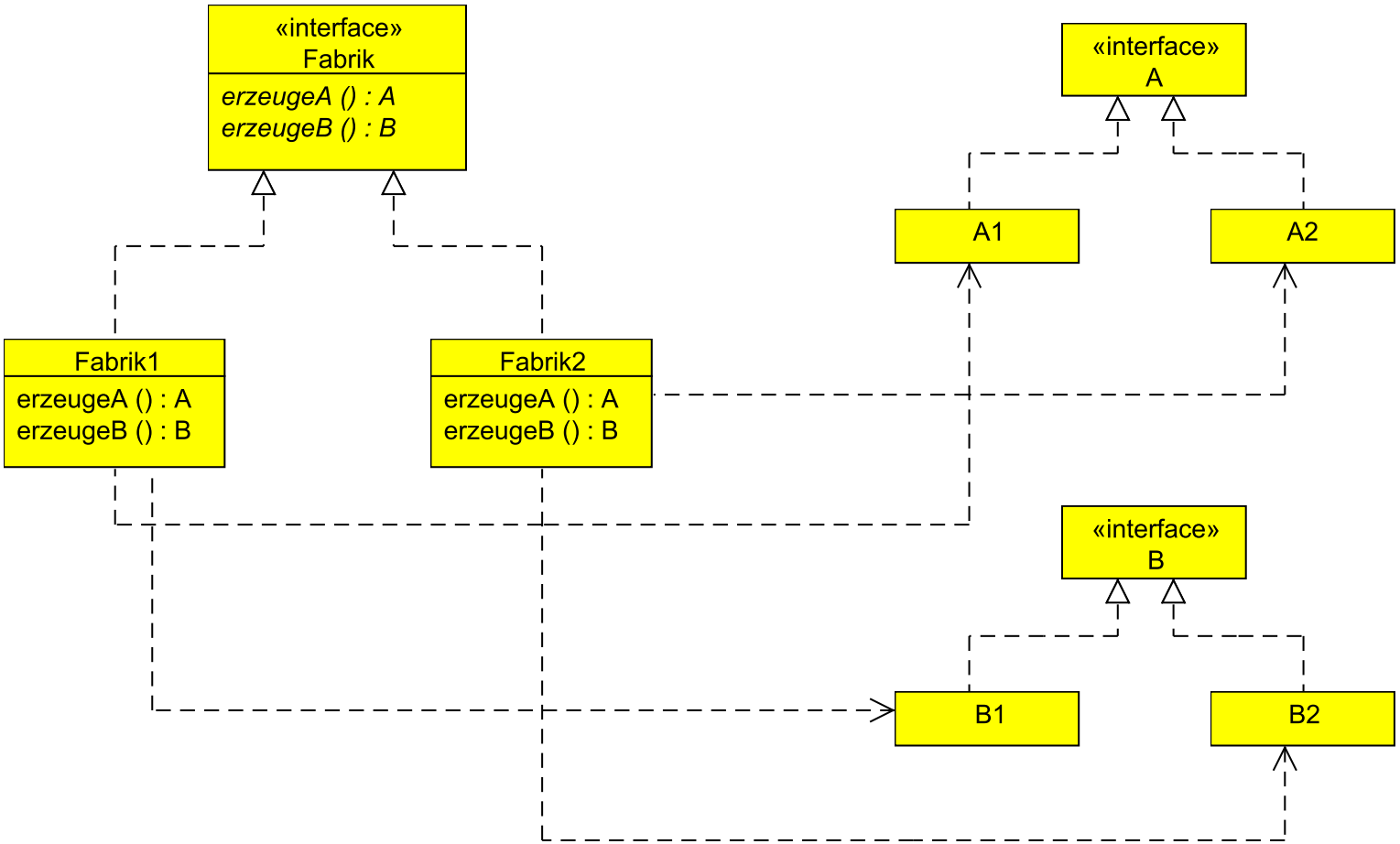
Analyse und Modellierung von Informationssystemen

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2014/2015

Weitere Erzeugungsmuster

Diagramm: Abstrakte Fabrik



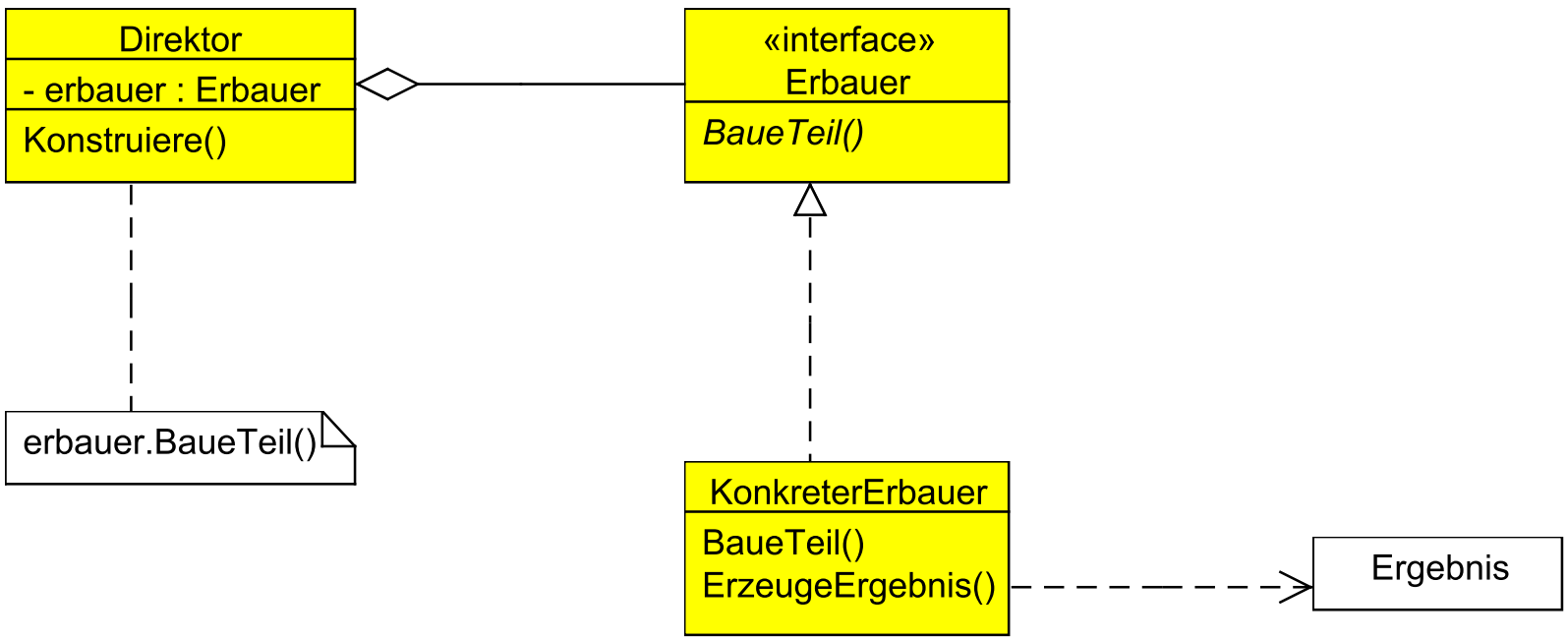
Wesentliche Eigenschaften

Die *abstrakte Fabrik* definiert ein oder mehrere Produkt-Interfaces, die bestimmte Operationen deklarieren. Weiterhin wird ein Fabrik-Interface definiert, das über Fabrikmethoden die Funktionalität zum Erzeugen von Objekten definiert, die die Produkt-Interfaces implementieren. Im Diagramm sind dies die Produkt-Interfaces A und B, wobei ersteres von konkreten Klassen A1 und A2 und letzteres von B1 und B2 implementiert werden.

Das Erzeugen der konkreten Objekte wird dabei vor der Anwendungslogik (weitestgehend) verborgen, indem der Konstruktoraufruf der konkreten Klassen von den in Fabrik1 bzw. Fabrik2 implementierten Fabrikmethoden erledigt wird.

Da die Anwendungslogik nur die Interfaces verwendet, kann eine konkrete Fabrik einfach gegen eine andere ausgetauscht werden.

Diagramm: Erbauer



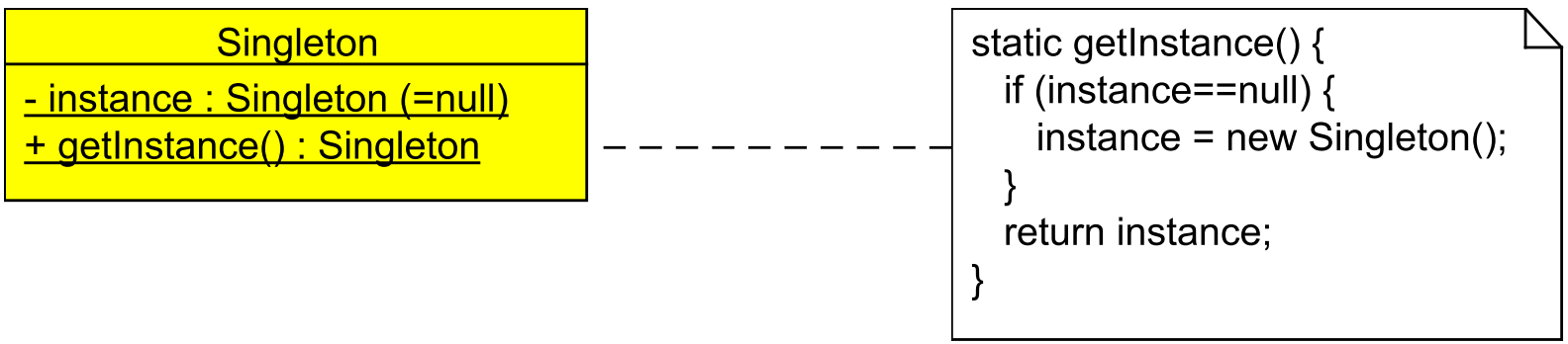
Wesentliche Eigenschaften

Der Erbauer (*Builder*) ist ein Entwurfmuster, in dem ein Direktor die Aufgabe übernimmt, über einen Erbauer Schritt für Schritt ein Endprodukt aufzubauen.

Hierbei verwendet der Direktor ein Erbauer-Interface, das konkrete Produkt wird über den konkreten Erbauer zusammen gebaut.

Häufig werden innerhalb des Erbauers weitere Entwurfsmuster verwendet, z. B. ist der Erbauer oft ein Kompositum.

Diagramm: Singleton



Wesentliche Eigenschaften

Das *Singleton* soll sicher stellen, dass nur eine Instanz einer Klasse existiert, wobei besondere Vorsichtsmaßnahmen bei Nebenläufigkeit erforderlich sind (s. 3. Vorlesung).

Das Singleton ist ein umstrittenes Entwurfsmuster und wird häufig sogar als *anti pattern* bezeichnet, dessen Verwendung vermieden werden sollte.

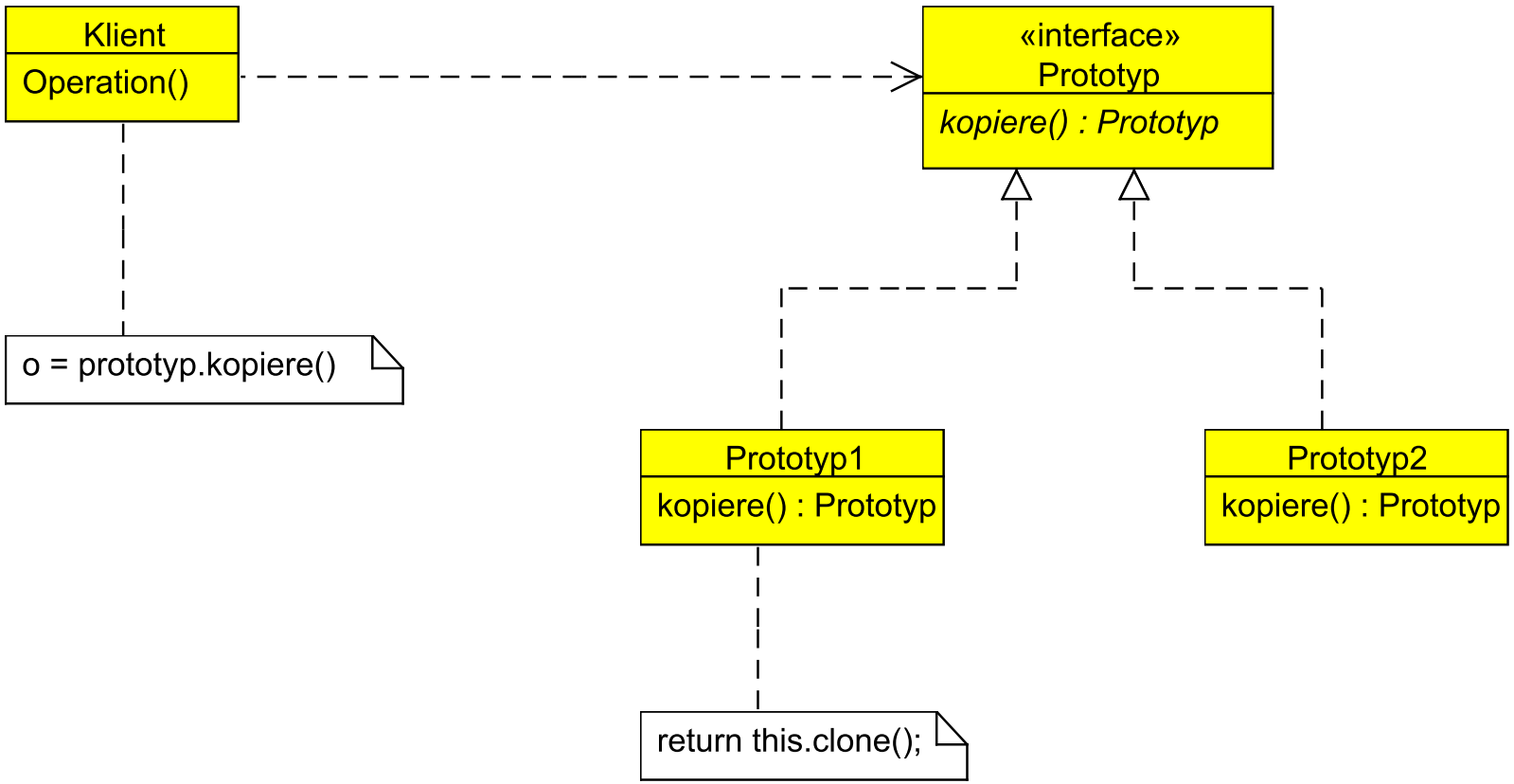
Gründe für Anti-Pattern

- Das Singleton verstößt gegen das *Single Responsibility Principle*: Ein Singleton erledigt neben der eigentlichen Funktionalität noch das Kümmern darum, dass es nur ein Exemplar gibt.
- Ein Singleton ist ein potenzielles Speicherloch: Wer kümmert sich darum, dass die bereitgestellte Instanz wieder zerstört und belegte Ressourcen freigegeben werden?
- Ein Singleton verführt zu prozeduralem Programmierstil, d. h. das Singleton stellt letztendlich das Gegenstück zu globalen Variablen zur Verfügung.

Gründe für Anti-Pattern (Forts.)

- Vererbung ist bei Singletons schwierig bis unmöglich.
- Ein Singleton verschleiert Abhängigkeiten zu anderen Objekten. Anstatt bei Operationen Referenzen auf dafür notwendige Objekte zu übergeben (damit expliziter Ausdruck der Abhängigkeit in der API), werden die (unsichtbar) innerhalb der Implementation über das Singleton geholt.

Diagramm: Prototyp



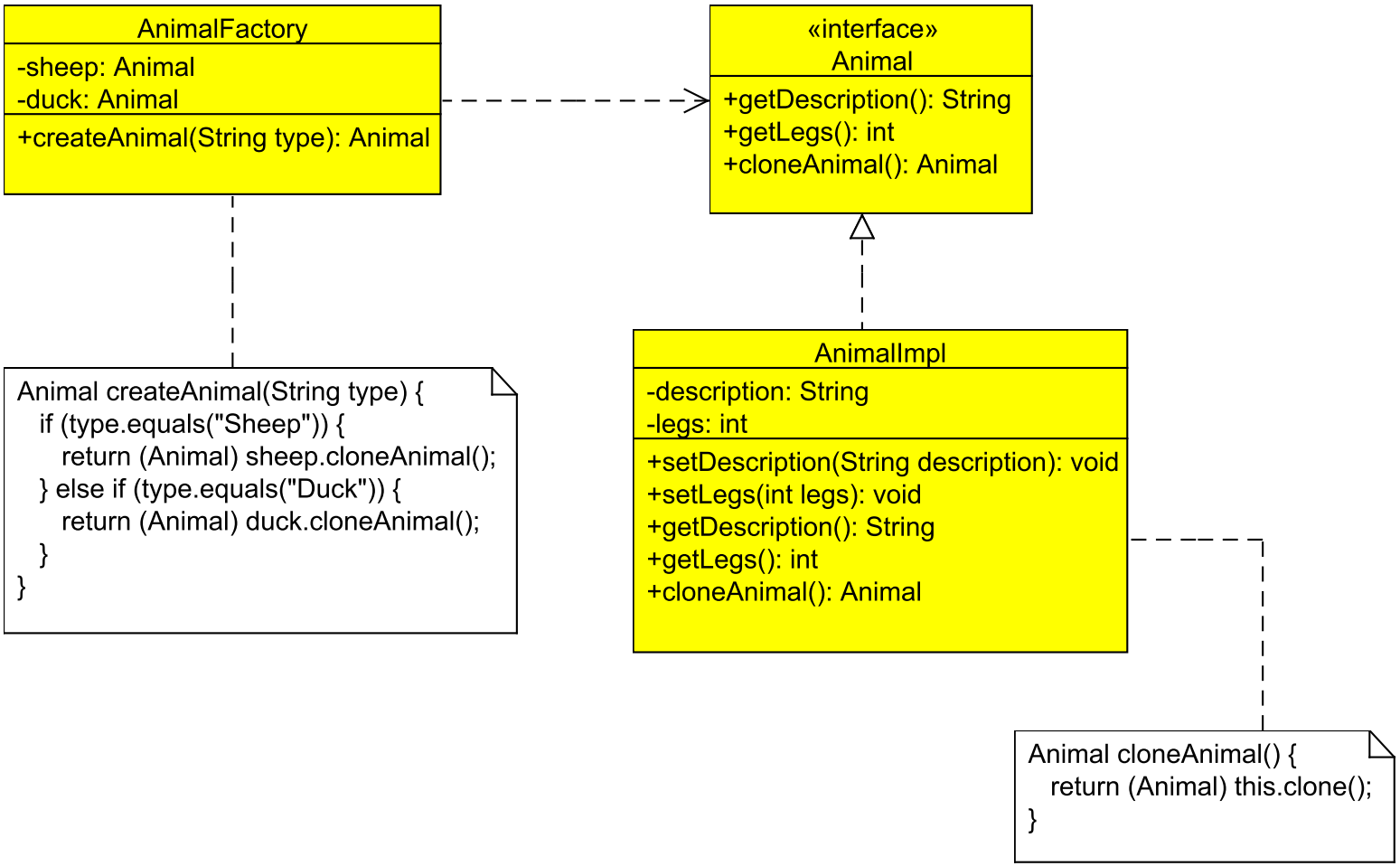
Wesentliche Eigenschaften

Das Prototype-Muster (*prototype*) dient wie die Abstrakte Fabrik dem Erzeugen von Instanzen von verschiedenen konkreten Produkten, die ein gemeinsames Produkt-Interface implementieren.

Im Gegensatz zur Abstrakten Fabrik wird jedoch nicht für jede konkrete Produktklasse eine Fabrikklasse implementiert, die die entsprechende Erzeugungsmethode bereitstellt, sondern hier unterstützt jedes Produkt die Operation, von sich selbst eine Kopie zu erstellen.

Typischerweise benötigt man eine Vorlage für das Produkt (den Prototypen), die zum Erstellen von Produkten als Kopie dient.
Analogie: Erzeugen eines Dokumentes in einem Office-Programm von einer Vorlage.

Beispiel


AnimalFactory
-sheep: Animal
-duck: Animal
+createAnimal(String type): Animal
«interface»
Animal
+getDescription(): String
+getLegs(): int
+cloneAnimal(): Animal
Animal createAnimal(String type) {
if (type.equals("Sheep")) {
return (Animal) sheep.cloneAnimal();
} else if (type.equals("Duck")) {
return (Animal) duck.cloneAnimal();
}
}
AnimalImpl
-description: String
-legs: int
+setDescription(String description): void
+setLegs(int legs): void
+getDescription(): String
+getLegs(): int
+cloneAnimal(): Animal
Animal cloneAnimal() {
return (Animal) this.clone();
}

Code: Interface Animal und dessen Implementation

Animal.java:

```
public interface Animal {
   public String getDescription();
   public int getLegs();
   public Animal cloneAnimal();
}
```

AnimalImpl.java:

```
class AnimalImpl implements Animal, Cloneable {
   private String description;
   private int legs;

   public void setDescription(String description) {
      this.description = description;
   }

   public void setLegs(int legs) {
      this.legs = legs;
   }
```

Code (Forts.)

AnimalImpl.java (Forts.):

```
    @Override
    public String getDescription() {
        return description;
    }

    @Override
    public int getLegs() {
        return legs;
    }

    @Override
    public Object cloneAnimal() {
        try {
            return (Animal) super.clone();
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }
}
```