Design Patterns (Entwurfsmuster)

Ulrich.Schaefer@dfki.de

History

- 1960ies and before: dark age of programming: ALGOL, COBOL, FORTRAN,...
- 1970ies: structured programming paradigm: use subroutines, data types: Pascal, Modula
- 1980ies: object-oriented programming (OOP) paradigm: (additionally) use objects, inheritance, encapsulation, polymorphism: Smalltalk, C++
- 1990ies: there are recurring patterns in OOP that one should be aware of when designing new code

The Gang of Four

- 1977: Christopher Alexander et al: *A Pattern Language* (architecture, not computer science!)
- 1995: Gamma, Helm, Johnson and Vlissides: Design Patterns – Elements of Reusable Software (GoF/"Gang of Four" book)
- describe most frequent patterns, their purpose, define basic methods, classes, structures, dependencies

Design Patterns

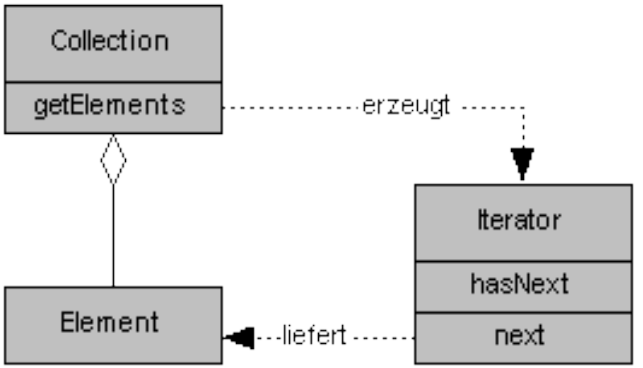
- "Design patterns are recurring solutions to design problems you see over and over." (Alpert et al.'98)
- "Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development." (Pree, '94)
- "Design patterns describe how objects communicate without become entangled in each other's data models and methods." (Cooper, '98)
- "A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it." (Buschmann, et. al. 1996)

- today, hundreds of patterns have been proposed, ranging from very simple to very complex ones
- there is no 'standard', only common sense
- independent of a programming language
- most patterns are not part of a programming language unlike structured programming or OOP
- but pattern implementations differ depending on programming language

- Pattern ≠ Class (in general)
  - some are trivial (single method)
  - some are part of the programming language
  - for some patterns holds pattern = class or interface
  - some can be implemented as independent class library
  - some require complex teamplay of multiple classes
  - names of methods and classes may differ (e.g. according to application context)

Patterns we already know

- **Interface**: part of Java language



- **Iterator**: e.g. `java.util.Collection` interface
- **Factory**: JAXP ParserFactory, TransformerFactory
- **Adapter**: JAXP Source
- many more in AWT / Swing / Java Foundation Classes

Singleton Pattern

- purpose: guarantee existence of a single object, e.g., a server, window manager, printer spooler
- declare constructor private to prevent it from being called (may throw exception instead)
- define getInstance() to return instance
- may be extended to create a limited number of instances ("Fewton", "Oligoton")

Singleton – Example


```
public class Singleton {
  private static Singleton instance = null;

  public static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }

  private Singleton() { } // hide constructor
}
```

Immutable Pattern

- purpose: guarantee that an object cannot be modified
  - when threads should not concurrently modify an object
  - share the same object in multiple references, example: `java.lang.String`
- may be declared final to prevent modification by methods introduced in subclasses

Immutable Pattern – Example

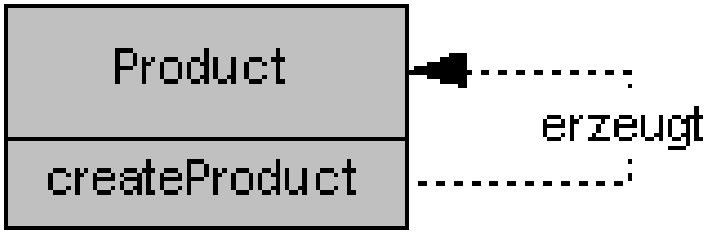

```
public class Immutable {  // make it final to be safe
  private int      value1;
  private String[] value2;    // hide

  public Immutable(int value1, String[] value2) {
    this.value1 = value1; // doesn't need to be cloned
    this.value2 = (String[])value2.clone();   }

  public int getValue1() {
    return value1; }

  public String getValue2(int index) {
    return value2[index]; }
}
```

- purpose: delegate object creation to **subclasses,** let them decide which object to return and how to create it

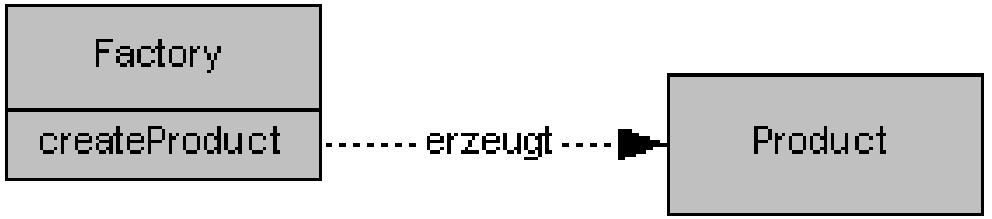


Factory Method Pattern – Example


```
public class Icon {
  private Icon() { } // hide constructor

  public static Icon loadFromFile(String name) {
    Icon ret = null;
    if (name.endsWith(".gif")) {
       ret = new GifIcon(name);
    } else if (name.endsWith(".jpg")) {
       ret = new JpegIcon(name);
    } else if (name.endsWith(".png")) {
       ret = new PngIcon(name);
    }
    return ret;
  }
}
```

- purpose:
  - generate complex objects from a configuration (parameters; e.g. color, engine, wheel type of a car)
  - return potentially different instances
  - provide, but hide multiple implementations



- cf. SAXParserFactory, TransformerFactory in JAXP

Factory example

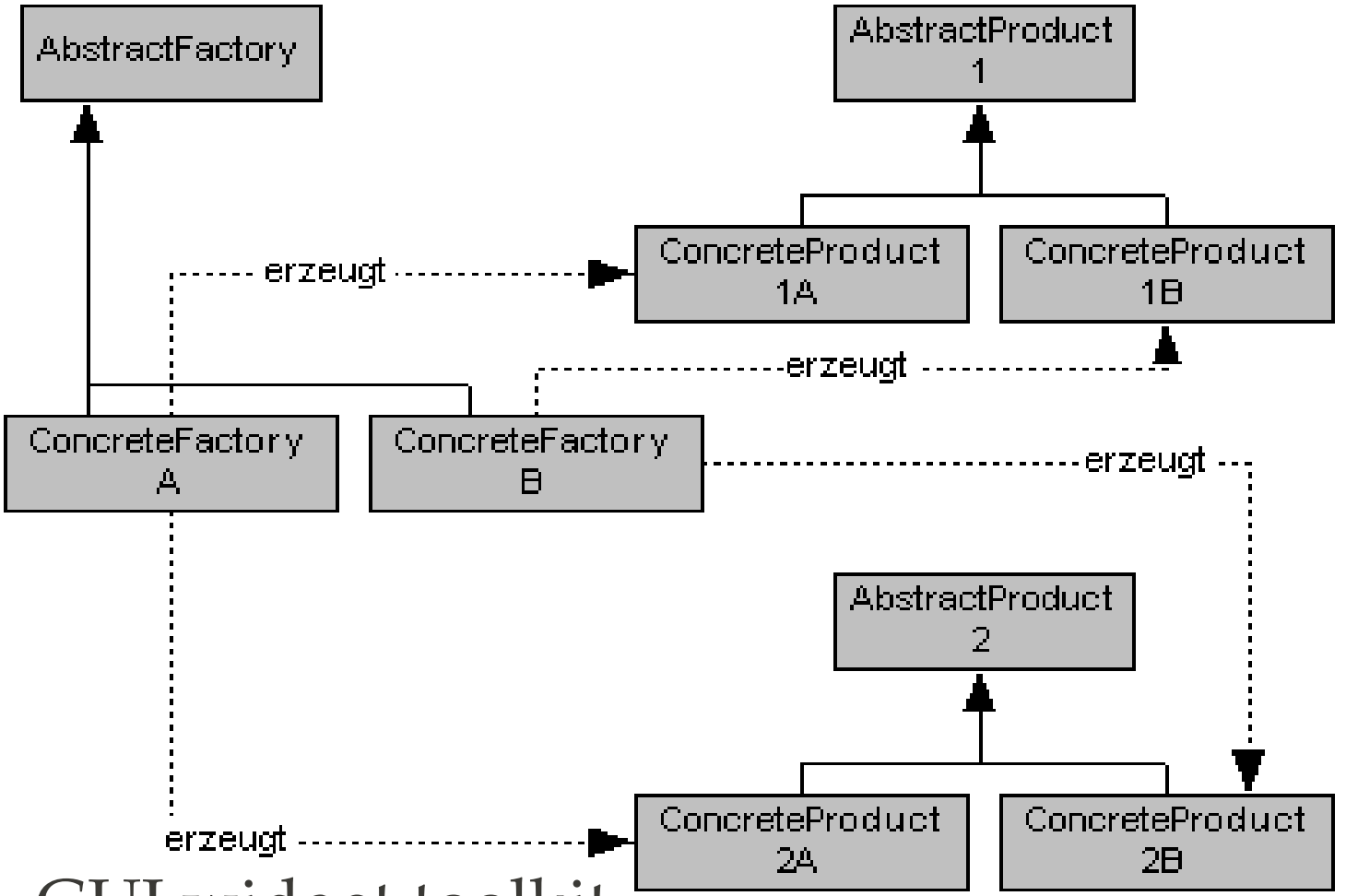

```
public class TypedFeatStructFactory {
  public TypedFeatStructFactory(TypeHierarchy th) {
     // create TFS factory for a given type hierarchy
  }

  public TypedFeatStruct createFromXmlFile(File f) {
    // create TFS using XML parser
  }

  public TypedFeatStruct createFromTextString(String s){
    // create TFS using 'ASCII' (javaCC) parser
  }
}
```

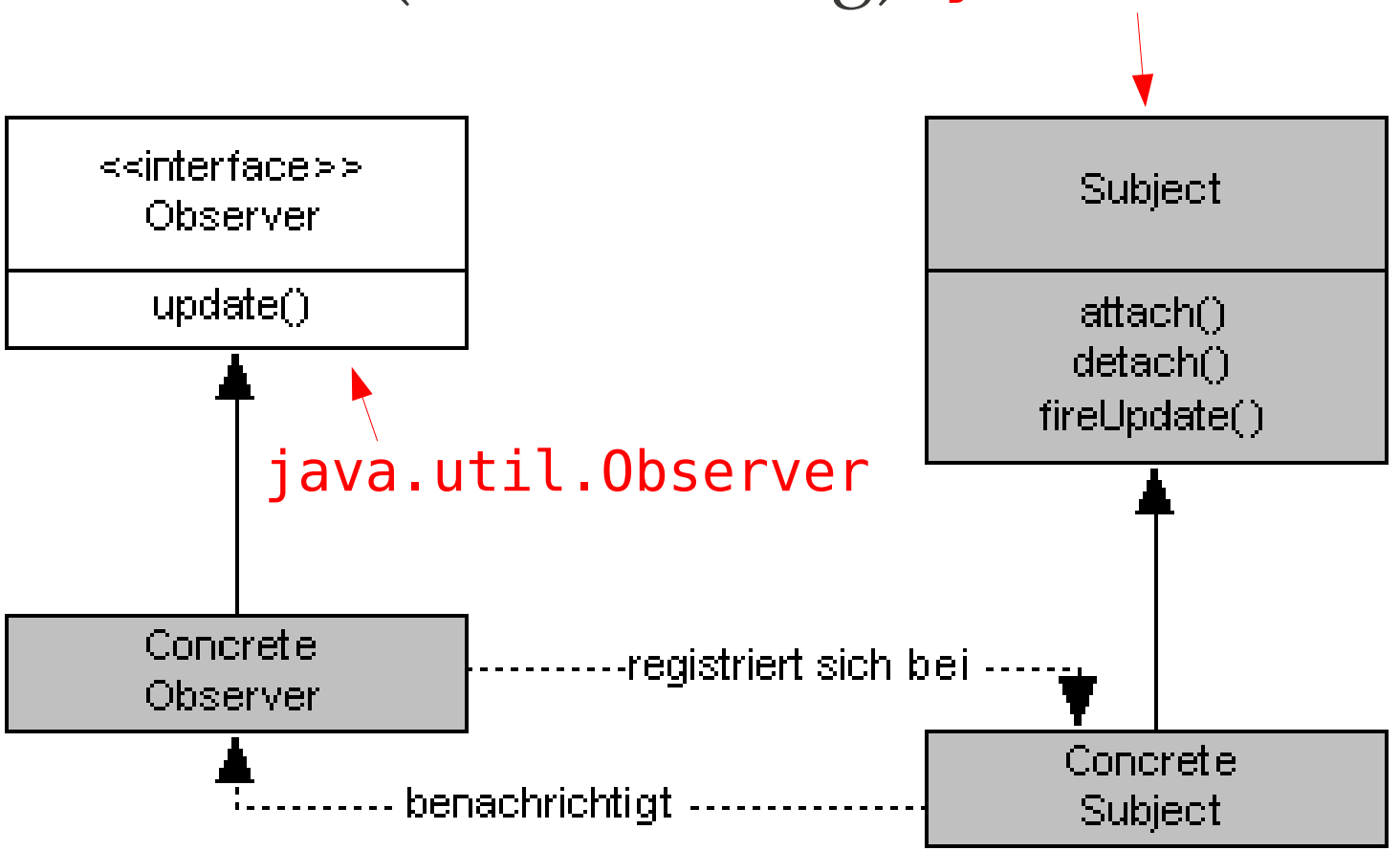
Abstract Factory Pattern ('Toolkit')

- one level of abstraction higher than Factory



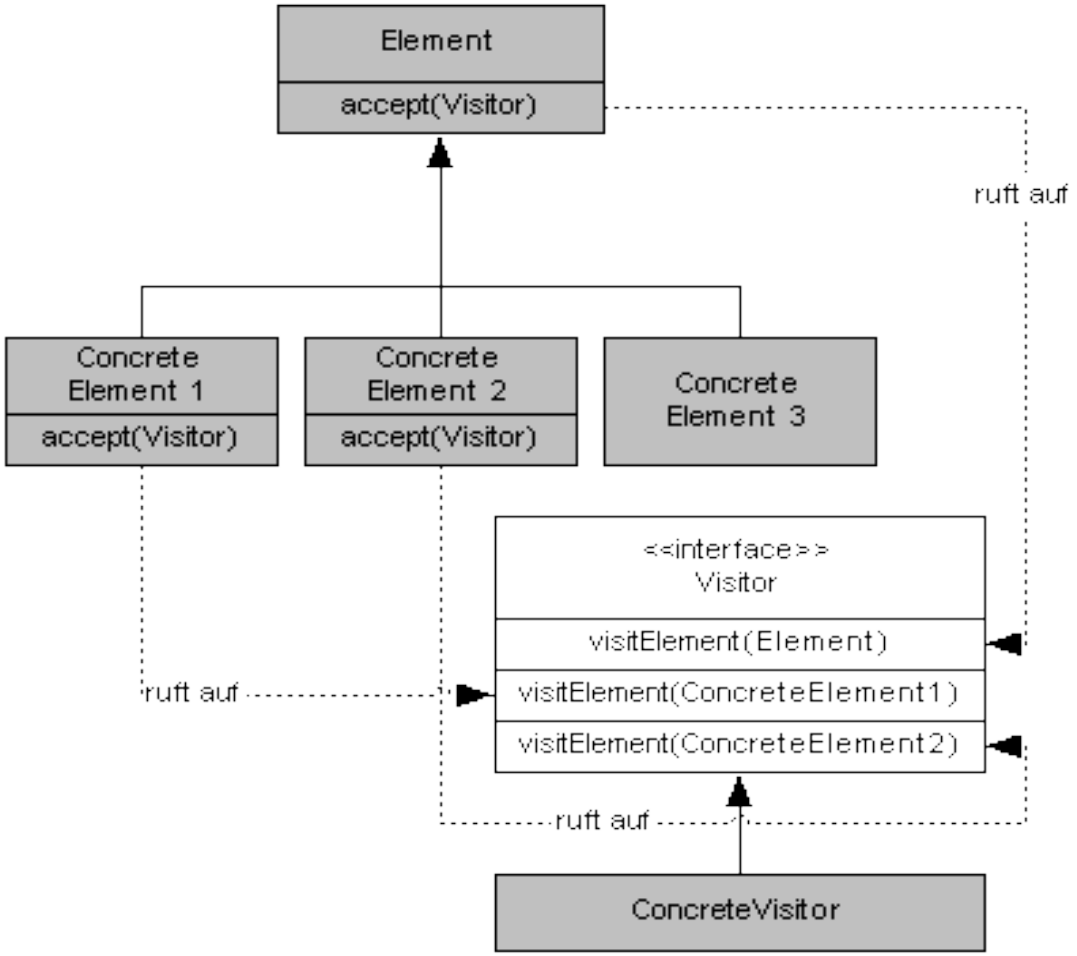
- e.g. GUI widget toolkit

- notify other objects of changes, e.g. updating GUI elements (AWT, Swing)



Visitor Pattern

- encapsulate operations on elements in an object



Pattern Types

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

Pattern types I: **Creational Patterns (1)**

- help creating objects – adding flexibility in deciding which objects need to be created for a given case, e.g.,
  - **Factory method**, (**Abstract**)**Factory**
  - **Singleton**
  - **Prototype**: construct by copying example object ('Chinese factory')
  - **Builder**: separate construction of a complex object from its representation (same builder can produce different representations)

- **Object Pool**: manage the reuse of objects when a type of object is expensive to create or only a limited number of objects can be created.

  A generic implementation can be found in http://jakarta.apache.org/commons/pool

- help composing groups of objects into larger structures, e.g.,
  - **Adapter**: change the interface of one class to that of another one (e.g. javax.xml.transform.Source)
  - **Composite**: collection of objects (recursively)
  - **Decorator**: modify the behavior of individual objects without having to create a new derived class
  - **Facade**: provide a simple interface hiding different complex interfaces (e.g., ODBC/JDBC)
  - **Proxy**: control an object by a representative (surrogat)

- help defining communication between objects and how the flow is controlled in a complex program, e.g.,
  - **Command**: encapsulate commands in objects
  - **Observer**: define the way a number of classes can be notified of a change
  - **Visitor**: encapsulate operations on elements of an object as another object
  - **Mediator**: simplify communication between objects by introducing another object that keeps coupling

Pattern Types III: **Behaviorial Patterns (2)**

- **Strategy**: abstract from algorithms (e.g., in a context), make them interchangeable (cf. AWT Layout Manager, Swing Look & Feel, Sorting algorithms)
- **Chain of Responsibility**: pass requests of an object not directly to the recipient, but through a chain of requests from object to object, until an appropriate recipient is found

  A generic implementation can be found in http://jakarta.apache.org/commons/chain

- how to know which design pattern(s) to use?
  - experience
  - intuition
  - discussion
  - (re-)implementation
- design patterns provide a common language when discussing software design and implementation with co-developers
- help to prevent (design) errors

Literature

- Gamma, Helm, Johnson, Vlissides: *Design-Patterns - Elements of Reusable Object-Oriented Software („GoF book“)*
- Chapter 10.4 in Guido Krüger: *Handbuch der Java-Programmierung* (http://www.javabuch.de) (*diagrams)
- Cooper: *The Design Patterns Java Companion* (PDF downloadable), with many Swing examples
- Grand: *Patterns in Java* (additional patterns)
- Design Patterns in Java – *Reference and Example site*
- Wikipedia: *Design_pattern_(computer_science)*