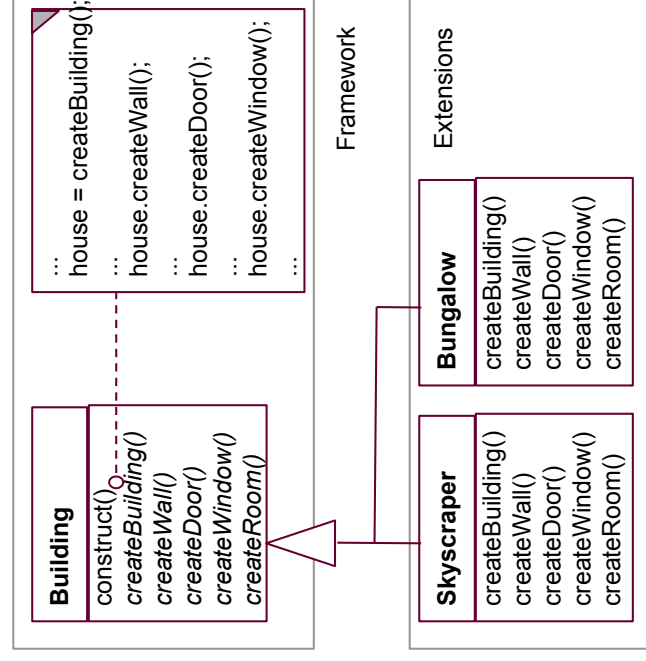
```
public class Client {
    ... Creator cr = [.. subclass ]...
    public void collect() {
        Set mySet = Creator.createSet(10);
        ...
    }
}
```

Structure for FactoryMethod

▲ FactoryMethod is a variant of TemplateMethod

▲ It hides the allocation of a product

Creator
- *FactoryMethod()*
- anOperation()

Product = FactoryMethod()

ConcreteCreatorA
- FactoryMethod()

return new ConcreteProductA

ConcreteCreatorB
- FactoryMethod()

return new ConcreteProductB

Product

ConcreteProductA

ConcreteProductB

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 createWall, createRoom, createDoor, createWindow must be implemented

▲ Problem: How can the framework treat new subclasses of Buildings? (unforeseen extension)

Building
- construct()
- *createBuilding()*
- *createWall()*
- *createDoor()*
- *createWindow()*
- *createRoom()*

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

Framework

Skyscraper
- createBuilding()
- createWall()
- createDoor()
- createWindow()
- createRoom()

Bungalow
- createBuilding()
- createWall()
- createDoor()
- createWindow()
- createRoom()

Extensions

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();
    }
    ...
}
```

Flexible Construction with 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();
    ...
}
```
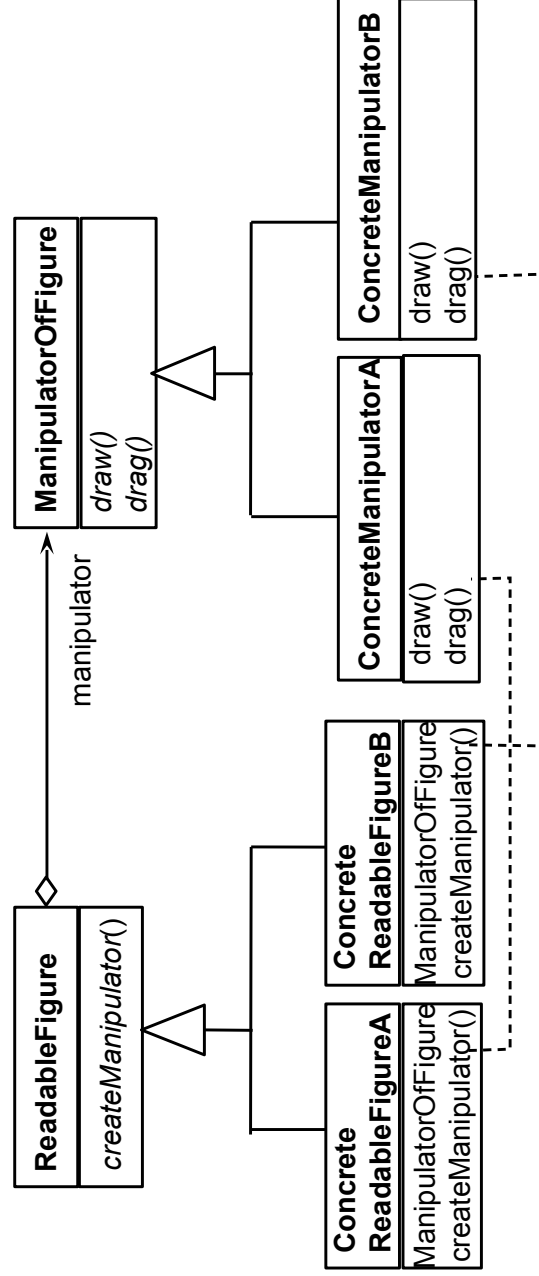
Combination of Factory Method and Default Implementation

▲ FactoryMethods can contain default implementations to share behavior

▲ Subclass has to call super()

```
// abstract class with default
// behavior
public abstract class Building {

  public abstract
  Building createBuildingInner();

  public
  Building createBuilding() {
    Building b = createBuildingInner();
    Door d = new Door();
    b.setDoor(d);
    return b;
  }
  ...
}
```

```
// concrete class with additional
// behavior
public class Skyscraper extends Building {
  // concretization of hook
  public Building createBuildingInner() {
    return new Skyscraper();
  }
  ...
}
```

Factory Methods in Parallel Class Hierarchies

▲ One class hierarchy offers uses 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

Creation of Product Subclasses with Generics

```
// Generic factory class
template<class TheProduct>
class StandardProducer<TheProduct> : public Producer {
  Product* StandardProducer<TheProduct>() {
    return new <TheProduct>();
  }
  ...
}
```

```
// Application of generic factory class creates concrete
// FactoryMethod automatically
Public abstract class Building {
  StandardProducer<MyProduct>  myProducer;
  myProducer = new myProducer.StandardProducer<MyProduct>()
  ...
}
```

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?

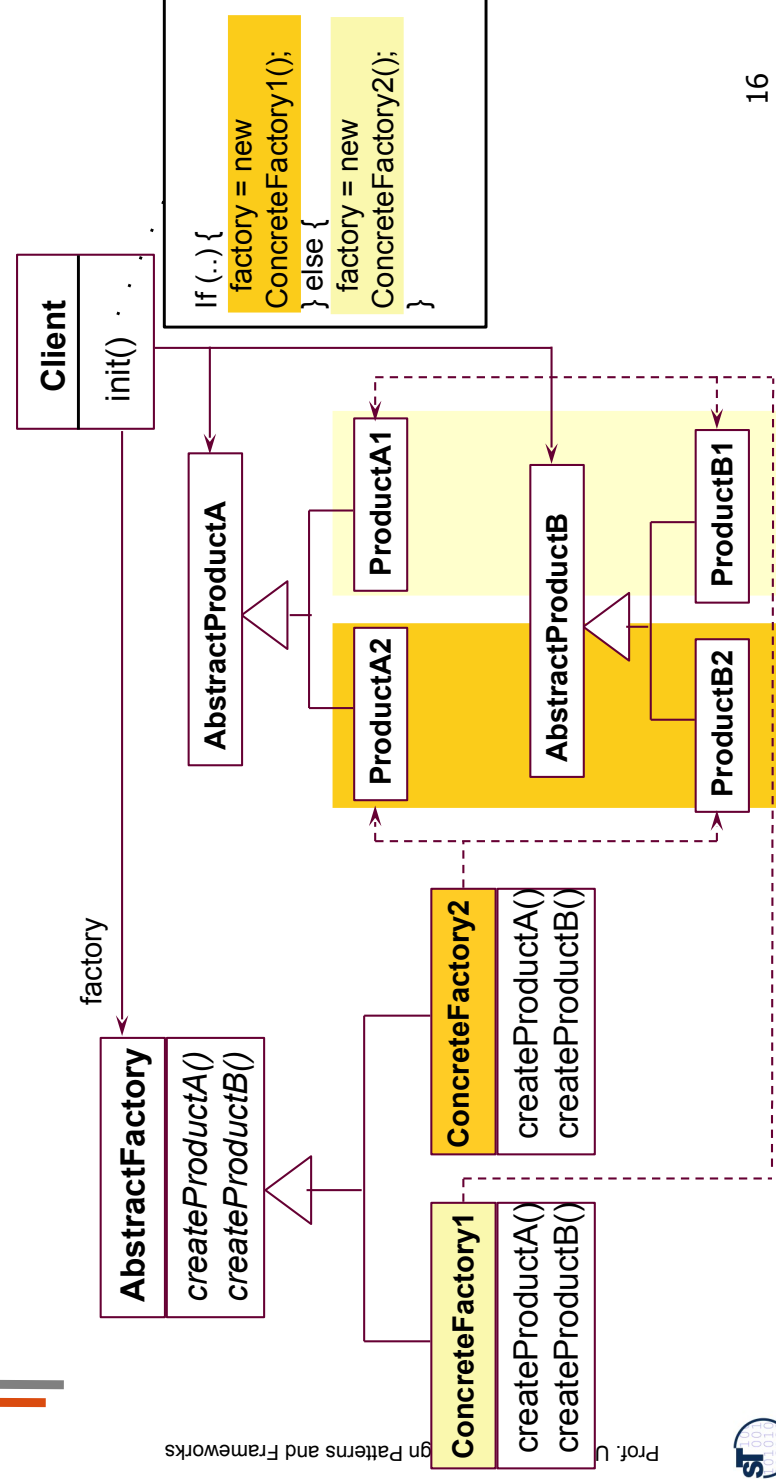   ▪ Cheap variants of the products of the company to expensive variants?
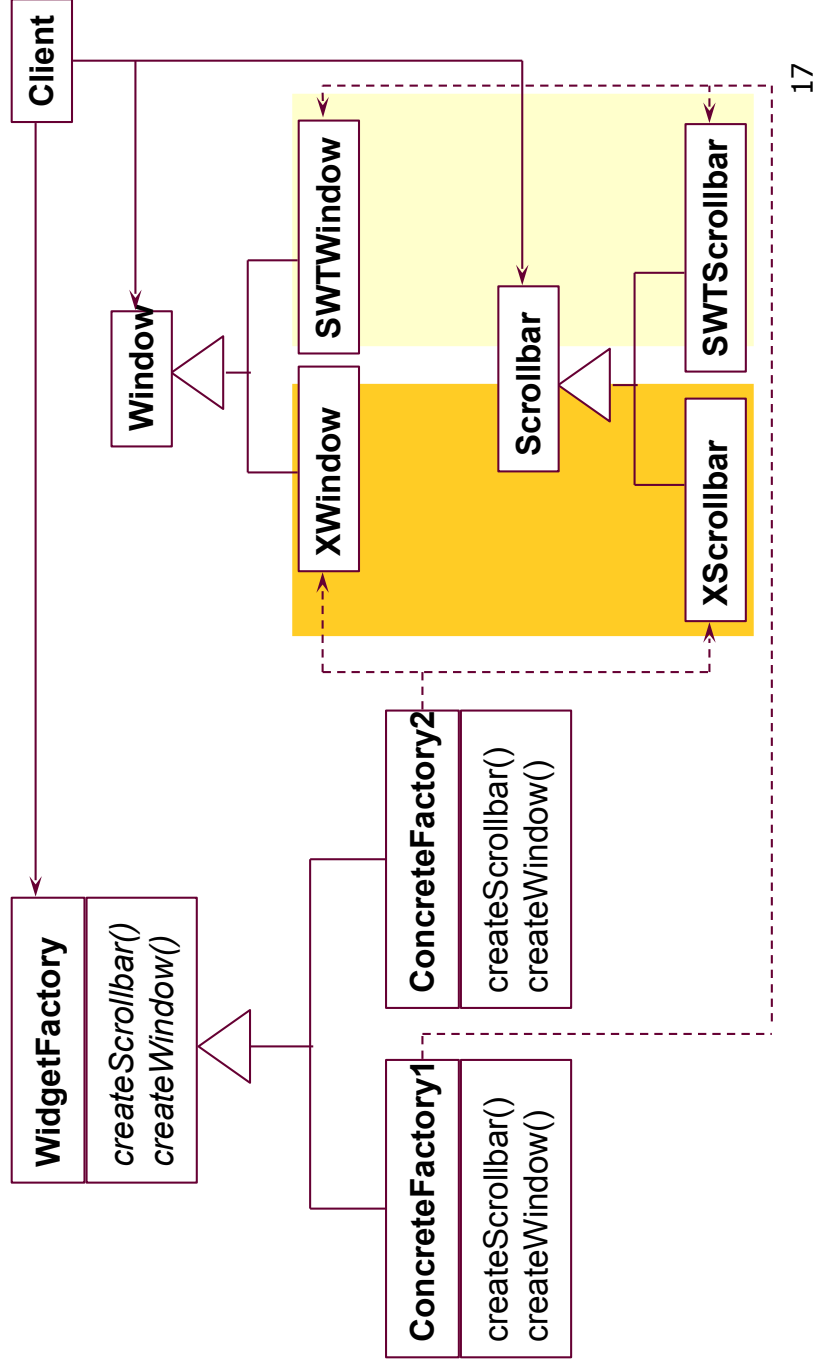
Factory Class Pattern

- ▲ 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
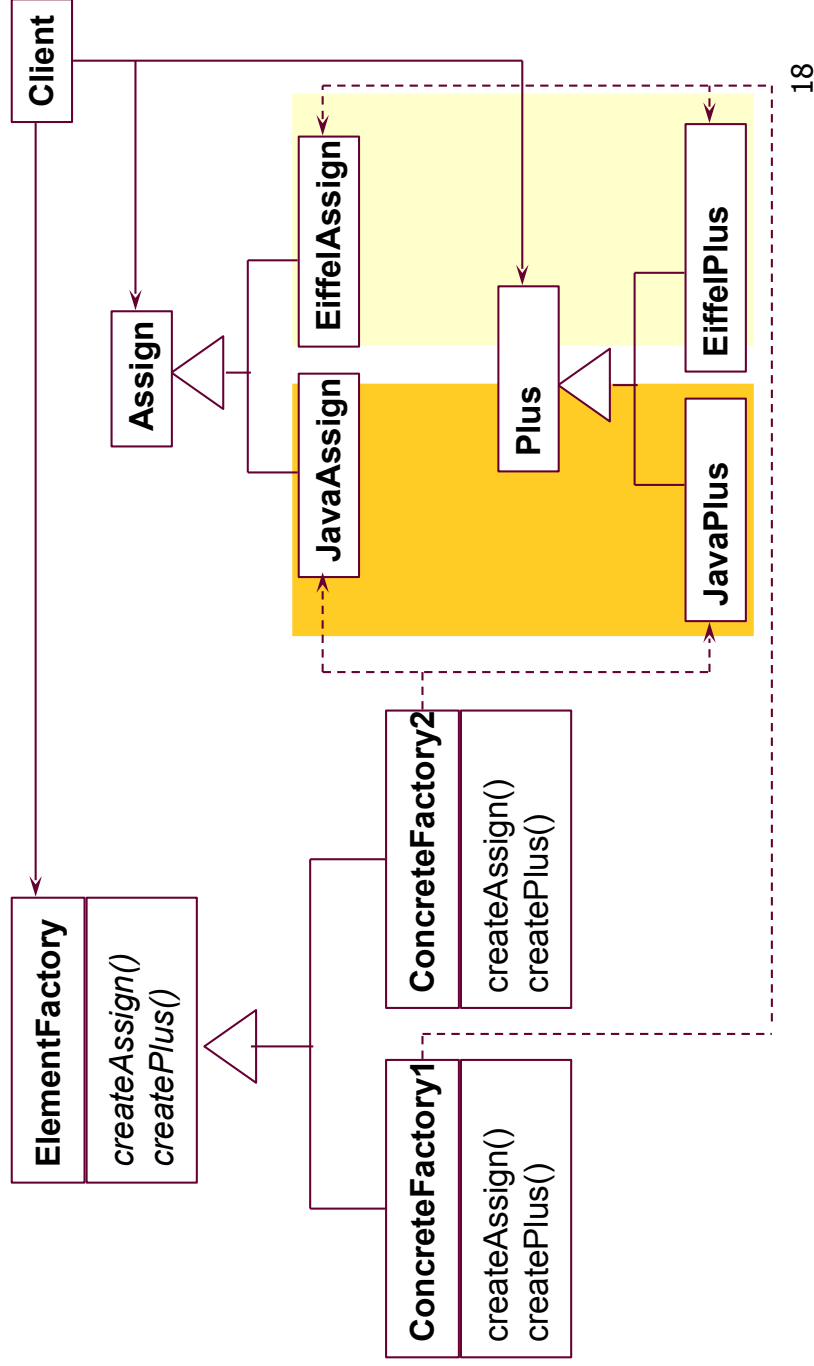

Structure for Factory Class

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

Client
init()

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

factory

AbstractFactory
createProductA()
createProductB()

ConcreteFactory1
createProductA()
createProductB()

ConcreteFactory2
createProductA()
createProductB()

AbstractProductA

ProductA1

ProductA2

AbstractProductB

ProductB1

ProductB2

Example for Factory Class

Client

Window

SWTWindow

XWindow

Scrollbar

SWTScrollbar

XScrollbar

WidgetFactory
createScrollbar()
createWindow()

ConcreteFactory2
createScrollbar()
createWindow()

ConcreteFactory1
createScrollbar()
createWindow()

Example for Factory Class in Compilers

Client

Assign

EiffelAssign

JavaAssign

Plus

EiffelPlus

JavaPlus

ElementFactory
createAssign()
createPlus()

ConcreteFactory2
createAssign()
createPlus()

ConcreteFactory1
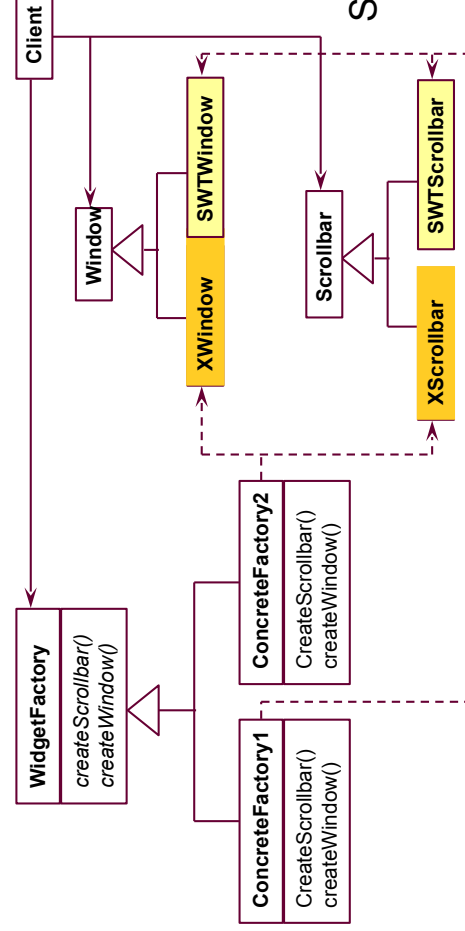createAssign()
createPlus()

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)
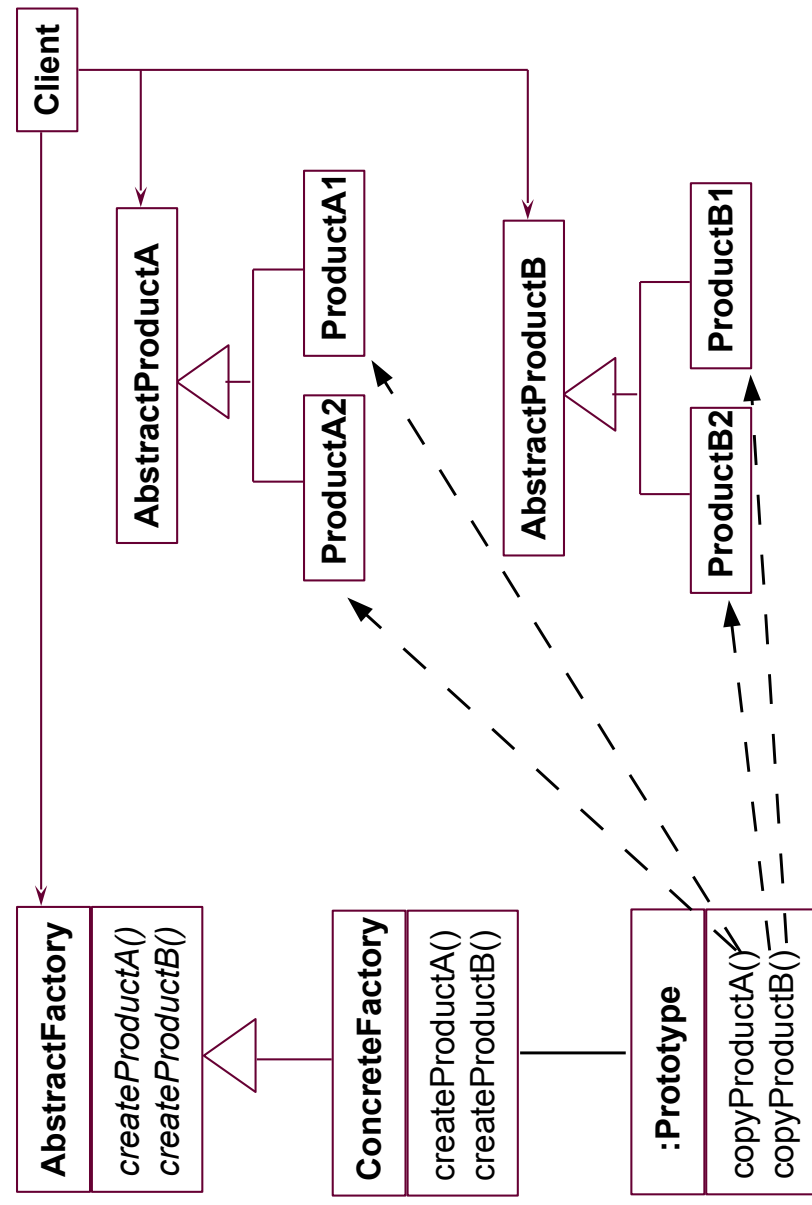

Pragmatics of Factory Class

- ▲ 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
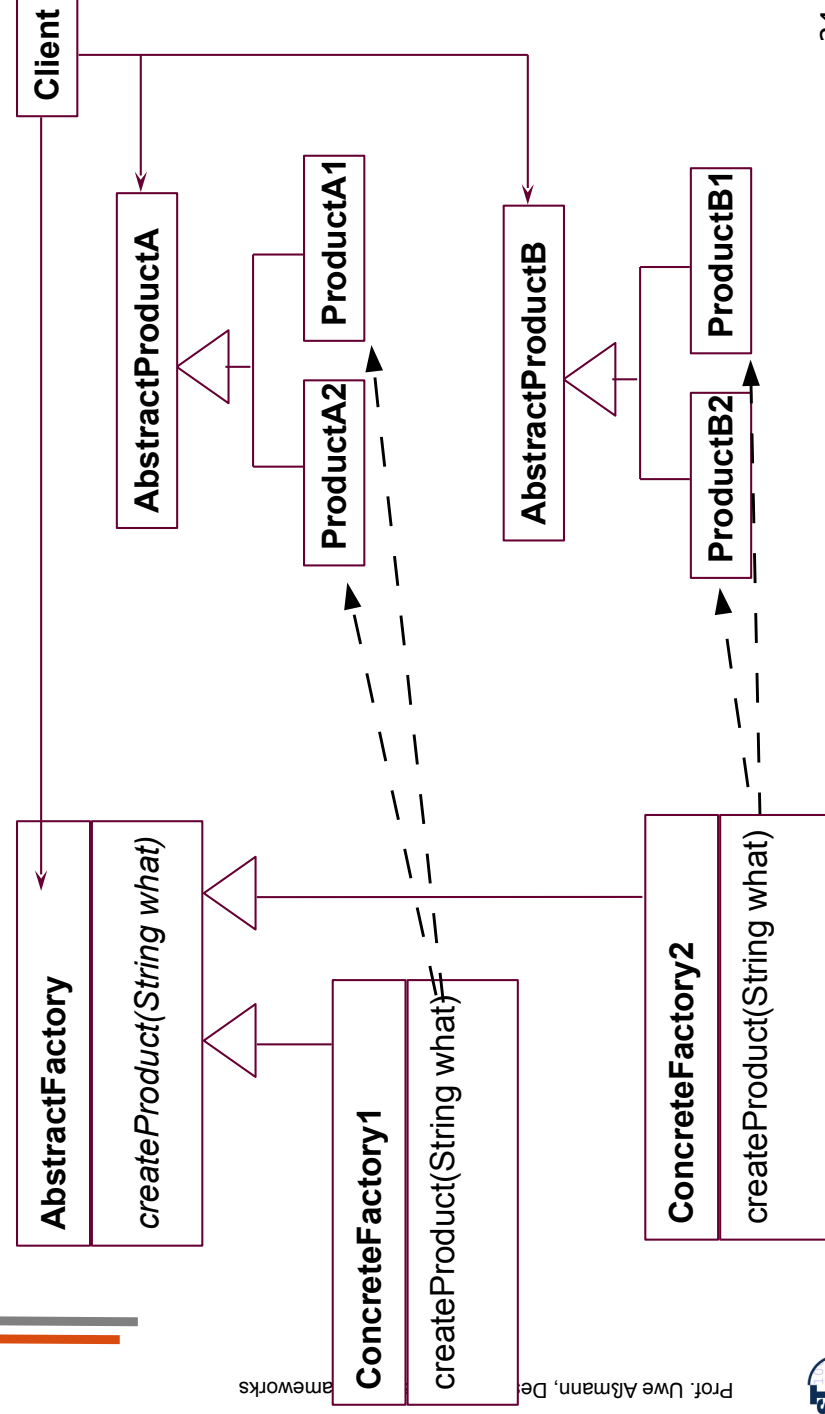
Structure for Prototyping Factory

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 .....
    }
}
```
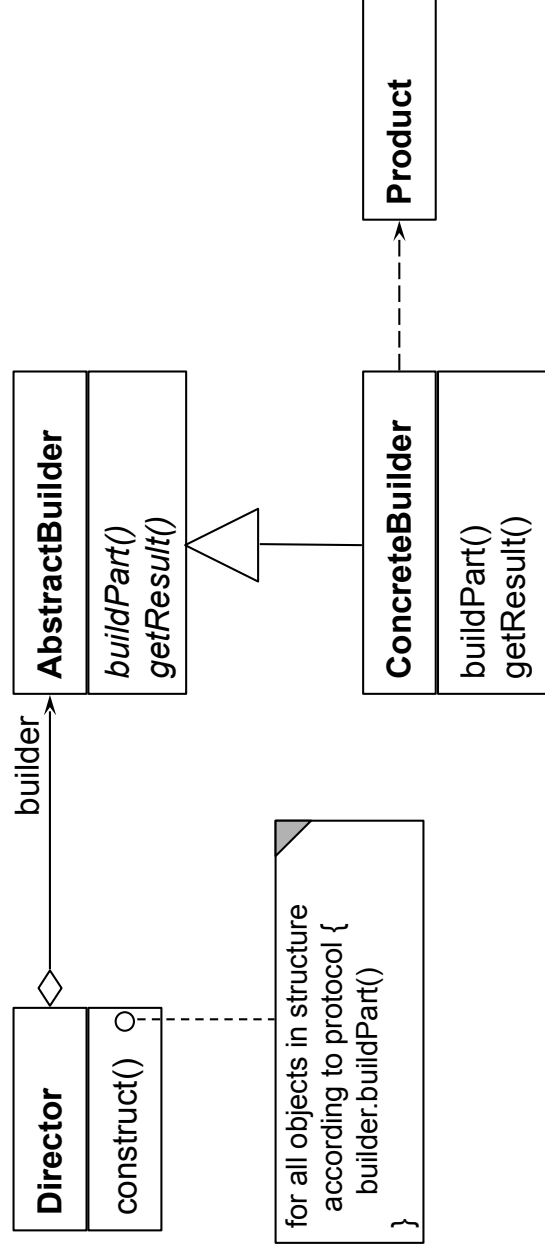
Structure for Interpretive Factory

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

---

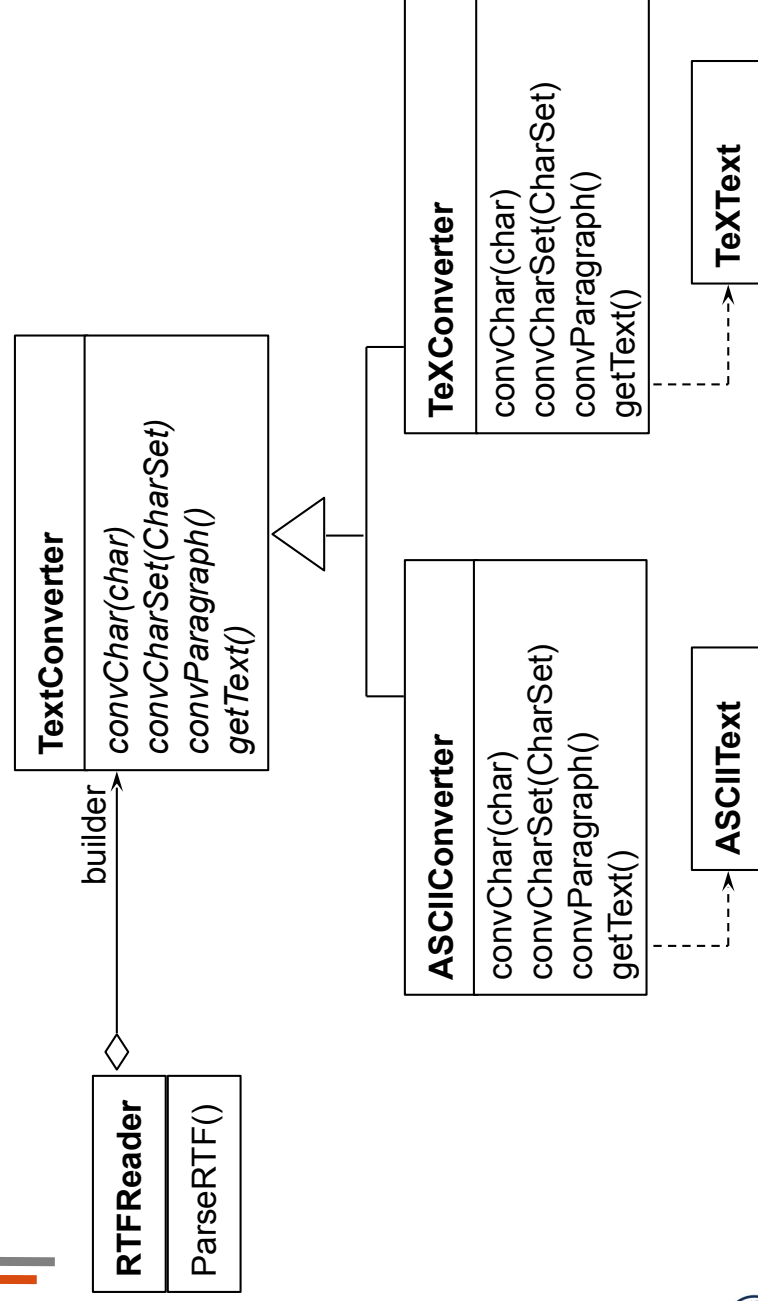# 3.3 Builder (Factory with Protocol, Structured Factory)

Structure for Builder

▲ The Builder is a Factory that produces a *structured* product (a whole with parts)
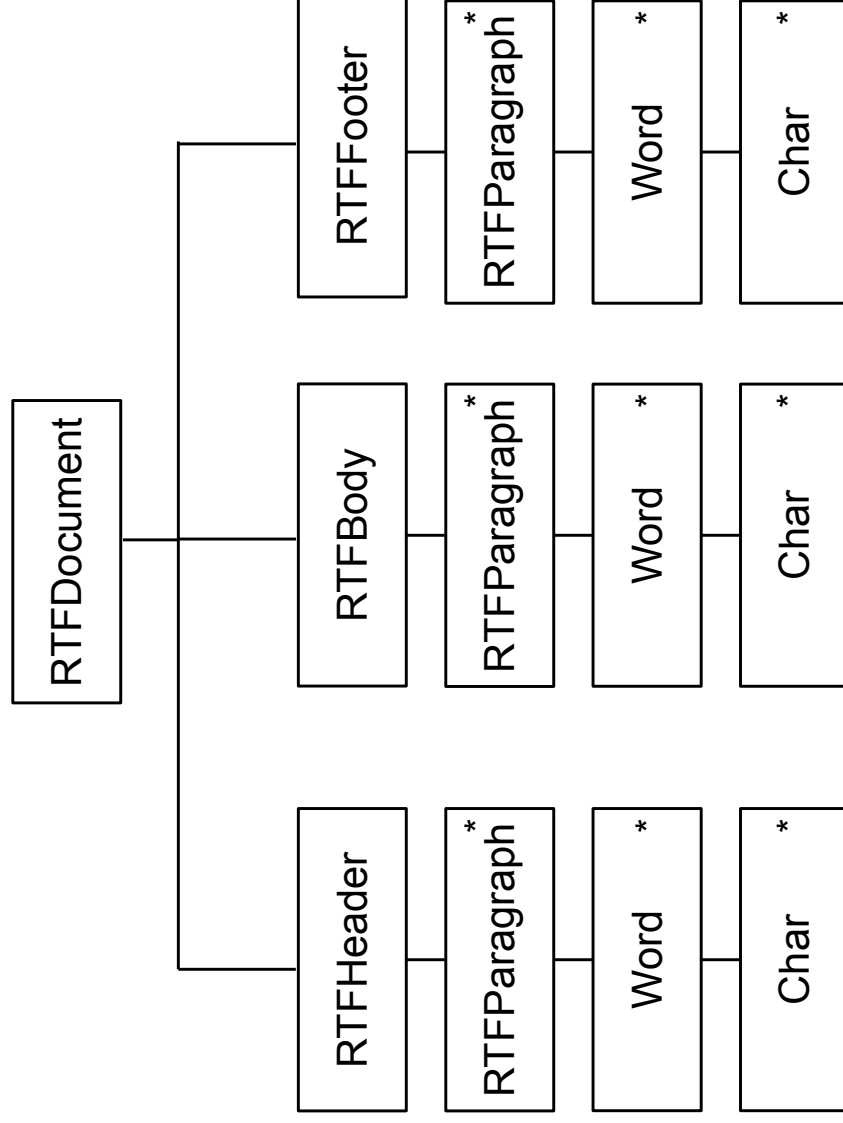
  ▪ e.g., a business object or product data of a PDM

Director
construct()

builder

AbstractBuilder
buildPart()
getResult()

ConcreteBuilder
buildPart()
getResult()

Product

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

Example Builder

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

RTFReader
ParseRTF()

builder

TextConverter
convChar(char)
convCharSet(CharSet)
convParagraph()
getText()

ASCIIConverter
convChar(char)
convCharSet(CharSet)
convParagraph()
getText()

TeXConverter
convChar(char)
convCharSet(CharSet)
convParagraph()
getText()

ASCIIText

TeXText

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

Diagram:
- RTFDocument
  - RTFHeader → RTFParagraph * → Word * → Char *
  - RTFBody → RTFParagraph * → Word * → Char *
  - RTFFooter → RTFParagraph * → Word * → Char *

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
  - JSP, 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, regular expressions, or JSP
  - 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

The End