Creational Patterns
Singleton, Prototype, Abstract Factory

Sebastian Haubner

Lehrstuhl für Software Engineering
Friedrich-Alexander-Universität Erlangen-Nürnberg

27.04.2016

Gliederung

Gliederung

Erzeugungsmuster – Was ist das?

Erzeugungsmuster ...

- klassifizieren die Erzeugung von Objekten
- kapseln die konkreten Klassen eines Systems
- verbergen konkrete Erzeugungsprozesse dieser Klassen

Konsequenzen

- Erhöhte Flexibilität *wer wann was wie* erzeugt.
- System wird unabhängiger von der Zusammenstellung und Erzeugung der verwendeten Objekte
- Es wird lediglich mit den abstrakten Schnittstellen gearbeitet

Typen von Erzeugungsmustern

Klassenbasiert

Bei klassenbasierten Mustern wird Vererbung verwendet

- Factory Method

Objektbasiert

Bei objektbasierten Mustern wird der Erzeugungsprozess an andere Objekte delegiert

- Singleton
- Prototype
- Abstract Factory
- Builder

Typen von Erzeugungsmustern

Klassenbasiert

Bei klassenbasierten Mustern wird Vererbung verwendet

- Factory Method

Objektbasiert

Bei objektbasierten Mustern wird der Erzeugungsprozess an andere Objekte delegiert

- Singleton
- Prototype
- Abstract Factory
- Builder

Gliederung

Definition

Zweck

Sicherstellen, dass von einer Klasse höchstens eine Instanz erzeugt wird, für die ein globaler Zugriffspunkt existiert.

Motivation

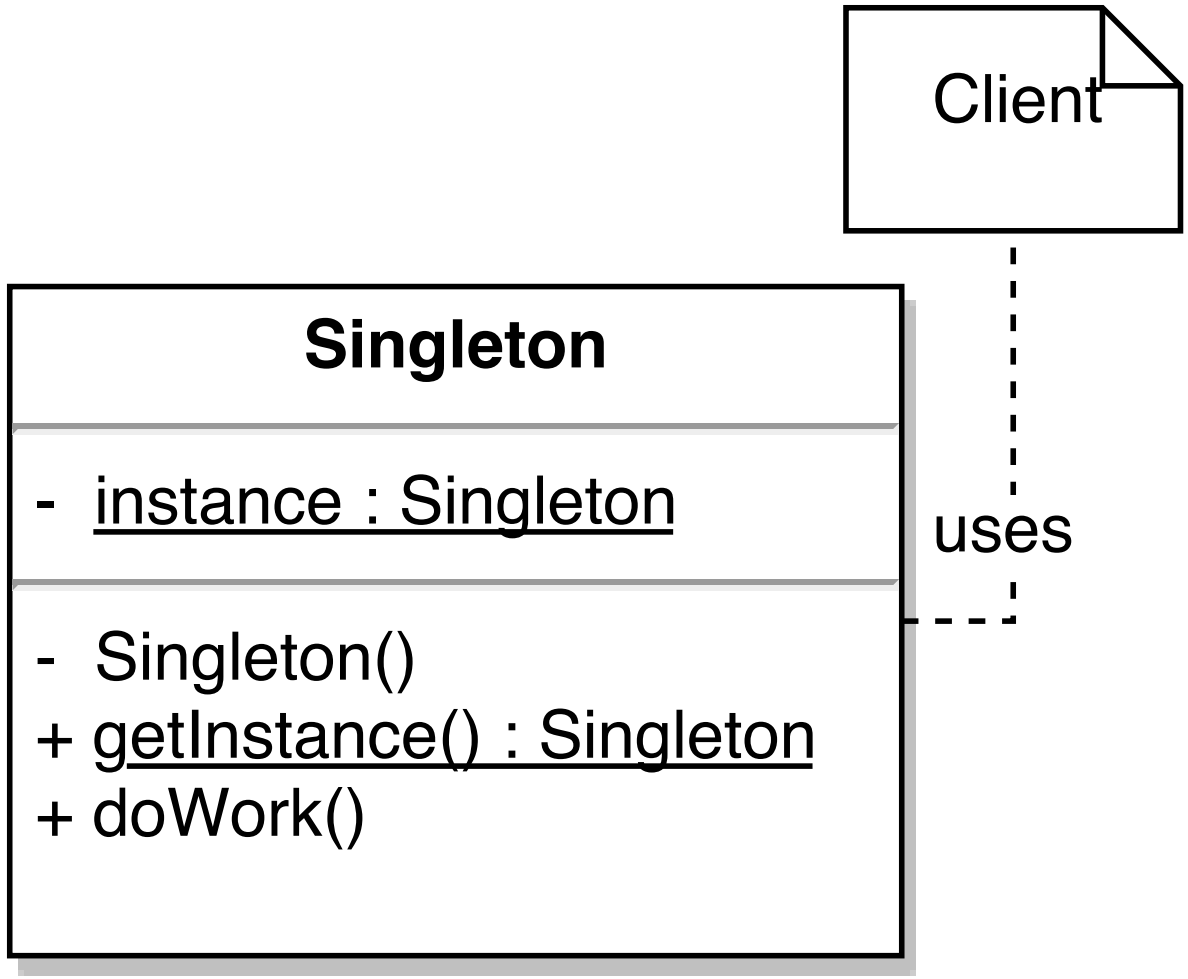
- Zugriff auf eine Resource die nur einmal existiert (z.B. der Applikations Log)
- Lazy Initialization

Sinnvoll, wenn ...

- nur eine Instanz mit globalem Zugriffspunkt erlaubt ist
- Instanz mit Unterklassen erweiterbar sein soll

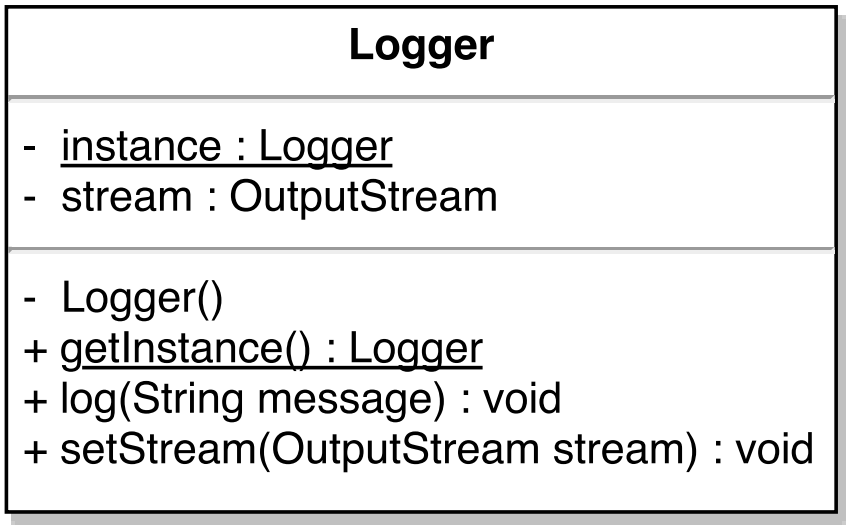
Klassendiagramm

1. privater Konstruktor
2. statische getInstance() Methode zum Zugriff



Beispiel - Logger

1. Globaler Logger
2. Default Stdout



Beispiel - Logger - Implementierung

```
1  public class Logger {
2      private static Logger instance;
3      private Writer writer;
4
5      private Logger(){
6          writer = new BufferedWriter(
7                  new OutputStreamWriter(System.out));
8          log("[Logger instance created]");
9      }
10     public static Logger getInstance(){
11         if (instance == null) {
12             instance = new Logger();
13         }
14         return instance;
15     }
16     public void log(String message){
17       //Logging logic
18     }
19 }
```

Beispiel - Logger - Problem

```
1 ExecutorService executor = Executors.newFixedThreadPool(10);
2 for(int i = 0; i < 10; i++) {
3     executor.submit(() -> {
4         String threadName = Thread.currentThread().getName();
5         Logger.getInstance().log("Hello from " + threadName);
6     });
7 }
8 executor.shutdown();
```

```
1 [2016-04-21T14:38:46.420] [Logger instance created]
2 [2016-04-21T14:38:46.420] [Logger instance created]
3 [2016-04-21T14:38:46.419] [Logger instance created]
4 [2016-04-21T14:38:46.421] [Logger instance created]
5 [2016-04-21T14:38:46.425] Hello from pool-1-thread-7
6 [2016-04-21T14:38:46.425] Hello from pool-1-thread-10
7 [2016-04-21T14:38:46.420] [Logger instance created]
8 ...
```

Beispiel - Logger - Implementierung - Threadsafe

```
1  public class Logger {
2      private static volatile Logger instance;
3      private Writer writer;
4      //the constructor gets called when the instance is used
5      private Logger(){
6          writer = new BufferedWriter(
7                  new OutputStreamWriter(System.out));
8          log("[Logger instance created]");
9      }
10     public static Logger getInstance(){
11         if (instance == null) {
12             synchronized (Logger.class){
13                 if (instance == null)
14                     instance = new Logger();
15             }
16         }
17         return instance;
18     }
19     public void log(String message){
20       //logging logic
21     }
22 }
```

Beispiel - Logger - Implementierung - Alternativ

Umsetzung mittels Java-Enum (Java >= 1.5)

- Einfachste Art ein Singleton zu erzeugen
- Thread-Sicher und Serialisierbar (Enum Eigenschaft)
- Mehrfach Instanziierung ausgeschlossen (auch nicht per Reflection)
- **Enums können nicht abgeleitet werden!**

Beispiel - Logger - Implementierung - Alternativ

```
1  public enum Logger {
2      //This is our instance variable
3      INSTANCE;
4      private Writer writer;
5      //the constructor gets called when the instance is used
6      Logger(){
7          writer = new BufferedWriter(
8                  new OutputStreamWriter(System.out));
9      }
10     public void setWriter(Writer writer){
11         this.writer = writer;
12     }
13     public void log(String message){
14         try {
15             String date = LocalDateTime.now().toString();
16             writer.write(
17                     "["
18                     + date + "] - "
19                     + message + "\n");
20             writer.flush();
21     }
22 }
```

Beispiel - Logger - Benutzung

```
1 public static void main(String[] args) {
2     //With this call to Logger.INSTANCE the private constructor is ca
3     Logger.INSTANCE.log("Test (Stdout)");
4     //Set a new writer
5     Logger.INSTANCE.setWriter(new BufferedWriter(
6             new OutputStreamWriter(System.err)));
7     //write to Stderr
8     Logger.INSTANCE.log("Test (Stderr)");
9 }
```

```
1 [2016-04-19T13:04:43.019] - Test (Stdout)
2 [2016-04-19T13:04:43.031] - Test (Stderr)
```

Erweiterbar durch Unterklassen

```
1  protected Logger() throws InstantiationException {
2      synchronized (Logger.class) {
3          if (instance != null)
4              throw new InstantiationException("error msg");
5      }
6  }
7
8  public static void init() throws InstantiationException {
9      instance = new ExtendedLogger();
10 }
```

Notwendige Änderungen

- Sichtbarkeit protected (package visible)
- Konstruktor prüft ob Instanz vorhanden
- statische init Methode

Fazit - Vor- / Nachteile

Vorteile

- Einfache Verwendung
- Lazy Initialization
- Einfache Zugriffskontrolle
- Spezialisierung möglich (wenn kein Java Enum :-))

Nachteile

- Verschleiert Abhängigkeiten (unklar ob eine Klasse das Singleton benutzt)
- Verletzt Single-Responsibility Prinzip
- Erschwert Testen durch globalen Zustand
- Probleme bei multi-threading und verteilten Systemen
- Erweiterbarkeit durch Unterklassen problematisch

Fazit - Zusammenfassung

Zusammenfassung

- Durch die vielen Nachteile sehen manche Leute das Singleton als Anti-Pattern
- Bei bedachter Nutzung ist es für bestimmte Szenarien hilfreich
- Verteilte Systeme und multi-threading erschweren die Verwendung

Gliederung

Definition I

Zweck

- Spezifiziert die zu erzeugenden Objekte anhand der Instanz eines Prototyps und erzeugt neue Objekte durch kopieren dieser Instanz

Motivation

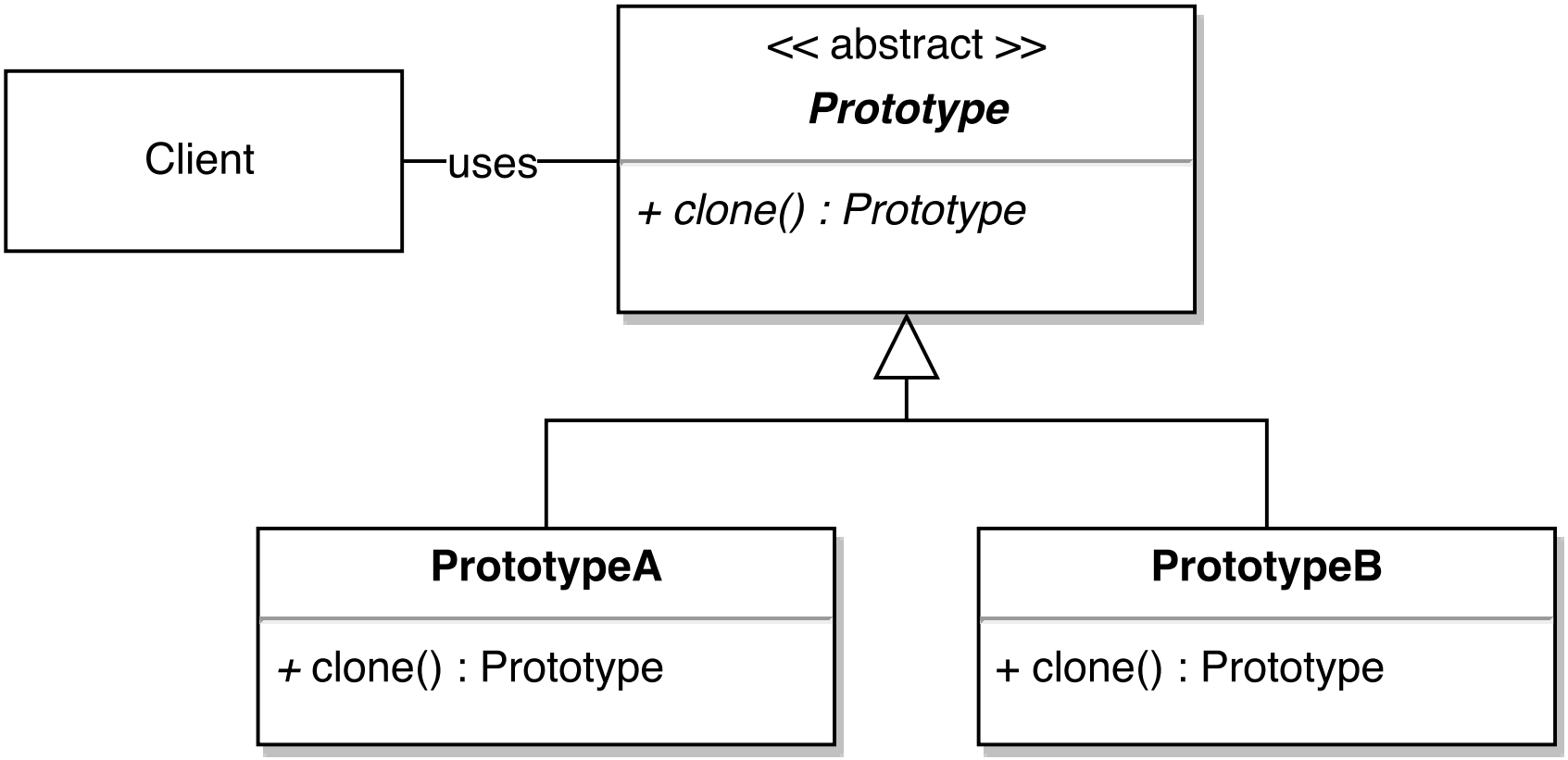
- Vermeidung von teuren Initialisierungsprozessen (zB. Daten über Netzwerkstream)
- Zu instanziierende Klassen werden erst zur Laufzeit bekannt
- Klassen unterscheiden sich nur in wenigen Eigenschaften

Definition II

Sinnvoll, wenn ...

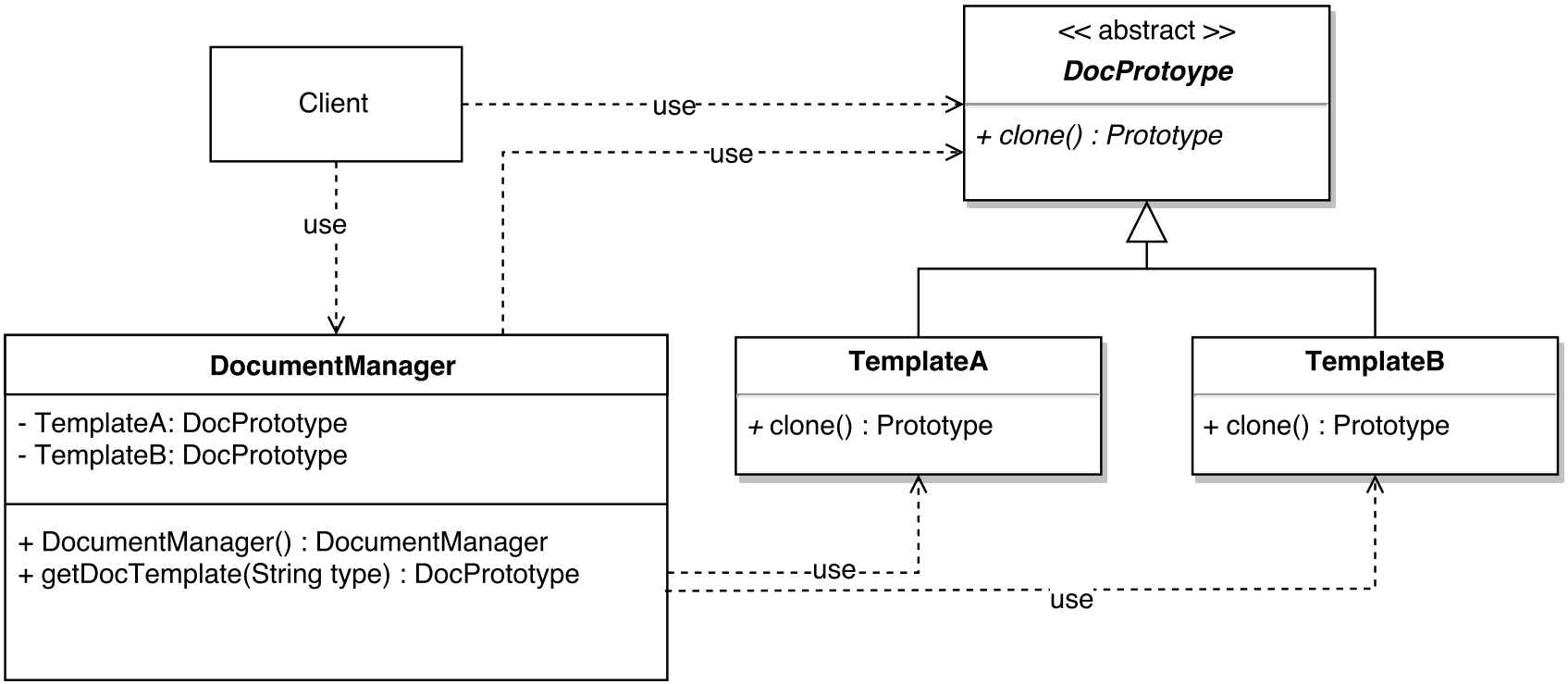
- Klassen zur Laufzeit spezifiziert werden sollen
- vermieden werden soll, eine Hierarchie von Factories parallel zur Hierarchie der Produkte zu erhalten
- es praktischer ist Prototypen zu klonen anstatt sie mit bestimmtem Zustand zu initialisieren

Klassendiagramm



Beispiel - Dokumentenvorlagen

- komplexe Dokumentenvorlagen
- Original soll beibehalten werden
- für die Verwendung wird das Original geklont
- DocumentManager kennt alle Prototypen



Beispiel - Dokumentvorlagen - Implementierung I

```java
//use java Cloneable
public abstract class DocPrototype implements Cloneable {
    @Override
    protected DocPrototype clone()
            throws CloneNotSupportedException {
        return (DocPrototype) super.clone();
    }
}
```

```java
public class TemplateA extends DocPrototype {
    private Date lastModified;
    private String content;
    //clone all fields
    protected DocPrototype clone()
            throws CloneNotSupportedException {
        TemplateA clone = (TemplateA) super.clone();
        clone.lastModified = (Date) lastModified.clone();
        return clone;
    }
    //Getters and setter skipped here
}
```

Beispiel - Dokumentvorlagen - Implementierung II

```
1  public class DocManager{
2      public static final String TEMPLATE_A = "A";
3      private HashMap<String, DocPrototype> prototypes;
4
5      public DocManager(){
6          prototypes = new HashMap<>();
7          //initialize prototype - long running operation
8          TemplateA templateA = new TemplateA();
9          templateA.setContent(getContentFromNetwork());
10         prototypes.put(TEMPLATE_A, templateA);
11     }
12     //Returns a clone from our prototypes
13     public DocPrototype getTemplate(String type)
14                 throws CloneNotSupportedException {
15         switch (type) {
16             case TEMPLATE_A:
17                 return prototypes.get(TEMPLATE_A).clone();
18             default:
19                 throw new Exception("No such template");
20         }
21     }
22 }
```

Beispiel - Dokumentvorlagen - Benutzung

```
1  public static void main(String[] args) {
2      DocManager manager = new DocManager();
3      TemplateA docA1 =
4        (TemplateA)manager.getTemplate(DocManager.TEMPLATE_A);
5      TemplateA docA2 =
6        (TemplateA)manager.getTemplate(DocManager.TEMPLATE_A);
7      System.out.println(docA1.getContent()
8            + " [modified : " + docA1.getLastModified() + "]");
9      System.out.println(docA2.getContent()
10           + " [modified : " + docA2.getLastModified() + "]");
11 }
```

```
1 TemplateA Content [modified : Tue Apr 19 15:34:09 CEST 2016]
2 TemplateA Content [modified : Tue Apr 19 15:34:09 CEST 2016]
```

DocumentManager verwaltet Prototypen

- Aufwändige Initialisierung nur einmalig
- Zentraler Zugriff auf die Prototypen

Fazit - Vor- / Nachteile

Vorteile

- Schnellere / Einfachere Erzeugung komplexer Objekte
- Erweiterbar zur Laufzeit durch Erzeugen neuer Prototypen
- Reduziert Subklassen (neue “Klassen“ durch Variation von Werten)
- Ermöglicht dynamische Konfiguration (dynamic loading)

Nachteile

- Jede Klasse muss clone() Funktion implementieren (Auch jedes interne Objekt)
- Klonen eines Objektes eventuell aufwändig
- Eventuell zusätzliche Initialisierung nach clone() notwendig

Fazit - Zusammenfassung

Zusammenfassung

- Sinnvoll, wenn sich Klassen in nur wenigen Eigenschaften unterscheiden
- Erspart teure Initialisierungsprozesse (z.B. Netzwerk)
- Wichtig ist, dass geklonte Objekte voneinander unabhängig sein müssen (deep copy). $\rightarrow$ ggf. aufwändig

Gliederung

Definition I

Zweck

- Stellt ein Interface zur Erzeugung von Familien verwandter oder voneinander abhängiger Produkte bereit, ohne deren konkrete Klassen zu spezifizieren.

Motivation

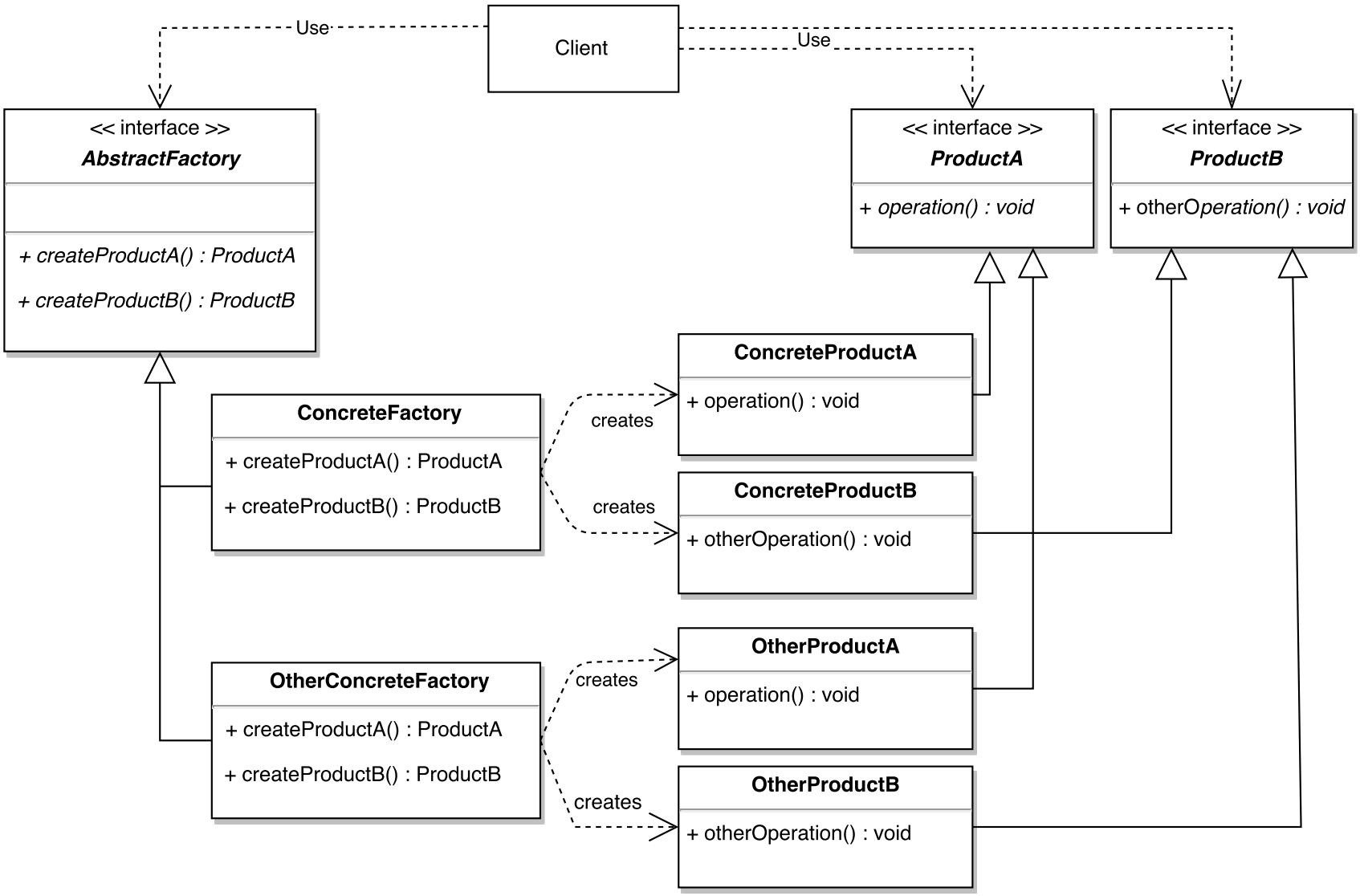
- Verschiedene Objekte müssen in einem Kontext gemeinsam erstellt werden
- System soll losgelöst von konkreten Implementierungen der Produkte einer Fabrik sein
- Konfiguration des Systems mit Zusammenstellungen von Objektgruppen

Definition II

Sinnvoll, wenn ...

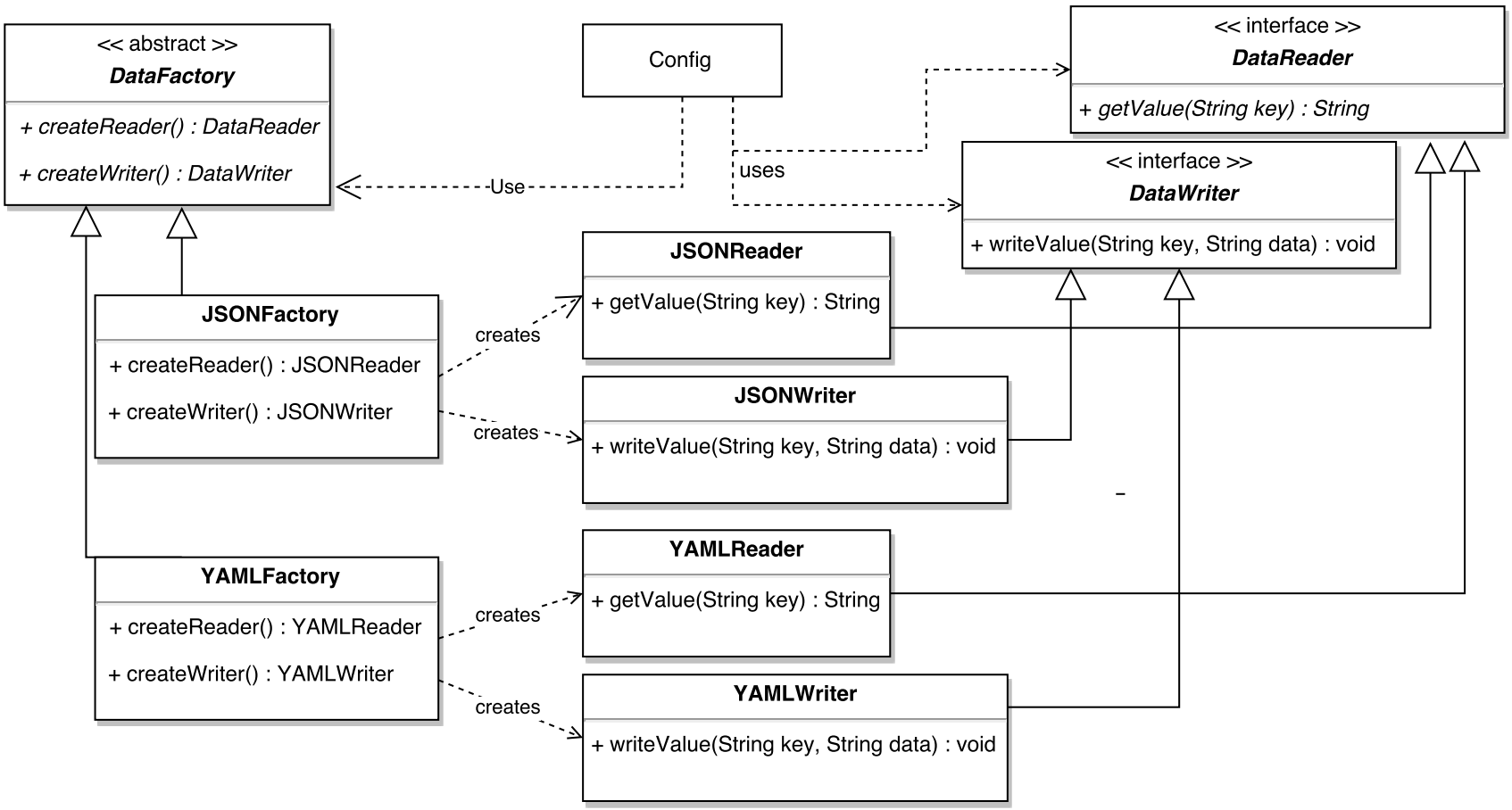
- das System unabhängig von der Darstellung, Zusammenstellung und Erzeugung seiner Produkte sein soll
- eine Konfiguration des Systems mit einer von mehreren Produktfamilien möglich sein soll
- die Konsistenz zusammengehörender Produkte gewahrt werden muss
- eine Bibliothek von Produkten bereitgestellt werden soll, die nur durch Ihre Schnittstellen spezifiziert sind.

Klassendiagramm



Beispiel - Config Reader

- Config in verschiedenen Typen (YAML, JSON, ...)
- Config bekommt Factory übergeben



Beispiel - Config Reader - Implementierung I

```
1 abstract class DataFactory {
2     abstract DataReader createReader();
3     abstract DataWriter createWriter();
4 }
```

```
1 public interface DataReader {
2     String getValue(String key);
3 }
```

```
1 public interface DataWriter {
2     void writeValue(String key, String data);
3 }
```

```
1 public class JSONFactory extends DataFactory {
2     DataReader createReader() { return new JSONReader(); }
3     DataWriter createWriter() { return new JSONWriter(); }
4 }
```

```
1 public class YAMLFactory extends DataFactory {
2     DataReader createReader() { return new YAMLReader(); }
3     DataWriter createWriter() { return new YAMLWriter(); }
4 }
```

Beispiel - Config Reader - Implementierung II

```
1  public class Config {
2      private DataReader reader;
3      private DataWriter writer;
4      //Constructor gets Factory instance
5      public Config(DataFactory dataFactory){
6          reader = dataFactory.createReader();
7          writer = dataFactory.createWriter();
8      }
9      String getConfigData(){
10         return reader.getValue("blub");
11     }
12     void setConfigData(String value){
13         writer.writeValue("bla", value);
14     }
15 }
```

Abstrakte Produktfamilien werden verwendet

- Config Klasse ist unabhängig von der konkreten Implementierung, verwendet die übergebene, konkrete Fabrik

Fazit - Vor- / Nachteile

Vorteile

- Kapseln konkreter Klassen
- Konsistenz durch die Gewährleistung zusammenpassender Produkte
- Konkrete Produkte können wiederverwendet werden
- Einfache Erweiterbarkeit und Austauschbarkeit von Produktfamilien

Nachteile

- Hinzufügen / Ändern eines Produktes (nicht Familie!) erfordert Änderung der Abstract Factory –> Alle konkreten Factories müssen angepasst werden

Fazit - Zusammenfassung

Zusammenfassung

- Die Abstract Factory verwendet oftmals das Factory Method Pattern (eine Factory Method für jedes Produkt)
- Ziel ist die Erzeugung zusammengehörender Produktfamilien
- Konkrete Fabriken sind oftmals als Singleton realisiert
- Die Fabrik kann das Prototype Pattern verwenden und Objekte durch kopieren der verwalteten Prototypen erzeugen

Gliederung

Zusammenhang der vorgestellten Muster

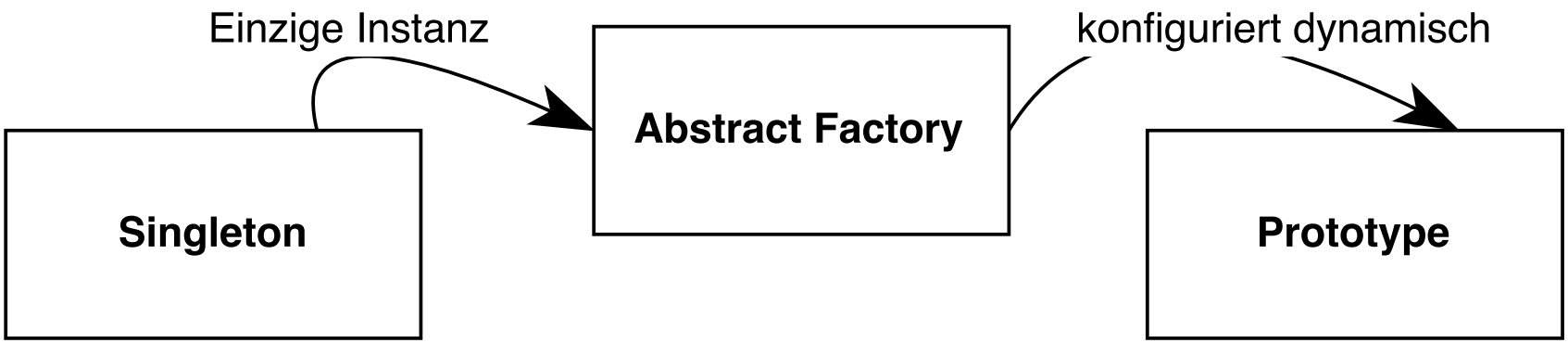


Abbildung : Zusammenhang der vorgestellten Muster - nach [1]

Auswahl des passenden Erzeugungsmusters

- Die vorgestellten Erzeugungsmuster stehen teils in Konkurrenz zueinander
- Auch die kombinierte Verwendung von Pattern ist möglich und in bestimmten Szenarien sinnvoll
- Flexibilität und Komplexität der Muster variiert
- Abwägen der Pattern spezifischen Vor- und Nachteile bei der Auswahl erforderlich

Quellen

[1] Gamma, Helm, Johnson, Vlissides: „Design Patterns - Entwurfsmuster als Elemente wiederverwendbarer Objektorientierter Software“.
1. Auflage, mitp Verlags GmbH, 2015.

[2] Eilebrecht, Starke: „Patterns Kompakt - Entwurfsmuster für effektive Software-Entwicklung“.
3. Auflage, Spektrum Verlag, Heidelberg 2010.

[3] Josh Bloch: „Effective Java Reloaded“
Talk at Google I/O 2008.
https://sites.google.com/site/io/effective-java-reloaded

[4] Alexander Shvets: „Design Patterns Explained Simply“
1.Auflage, sourcemaking.com, 2015