Java for Advanced Programmers

Design Patterns

Bernd Kiefer
Jörg Steffen

January 6, 2023

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 ("Gang of Four" / GoF 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)

Design Patterns

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

Design Patterns

Pattern $\neq$ 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. in the `java.util.Collection` interface
- Strategy
- Factory / FactoryMethod
- Prototype

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) {
      // lazy (late) initialization
      instance = new Singleton();
    }
    return instance;
  }

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

Immutable Pattern

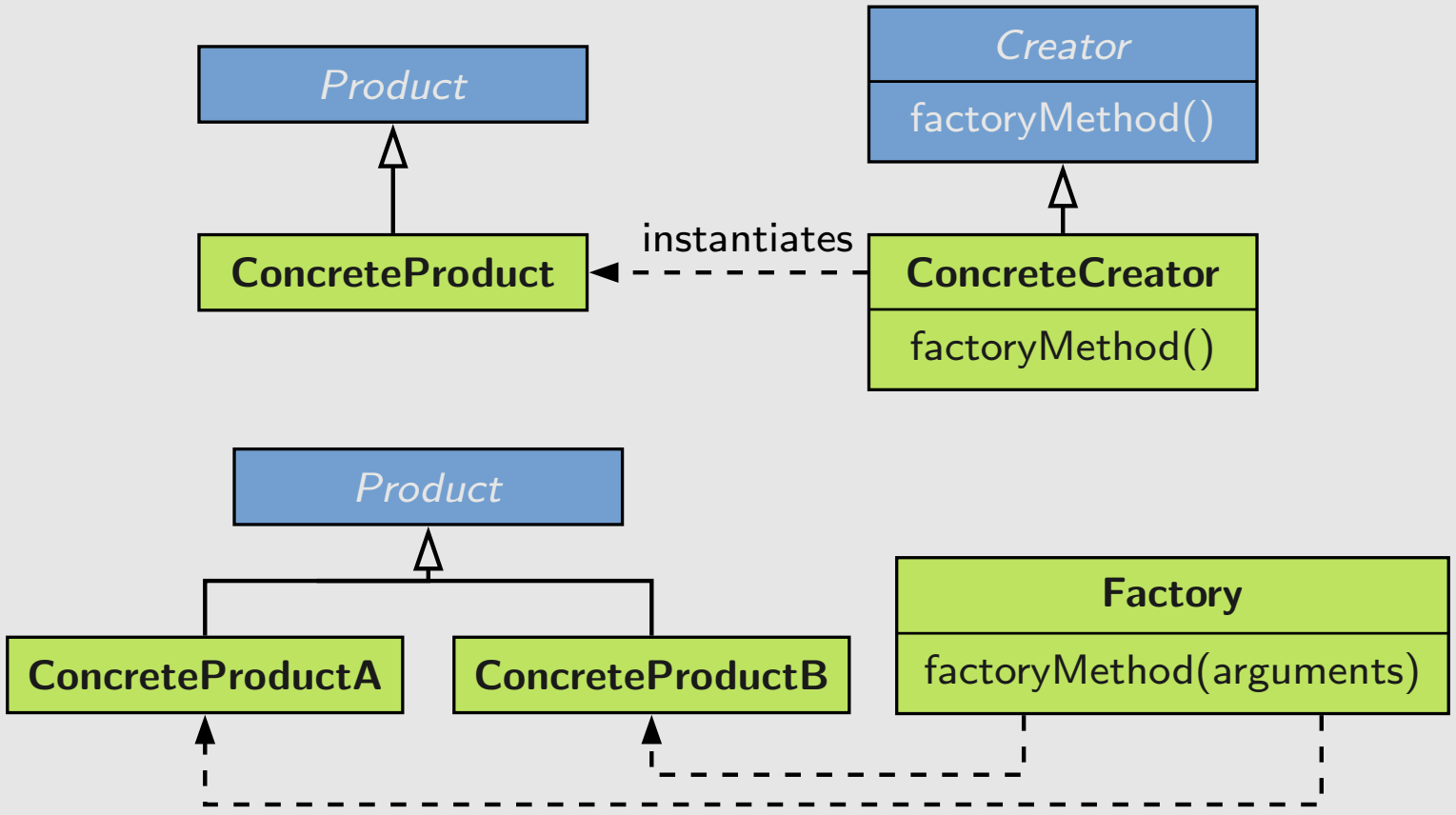
- purpose: guarantee that an object cannot be modified
  - when threads should not concurrently modify an object
  - share the same object in multiple references, example: `java.lang.String`
- may be declared `final` to prevent modification by methods introduced in subclasses
- see also the `Collections`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

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]; }
}
```

Factory / Factory Method Pattern

Delegate object creation to subclasses, let them decide which object to return and how to create it



Factory Pattern – Example

- 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

```
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;
  }
}
```

Factory Example

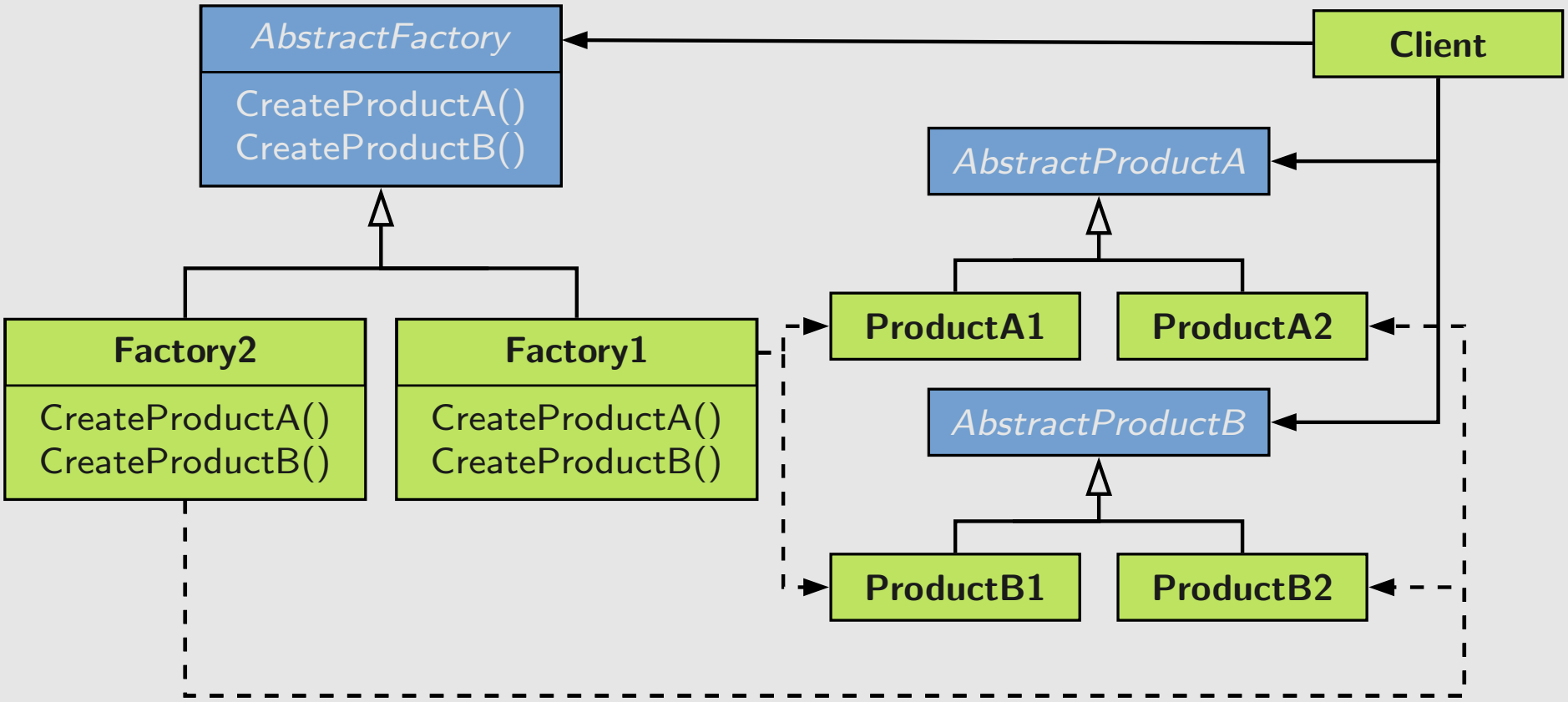
Programmable Calculator / Spreadsheet

```
f(x) := x * (2 + x) - g(x)
```

- (abstract) `Expression` class with subclasses `Mul`, `Add`, `Funcall`, ...
- When an expression is parsed , call `ExprFactory` with operator symbol to get back an object of the proper `Expression` subtype
- Work out "programmable calculator with functions" as example

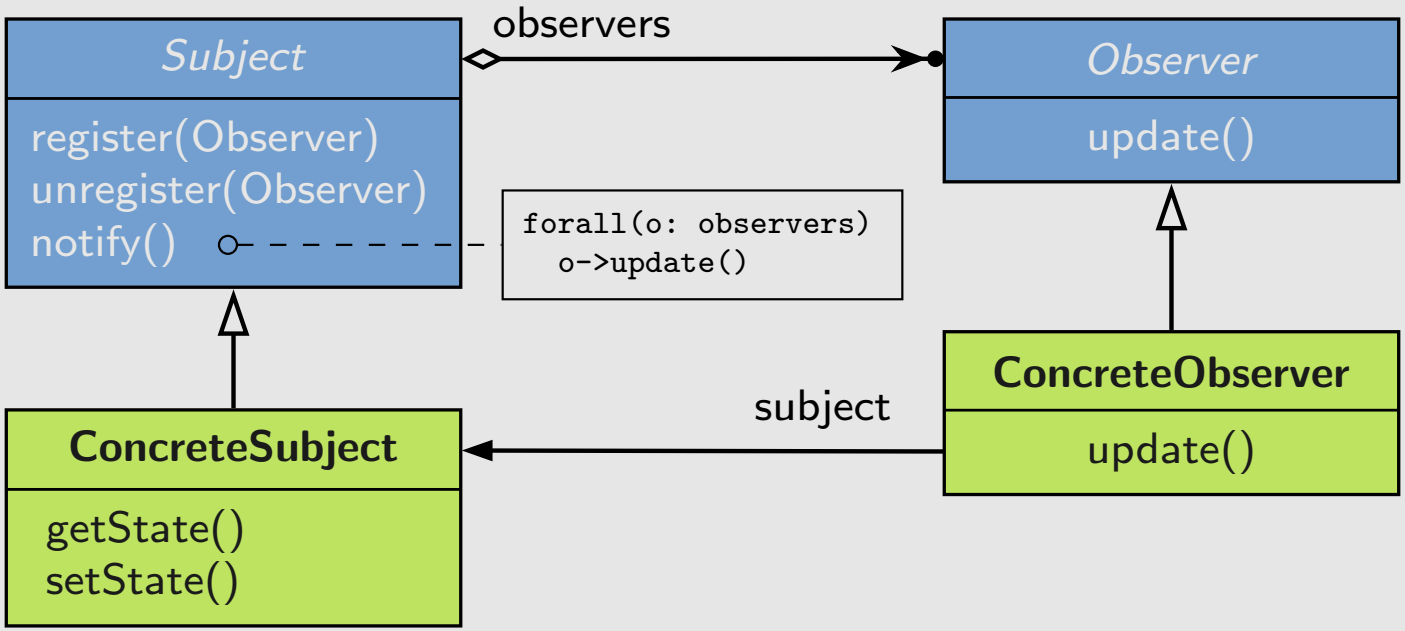
Abstract Factory Pattern ('Toolkit')

Add one level of abstraction to Factory (Window toolkit)



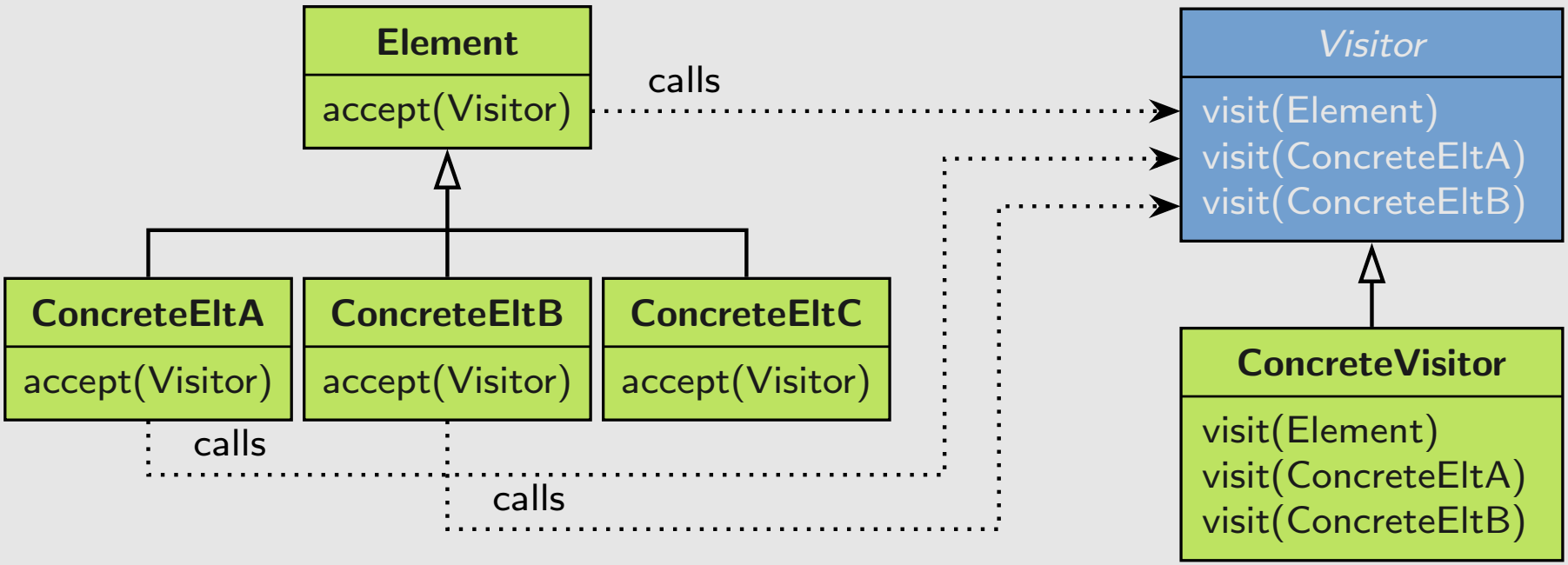
Observer Pattern

Notify other objects of changes, e.g. updating GUI elements



Visitor Pattern

Encapsulate operations on elements in an object



Pattern Types

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

Creational Patterns

help creating objects – adding flexibility in deciding which objects need to be created for a given case

- 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 creation is expensive or only a limited number of objects can be created. A generic implementation can be found in `http://commons.apache.org/pool/`

Structural Patterns

Composing groups of objects into larger structures

- 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)

Behaviorial Patterns (1)

define communication between objects and how the flow is controlled in a complex program

- 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

Behaviorial Patterns (2)

define communication between objects and how the flow is controlled in a complex program

- 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 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://commons.apache.org/chain/`

Using Design Patterns

- 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 11.4 in Krüger & Hansen: 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: `http://en.wikipedia.org/w/index.php?title=Design_pattern_(computer_science)`