Einführung in die Softwareentwicklung SS 13

Lernskript

Dozent:
Dr. Hildebrandt

LaTeX von:
Sven Bamberger

Zuletzt Aktualisiert:
30. Juli 2013

Inhaltsverzeichnis

1 Lernskript

1.1 Enum-Klasse

```
enum Color {Red, Green, Blue, Yellow}
```

in etwa äquivalent zu

```
class Color
{
    final static Color Red    = new Color();
    final static Color Green  = new Color();
    final static Color Blue   = new Color();
    final static Color Yellow = new Color();
}
```

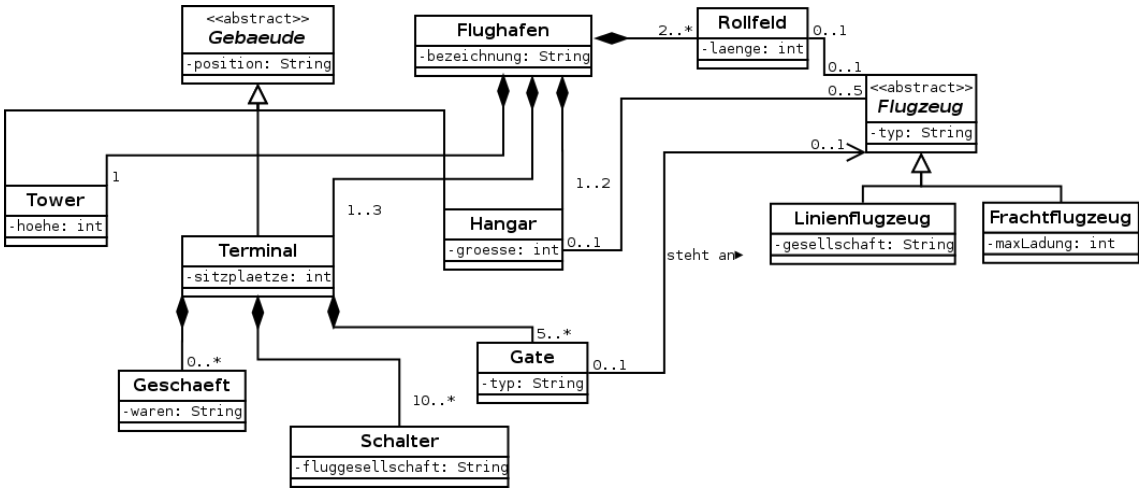
- Gleich benannte Enumwerte in unterschiedlichen Enumtypen sind inkompatibel
- Enum-Literal = enumtype.enumelement
- Definition einer Variablen

```
        Color c;
    c = Color.Red;
    if (c == Color.Yellow) ...
```

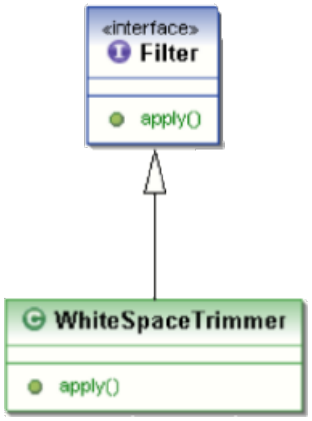
- Enumtypen = Referenztypen, Enumwerte = Klassenvariablen
- Vordefinierte Methoden in allen Enumtypen E:
  - static E valueOf(String s) = Enumwert mit dem Namen s
  - static E[] values () = Array mit allen Enumelementen
- Begriffe Enumelement = Enumwert = Enumobjekt
  - Enumobjekte sind eindeutig
  - zusätzliche Objekte können nicht erzeugt werden,
  - Vergleich mit == statt equals ausreichend

1.2 Unified Modelling Language (UML)

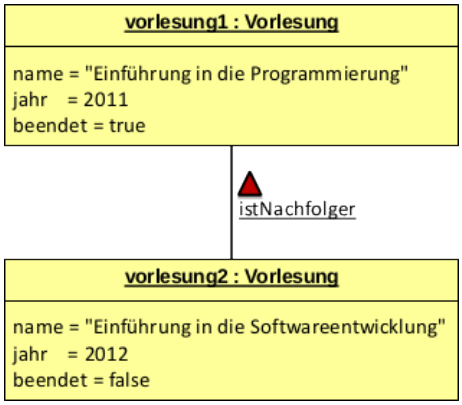
1.2.1 Basics

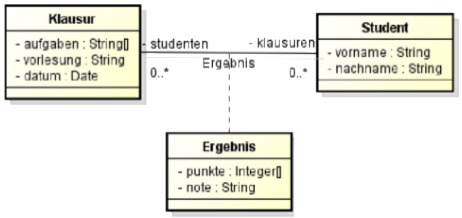


1.2.2 Interfaces



1.2.3 Objektdiagramme





1.3 Design Pattern

1.3.1 Decorator-Pattern

„Decorator fügt einem Objekt dynamisch zusätzliche Verantwortlichkeiten hinzu. Dekorierer bieten eine flexible Alternative zur Ableitung von Unterklassen zum Zweck der Erweiterung der Funktionalität“

- Vorteile:
  - Sehr flexibel und mächtig
  - Sehr dynamisch – Dekorationen können zur Laufzeit geändert werden
  - Basisklasse (Komponente, Eis, InputStream) muss zur Anwendung des Patterns nicht geändert werden!
  - Decorator können um bereits bestehende Klassen gelegt werden, ohne diese zu modifizieren!
  - Dekoratoren sind für Client-Code völlig transparent – sie können wie die ursprüngliche Klasse verwendet werden
  - Führt zu flachen Vererbungshierarchien
- Nachteile:
  - Fehler zu finden ist in langen Decorator-Ketten oft schwierig
  - API wird schnell durch viele kleine Objekte aufgebläht und schwer verständlich
  - Direkte Erstellung dekorierter Klassen ist umständlich (dafür gibt es aber auch wieder Pattern...)
  - Sinnlose oder gefährliche Kombinationen von Decorators können nur schwer verhindert werden
  - Decorator ist nicht angebracht, falls genaue Informationen über die Typen und Anzahlen von Dekorationen benötigt werden

1.4 Wrapperklassen

Primitiver Typ	Wrapper-KLasse
byte	Byte
short	Short
int	Integer
long	Long
double	Double
float	Float
char	Character
void	Void

Wrapper-Objekte können auf drei Arten entstehen:

- Durch Aufruf einer statischen valueOf()-Methode der Wrapper-Klasse, der ein primitiver Ausdruck oder ein String übergeben wird
- Durch Boxing: für primitive Werte erzeugt der Compiler automatisch valueOf()-Methodenaufrufe, die eine Instanz der Wrapper-Klasse zurückliefern
- Durch Aufruf von new und die Konstruktoren der Wrapper-Klassen

```
1  Integer i = Integer.valueOf("42"); // statische valueOf()-Methode
2  Long l = new Long(123);           // Konstruktoraufruf
3  Double d = 12.3;                  // Boxing
4  //
5  //
6  Integer int_wrap_array[] = {1, 2, 3}; // Ok ueber Boxing
7  int int_array[] = int_wrap_array; // Fehler!
8  int i = int_wrap_array[0]; // Ok ueber Unboxing
9  //
10 //
11 Integer i = 42;
12 i = i +1 // Integer hat kein + Unboxing
13
14 Integer j = 42;
15 if (i==j) {...} // == auf Integer moeglich: kein Unboxing !Vergleich auf Identitaet
16 //Daher Fehler
```

1.5 Generics

1.6 Collection