Programmieren in Java

Vorlesung 04: Rekursive Klassen

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2015

Inhalt

Verkettete Listen

- Unveränderliche Daten
- Entwurf von Methoden auf Listen

Arithmetische Ausdrücke

- Entwurf von Methoden auf Ausdrücken
- Erweiterung I: Neue Art von Ausdruck hinzufügen
- Erweiterung II: Neue Operation hinzufügen

Adventure

- Decorator
- Template-Methode
- Beispiele für Anwendung des Decorator

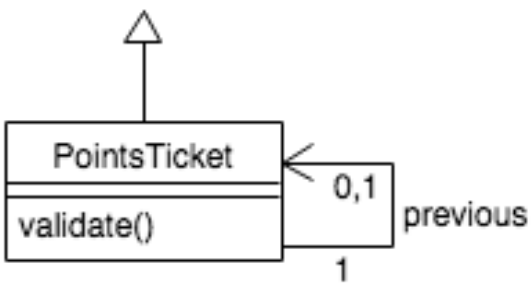
Executive Summary — Rekursive Klassen

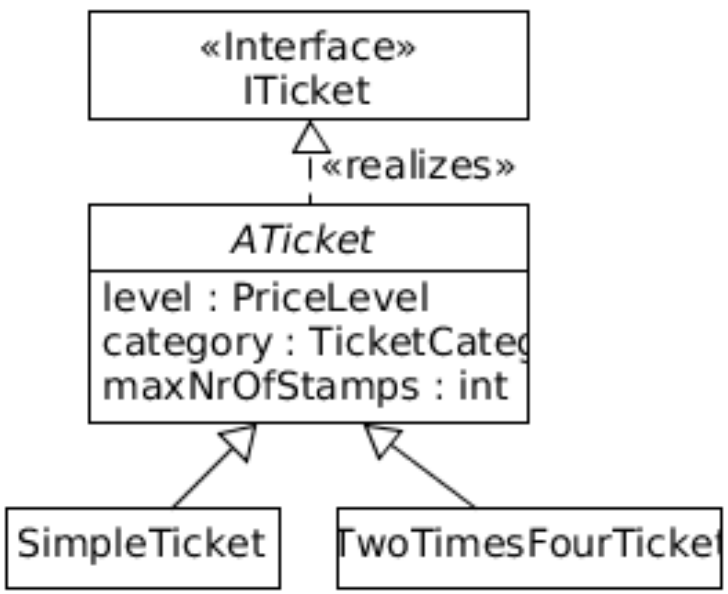
1. Datenmodellierung
   - Unveränderliche Daten (Immutable Data)
   - Verkettete Listen
   - Arithmetische Ausdrücke
2. Methodenentwurf auf rekursiven Klassen
   - Rekursive Methoden
   - Composite Pattern
   - Decorator Pattern
   - Template Method

Rekursive Klassen

- Eine Klasse ist **rekursiv**, falls sie eine rekursive Assoziation besitzt.
- Eine Assoziation ist **rekursiv**, falls sie von einer Klasse zur gleichen Klasse, einer Superklasse oder einem implementierten Interface führt.

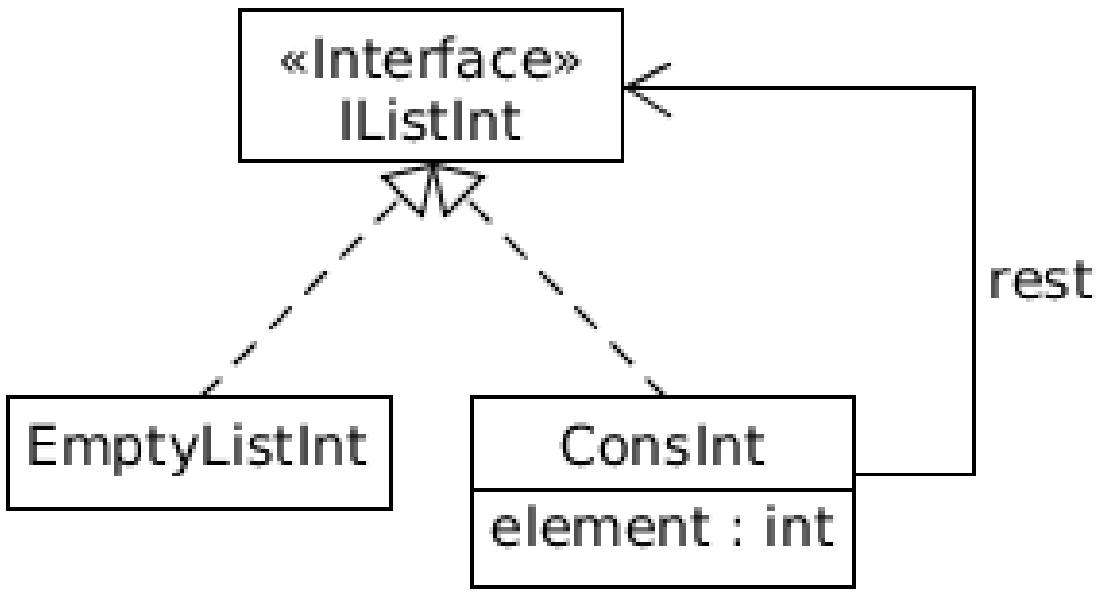
Beispiele





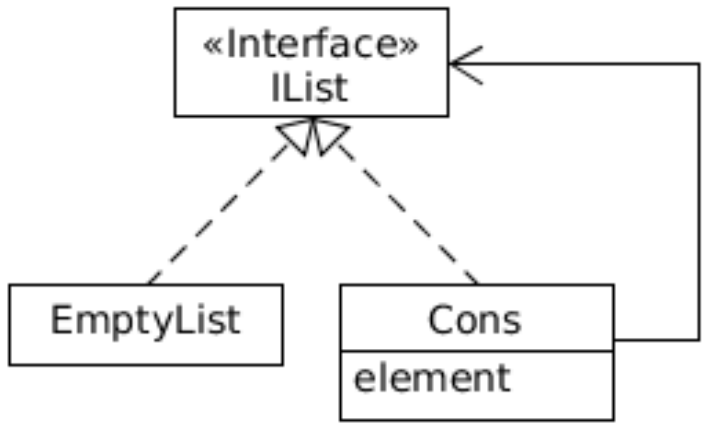
Verkettete Listen

Verkettete Listen von Zahlen



- Typ der Elemente instanziert zu **int**

Verkettung der Objekte




```
1   new ConsInt (1, new ConsInt (4, new ConsInt (9, new NilInt())))
```

Verkettete Liste in Java

```
1 public interface IListInt { }
```

```
1 public class NilInt implements IListInt { }
```

```
1 public class ConsInt implements ConsInt {
2   private final int element;
3   private final IList rest;
4   public ConsInt (int element, IList rest) {
5     if (rest == null)
6       throw new IllegalArgumentException("rest of list must not be null");
7     this.element = element;
8     this.rest = rest;
9   }
```

Einschub: Unveränderliche Daten

- Es ist oft vorteilhaft, wenn die Felder von Objekten **unveränderlich** sind ("immutable").
- Erreicht durch Deklaration aller Felder als **final** (vgl. ConsInt).
- Vorteile
  - Ein immutable Objekt verhält sich immer gleich bezüglich equals() und hashCode().
  - Invarianten des Objekts müssen nur vom Konstruktor kontrolliert werden.
  - Ein immutable Objekt muss nicht kopiert werden.
  - Nebenläufiges Programmieren wird einfacher, wenn viele Objekte immutable sind.
- Nachteile
  - Ggf. geringe Einbußen der Performance

Entwurf von Methoden auf Listen

- **int** length();
  Länge einer Liste
- **int** sum();
  Summe der Elemente einer Liste
- IListInt append (IListInt other);
  Verketten der Liste mit other

Erweitern des Interface (Composite Pattern)

```
1 public interface IListInt {
2   int length();
3   int sum();
4   IListInt append();
5 }
```

Entwurf von Methoden auf Listen

Länge einer Liste: **int** length();

Leere Liste

Die leere Liste (NilInt) hat Länge 0.

```
1   new NilInt ().length() == 0
```

Implementierung

```
1  public class NilInt implements IListInt {
2    int length() {
3      return 0;
4    }
5    //...
6  }
```

Entwurf von Methoden auf Listen

Länge einer Liste: **int** length();

Nichtleere Liste

Eine nicht-leere Liste (ConsInt) hat Länge 1 plus Länge der Restliste.

```
1  new ConsInt(51, new NilInt ()).length() == 1
2  new ConsInt(42, new ConsInt(51, new NilInt ())).length() == 2
```

Implementierung

```
1  public class ConsInt implements IListInt {
2    int length() {
3      return 1 + rest.length();
4    }
5    //...
6  }
```

Entwurf von Methoden auf Listen

Summe einer Liste: **int** sum();

Leere Liste

Die leere Liste (NilInt) hat Summe 0.

```
1   new NilInt ().sum() == 0
```

Implementierung

```
1  public class NilInt implements IListInt {
2     int sum() {
3        return 0;
4     //...
5     }
6  }
```

Entwurf von Methoden auf Listen

Summe einer Liste: **int** sum();

Nichtleere Liste

Eine nicht-leere Liste (ConsInt) hat als Summe ihr Element plus Summe der Restliste.

```
1        new ConsInt(51, new NilInt ()).sum() == 51;
2        new ConsInt(42, new ConsInt(51, new NilInt ())).sum() == 93;
```

Implementierung

```
1 public class ConsInt implements IListInt {
2   int sum() {
3     return this.element + rest.sum();
4   //...
5   }
6 }
```

Entwurf von Methoden auf Listen

Listenverkettung: IListInt append(IListInt other);

Leere Liste

Verketten der leeren Liste (NilInt) mit other ist other.

```
1   new NilInt ().append(other) == other
```

Implementierung

```
1 public class NilInt implements IListInt {
2   IListInt append(IListInt other) {
3     return other;
4   }
5 }
```

Entwurf von Methoden auf Listen

Listenverkettung: IListInt append(IListInt other);

Nichtleere Liste

Verketten einer nicht-leeren Liste (ConsInt) mit other liefert eine nicht-leere Liste mit gleichem ersten Element und als Restliste: die Verkettung der ursprünglichen Restliste mit other.

```
1  new ConsInt(51, new NilInt ()).append(other)
2  .equals (new ConsInt(51, ( new NilInt ().append(other) ) ));
3
4  new ConsInt(42, new ConsInt(51, new NilInt ())).append(other)
5  .equals (new ConsInt(42, ( new ConsInt(51, new NilInt ()).append(other) ) ));
```

Implementierung

```
1 public class ConsInt implements IListInt {
2   IListInt append(IListInt other) {
3     return new ConsInt(this.element, rest.append(other));
4   }
5   // ...
6 }
```

Komplette Implementierung NilInt

```
1  public class NilInt implements IListInt {
2    // empty list has length 0
3    int length() {
4      return 0;
5    }
6    // empty list has sum 0
7    int sum() {
8      return 0;
9    }
10   // empty list appended to other is other
11   IListInt append(IListInt other) {
12     return other;
13   }
14 }
```

Komplette Implementierung ConsInt

```
1  public class ConsInt implements IListInt {
2    // constructor and fields omitted
3    // length is 1 plus length of rest
4    int length() {
5      return 1 + rest.length();
6    }
7    // sum is element plus sum of rest
8    int sum() {
9      return element + rest.sum();
10   }
11   // Cons(element, rest) appended to other reconstructs the element
12   // around rest appended to other
13   IListInt append(IListInt other) {
14     return new ConsInt(element, rest.append(other));
15   }
16 }
```

Arithmetische Ausdrücke

Verwendung von Bäumen

- Listen und Bäume werden oft verwendet um abstrakte Datentypen effizient zu implementieren
- In diesen Anwendungen ist die Struktur ein Hilfsmittel um Effizienz zu erreichen
- Es gibt auch Anwendungen, wo die Baumstruktur ein essentieller Teil des Datenmodells ist
- Paradebeispiel dafür sind **arithmetische Ausdrücke**

Arithmetische Ausdrücke

Datenmodell

Ein arithmetischer Ausdruck ist entweder

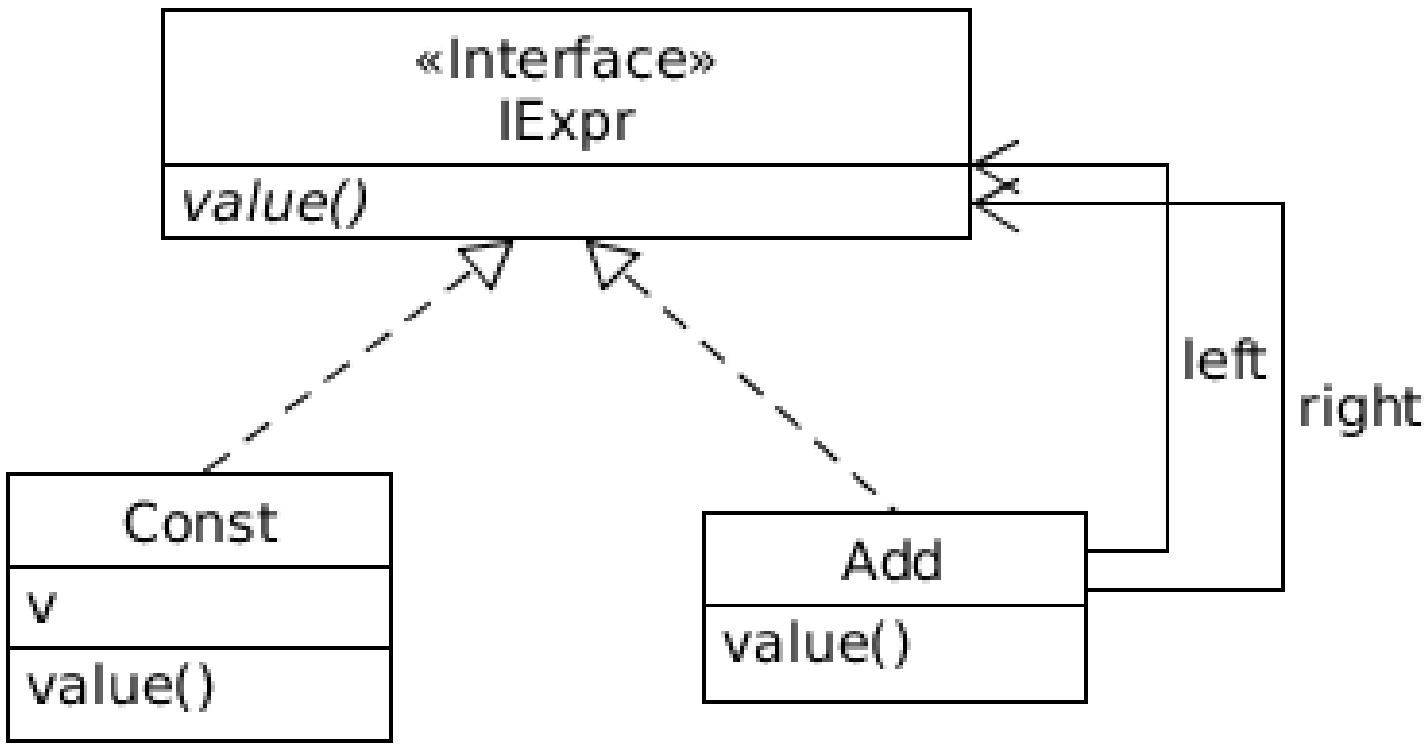
- eine Konstante (eine ganze Zahl)
  Beispiele: 0, 51, −42
- eine Summe von zwei arithmetischen Ausdrücken
  Beispiele: 3+4, 17+4, 17+ (2+2)

Operation

- Der Wert eines arithmetischen Ausdrucks (eine ganze Zahl)

Klassendiagramm: Arithmetische Ausdrücke

Modellierung mit Composite Pattern



Java: Gerüst für arithmetische Ausdrücke

```
1   public interface IExpr {
2     int value();
3   }
4   public class Const implements IExpr {
5     private final int v;
6     public Const (int v) {
7       this.v = v;
8     }
9     // ... method implementations
10  }
11  public class Add implements IExpr {
12    private finale IExpr left, right;
13    public Add(IExpr left, IExpr right) {
14      this.left = left;
15      this.right = right;
16    }
17    // ... method implementations
18  }
```

Methodenentwurf Const.value()

- Der Wert einer Konstanten ist die Konstante selbst.

```
1   new Const(42).value() == 42
```

- Implementierung

```
1   public class Const implements IExpr {
2     // fields and constructor ...
3     public int value() {
4       return this.v;
5     }
6   }
```

Methodenentwurf Add.value()

- Der Wert eines Add Ausdrucks ist die Summe der Werte seiner Teilausdrücke.

```
1  new Add(new Const (17), new Const (4)).value() == 21;
2  new Add(new Const (17), new Add (new Const (2), new Const (2))).value()
3    == 21;
```

- Implementierung

```
1      public class Add implements IExpr {
2        // fields and constructor ...
3      public int value() {
4        return this.left.add() + this.right.add();
5      }
6    }
```

Erweiterung I: Neue Art von Ausdruck hinzufügen

Erweitertes Datenmodell

Ein arithmetischer Ausdruck ist entweder

- eine Konstante (eine ganze Zahl)
  Beispiele: 0, 51, -42
- eine Summe von zwei arithmetischen Ausdrücken
  Beispiele: 3+4, 17+4, 17+ (2+2)
- ein Produkt von zwei arithmetischen Ausdrücken
  Beispiele: 3∗4, $2 * (17+4)$, $(2 * 3) * 4$

Erweiterung I: Neue Art von Ausdruck

Zum Hinzufügen einer neuen Art von Ausdruck muss nur eine neue Klasse definiert werden, die das Interface IExpr implementiert.

```
1   class Product implements IExpr {
2     private final IExpr left, right;
3     public Product (IExpr left, IExpr right) {
4       this.left = left; this.right = right;
5     }
6     // value is product of factors
7     public int value() {
8       return this.left.value() * this.right.value();
9     }
10  }
```

Beispiele mit Produkt

```
1 new Product (new Const (3), new Const (4)) . value() == 12;
2 new Product (new Const (2), new Add (new Const (17), new Const (4))) . value()
3       == 42;
```

Erweiterung II: Neue Operation hinzufügen

Erweitertes Aufgabenstellung

Ein arithmetischer Ausdruck kann

- mit value() seinen Wert berechnen;
- mit size() seine Größe berechnen.
  Die Größe eines arithmetischen Ausdrucks ist die Anzahl der Operatoren und Konstanten.

Erweiterung II: Neue Operation hinzufügen

Was ist zu tun?

- Interface anpassen

```
1  public interface IExpr {
2    int value();
3    int size();
4  }
```

- **In jeder Klasse, die das Interface implementiert, muss die neue Methode size() hinzugefügt werden**

Erweiterung II: Neue Operation hinzufügen

Was ist zu tun?

- Interface anpassen

```
1  public interface IExpr {
2    int value();
3    int size();
4  }
```

- **In jeder Klasse, die das Interface implementiert, muss die neue Methode size() hinzugefügt werden**

Nachteil

- Die Änderung ist nicht lokal.
- Es müssen **ggf. viele Klassen angepasst werden**.

Methodenentwurf Const.size()

- Die Größe einer Konstanten ist 1.

```
1   new Const(42).size() == 42
```

- Implementierung

```
1   public class Const implements IExpr {
2     // fields and constructor ...
3   public int size() {
4     return 1;
5   }
6 }
```

Methodenentwurf Add.size()

- Die Größe eines Add Ausdrucks ist die Summe der Größen seiner Teilausdrücke plus 1 für die Addition.

```
1  new Add(new Const (17), new Const (4)).size() == 3;
2  new Add(new Const (17), new Add (new Const (2), new Const (2))).size()
3    == 5;
```

- Implementierung

```
1      public class Add implements IExpr {
2        // fields and constructor ...
3      public int size() {
4        return 1 + this.left.size() + this.right.size();
5      }
6    }
```

Methodenentwurf Product.size()

- Die Größe eines Product Ausdrucks ist die Summe der Größen seiner Teilausdrücke plus 1 für die Multiplikation.

```
1  new Product(new Const (17), new Const (4)).size() == 3;
2  new Add(new Const (17), new Product (new Const (2), new Const (2))).size()
3    == 5;
```

- Implementierung

```
1      public class Product implements IExpr {
2        // fields and constructor ...
3      public int size() {
4        return 1 + this.left.size() + this.right.size();
5      }
6    }
```

Das Decorator Pattern

Adventure

Neue Anforderung

Ein Monster kann während des Spiels unterschiedliche Eigenschaften erhalten und wieder ablegen. Zum Beispiel kann es verflucht sein (dann schlägt es härter zu), es kann schläfrig sein (dann ist es langsamer), oder es kann unsichtbar sein.

Modellierung mit Decorator Pattern

Idee des Decorator Pattern

- Jede Eigenschaft/Modifikation wird durch eine eigene **Decorator-Klasse** repräsentiert.
- Jedes Decorator-Objekt verweist auf das Objekt vor der Modifikation.
- Die Operationen der Decorator-Klasse können ein verändertes Verhalten implementieren und/oder die entsprechende Operation auf dem Objekt vor der Modifikation aufrufen (delegieren).

Erweitertes Monster-Interface: Name

```
1 public interface IMonster {
2   // ...
3   // return the monster's name
4   String name ();
5 }
```

Beispielhafte Implementierung in Troll

```
1 public class Troll extends AMonster {
2   // ...
3   public String name() {
4     return "Troll";
5   }
6 }
```

Template-Methode

Die Methode name() wird als **Template-Methode** verwendet, d.h.

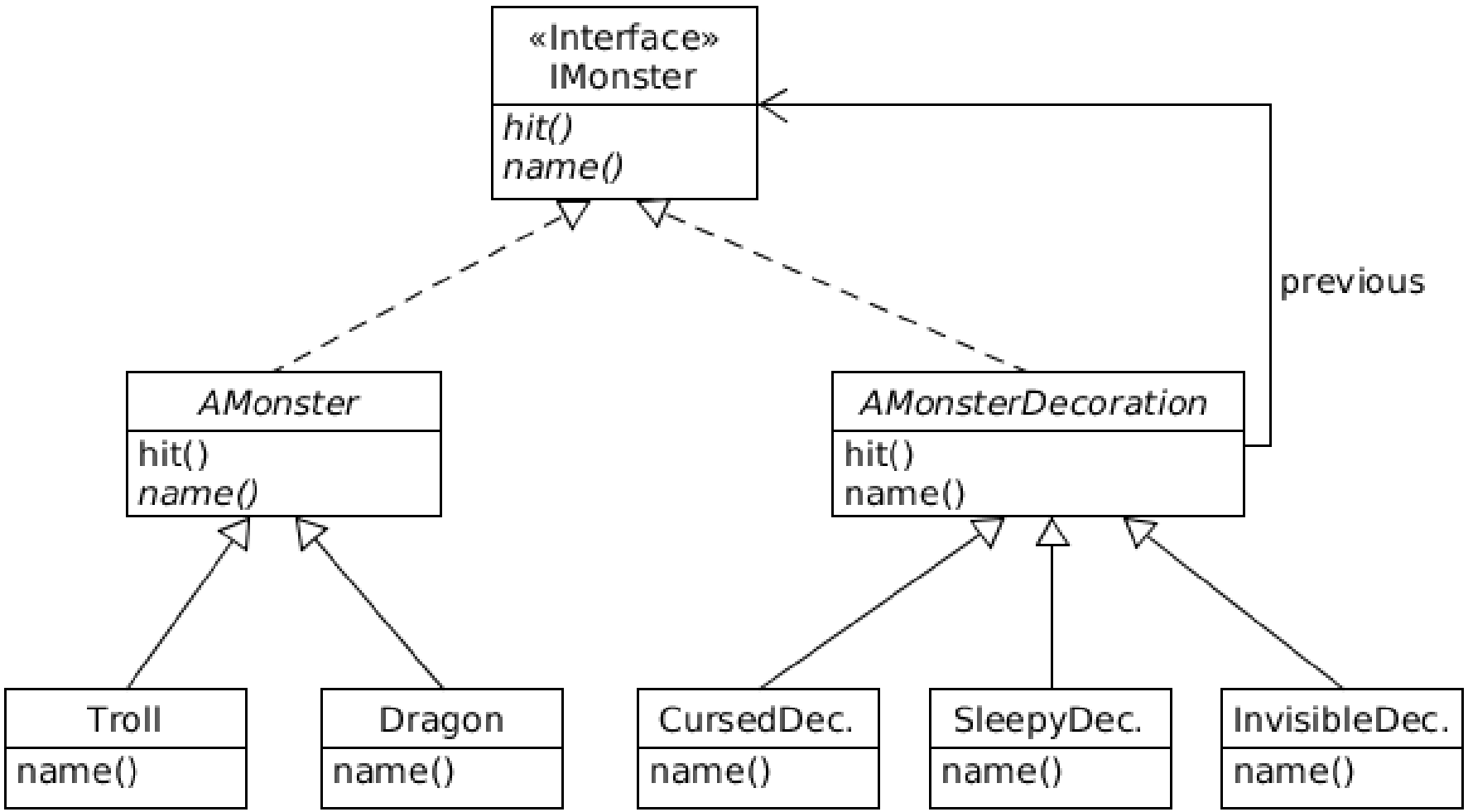
- Sie wird in AMonster verwendet.
- Sie bleibt abstrakt in AMonster (d.h. wird nicht implementiert).
- Sie wird nur in konkreten Subklassen (von AMonster) implementiert.

Effekt: Verwendung von name() in AMonster ist konfigurierbar durch die konkreten Subklassen.

```
1  public abstract class AMonster implements IMonster {
2    // ...
3    // only implemented in concrete subclasses
4    public abstract String name();
5    // ... but used right here:
6    public boolean hit(int force) {
7      System.out.println ("You hit the " + name() + "!");
8      return true;
9    }
10 }
```

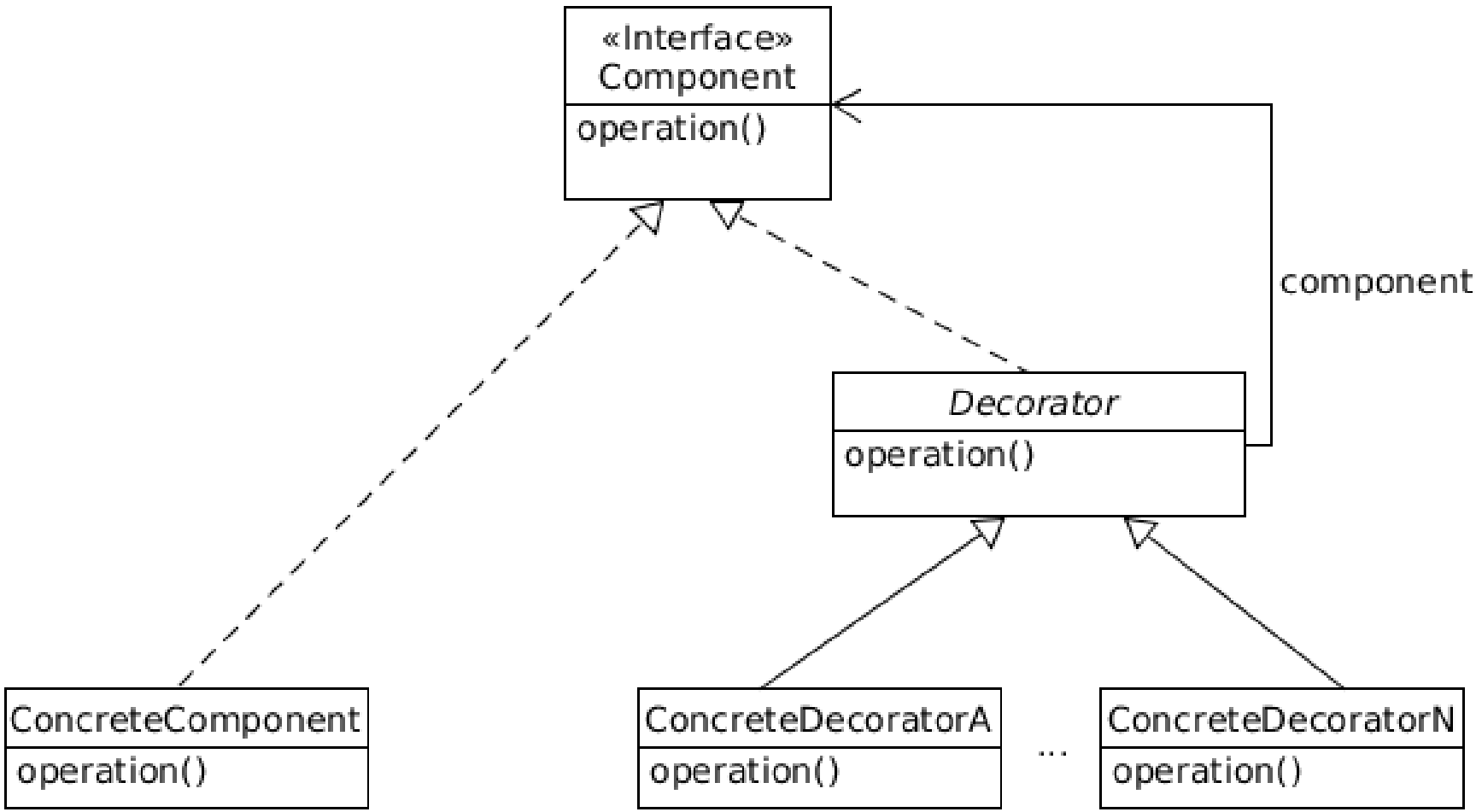
Klassendiagramm: Monster mit Eigenschaften

Modellierung mit Decorator Pattern



Klassendiagramm: Decorator Pattern

Allgemeine Struktur



Java: Monster mit Eigenschaften

```
1  public abstract class AMonsterDecoration implements IMonster {
2    // reference to original monster
3    private final IMonster monster;
4
5    protected AMonsterDecoration(IMonster monster) {
6      this.monster = monster;
7    }
8    // delegate method to original monster
9    public boolean hit(int force) {
10     return monster.hit(force);
11   }
12   // delegate method to origin
13   public String name() {
14     return monster.name();
15   }
16 }
```

Der verfluchte Decorator

```
1 public class CursedDecoration extends AMonsterDecoration {
2   protected CursedDecoration(IMonster monster) {
3     super(monster);
4   }
5
6   public String name() {
7     return "cursed " + super.name();
8   }
9 }
```

Der verfluchte Decorator — Beispiele

```
1  IMonster m = new Troll();
2  m.name(); // returns: "Troll"
3  m = new CursedDecoration (m);
4  m.name(); // returns: "cursed Troll"
5
6  IMonster d = new Dragon();
7  d.name(); // returns: "Dragon"
8  d = new CursedDecoration (d);
9  d.name(); // returns: "cursed Dragon''
```

Der unsichtbare Decorator

```
1 public class InvisibleDecoration extends AMonsterDecoration {
2   protected InvisibleDecoration(IMonster monster) {
3     super(monster);
4   }
5
6   public String name() {
7     return "invisible monster";
8   }
9 }
```

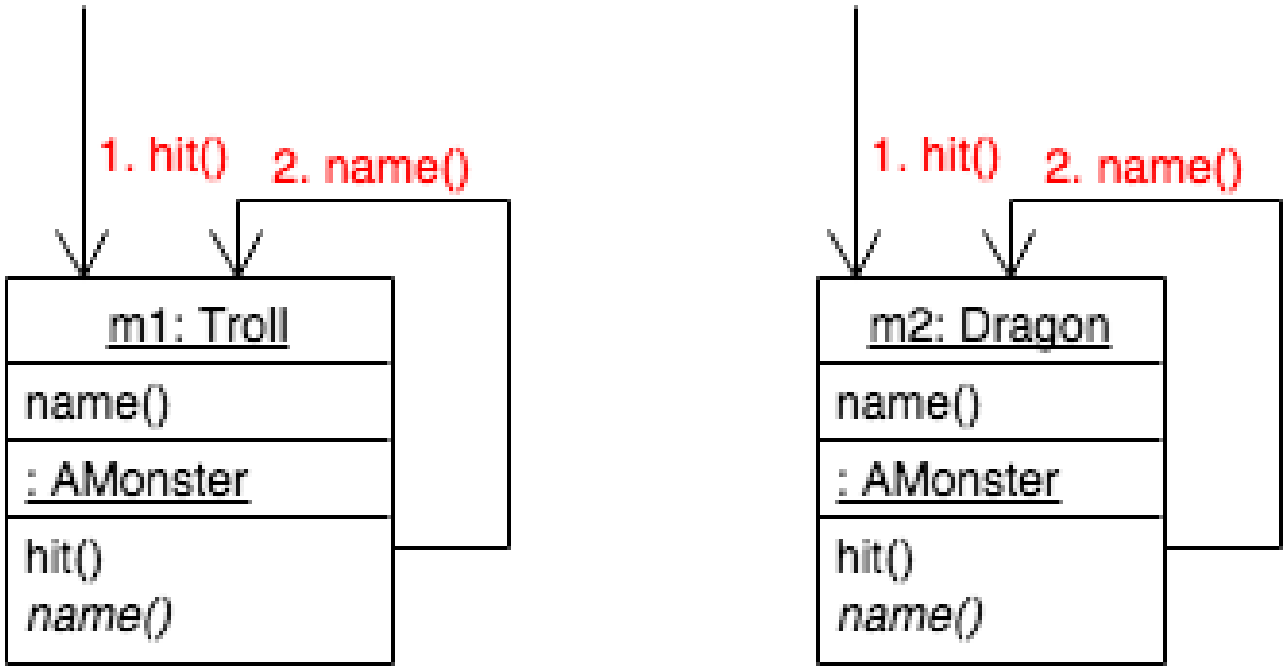
Der unsichtbare Decorator — Beispiele

```
1  IMonster m = new Troll();
2  m.name(); // returns: "Troll"
3  m = new InvisibleDecoration (m);
4  m.name(); // returns: ???
5  m = new CursedDecoration (m);
6  m.name(); // returns: ???
7
8  IMonster d = new Dragon();
9  d.name(); // returns: "Dragon"
10 d = new CursedDecoration (d);
11 d.name(); // returns: ???
12 d = new InvisibleDecoration (d);
13 d.name(10); // returns: ???
```

Decorator vs Template Method — Probleme

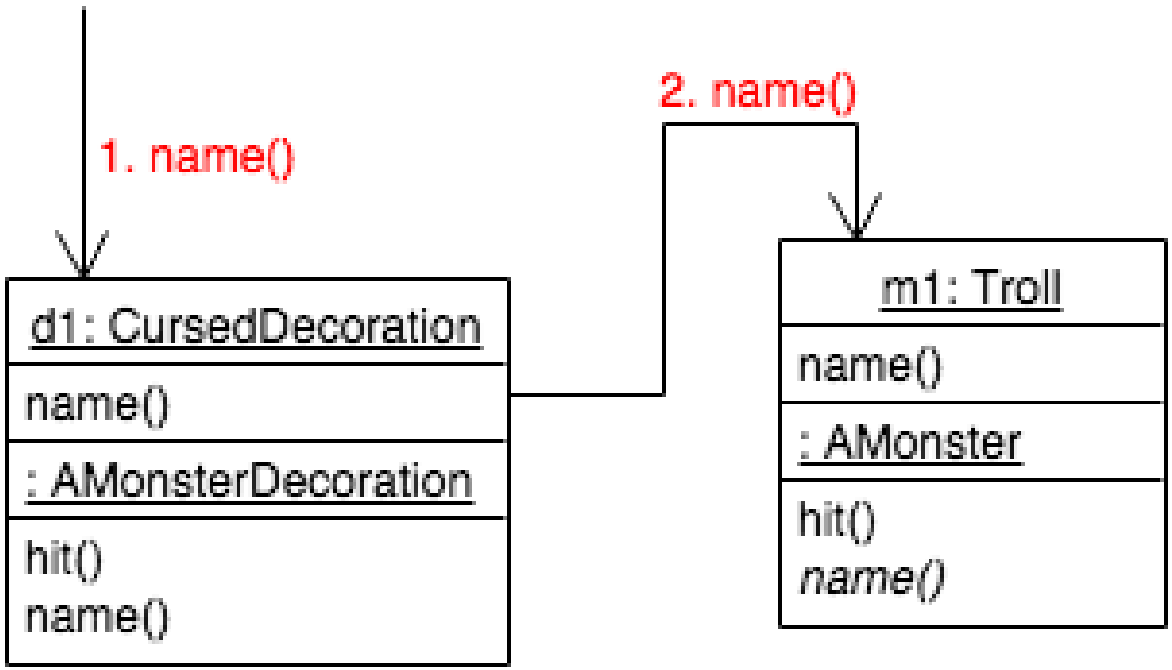
```
1  IMonster m = new Troll();
2  m.hit(10); // output: You hit the Troll!
3  m = new InvisibleDecoration (m);
4  m.hit(10); // output: ???
5  m = new CursedDecoration (m);
6  m.hit(10); // output: ???
7
8  IMonster d = new Dragon();
9  d.hit(10); // output: You hit the Dragon!
10 d = new CursedDecoration (d);
11 d.hit (10); // output: ???
12 d = new InvisibleDecoration (d);
13 d.hit(10); // output: ???
```

Warum Template Method funktioniert



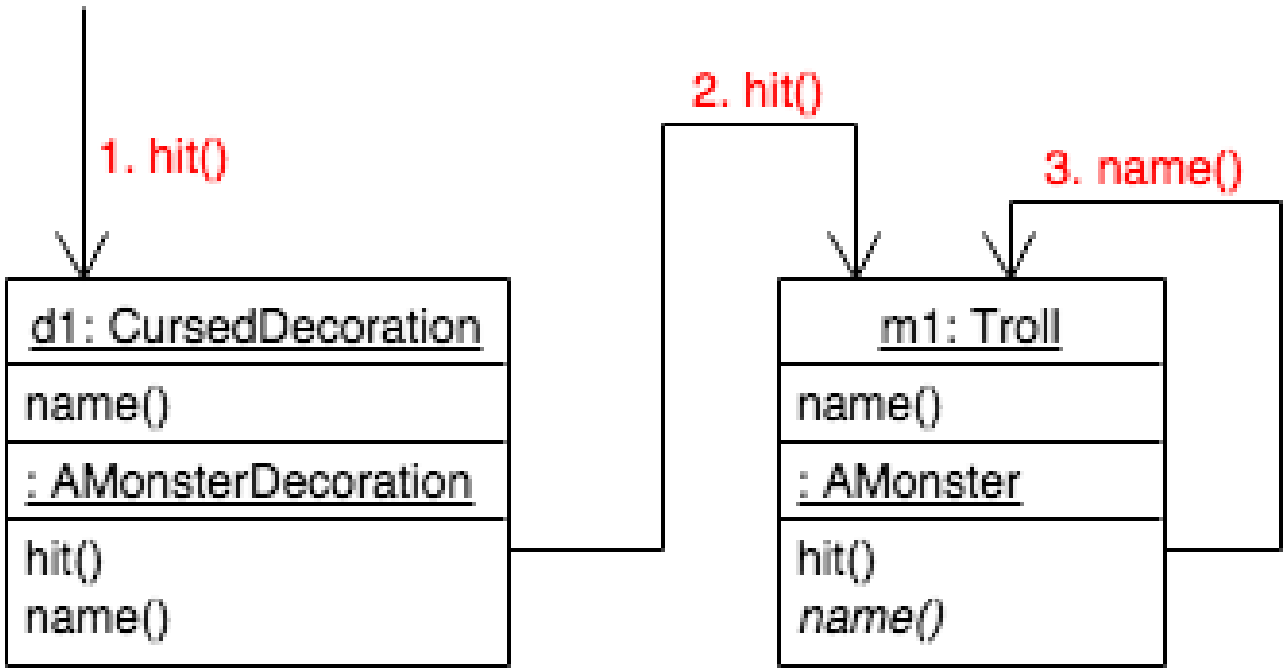
1. Aufruf von hit() auf m1 startet Code in AMonster
2. Führt zu Aufruf von name() auf **demselben Objekt** m1

Warum Decorator funktioniert



1. Aufruf von name() auf Decoration d1 started Code in CursedDecoration.
2. Dieser ruft name() auf dem Troll-Objekt m1 auf. (Wie gewollt)

Warum Decorator + Template Method nicht funktioniert



1. Aufruf von hit() auf d1 startet Code in AMonsterDecoration.
2. Dieser leitet weiter an hit() auf dem Troll-Objekt m1.
3. Dieser ruft name() auf **demselben Objekt** auf und **ignoriert somit die Decoration**!

Problem: „Object Schizophrenia“

- Das schlechte Zusammenwirken von Decorator und Template Method ist ein Fall von „Object Schizophrenia“.
- O.S. tritt auf, wenn die Funktionalität eines Objektes in der Implementierung auf mehrere Objekte verteilt ist
  (Hier: Decorations und eigentliches Objekt)
- In einem solchen Fall spricht man von **Objektkomposition**.
- Sie tritt meist zusammen mit **Delegation** auf.
- **Delegation** bedeutet, dass Methodenaufrufe unverändert von einem Objekt an ein anderes weitergereicht werden.
  Vgl. AMonsterDecoration.hit()

Lösungsansätze

1. Decorator und Template Method nicht zusammen benutzen.
2. Füge eine weitere Methode hinzu, die sich den ursprünglichen Empfänger der Methode hit() merkt (also den Eintrittspunkt in die Objektkomposition).

Zusammenfügen von Decorator und Template Method

- Das Empfängerobjekt des ersten Eintritts in das komponierte Objekt muss erhalten bleiben.
- Also wird dieses Objekt als weiterer Parameter mitgegeben!

```
1  public abstract class AMonster {
2    // external interface; delegate to internal method
3    // final = do not override
4    public final boolean hit(int force) {
5      return hit(force, this); // remember initial receiver
6    }
7    // internal interface; standard implementation for all monsters
8    protected boolean hit(int force, AMonster receiver) {
9      System.out.println ("You hit the " + receiver.name() + "!");
10     return true;
11   }
12
13   public abstract String name();
14 }
```

Der abstrakte Decorator

- Standardmäßig delegiert ein Decorator die Methoden hit() und name().
- Hier: Delegation bedeutet den Aufruf der gleichen Methode mit den gleichen Parametern auf dem previous Objekt.

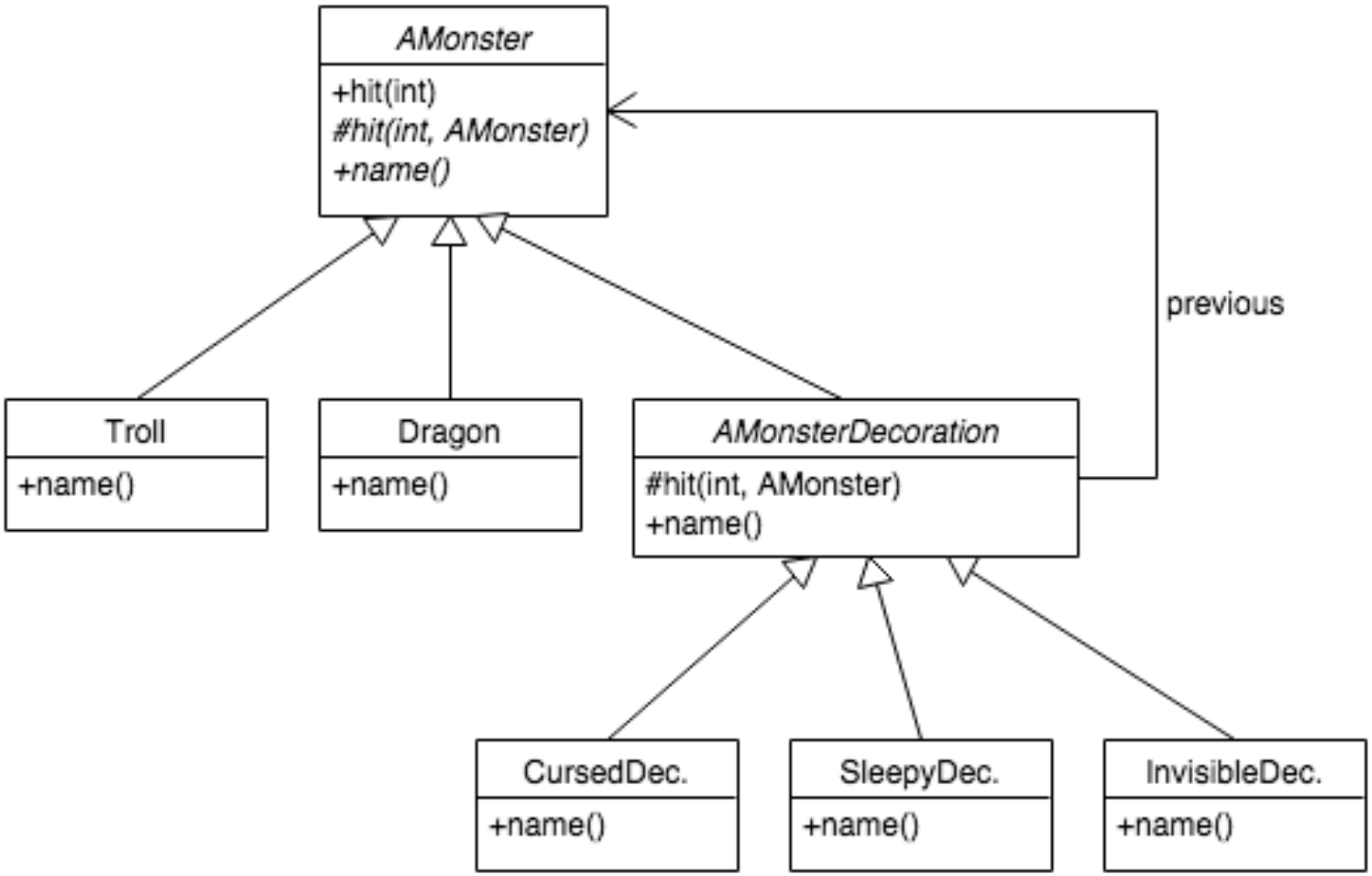
```
1   public abstract class AMonsterDecoration extends AMonster {
2     private AMonster previous;
3     // ...
4     protected boolean hit(int force, AMonster receiver) {
5       return previous.hit (force, receiver);
6     }
7     public String name() {
8       return previous.name();
9     }
10  }
```

Konkreter Decorator

- Modifiziert nur das Verhalten von name().

```
1 public class CursedDecoration extends AMonsterDecoration {
2   protected CursedDecoration(AMonster monster) {
3     super(monster);
4   }
5   @Override
6   public String name() {
7     return "cursed " + super.name();
8   }
9 }
```

Klassendiagramm Decorator + Template Method



- NB: + bedeutet **public**; ♯ bedeutet **protected**; − bedeutet **private**

Fragen?