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();
}
```

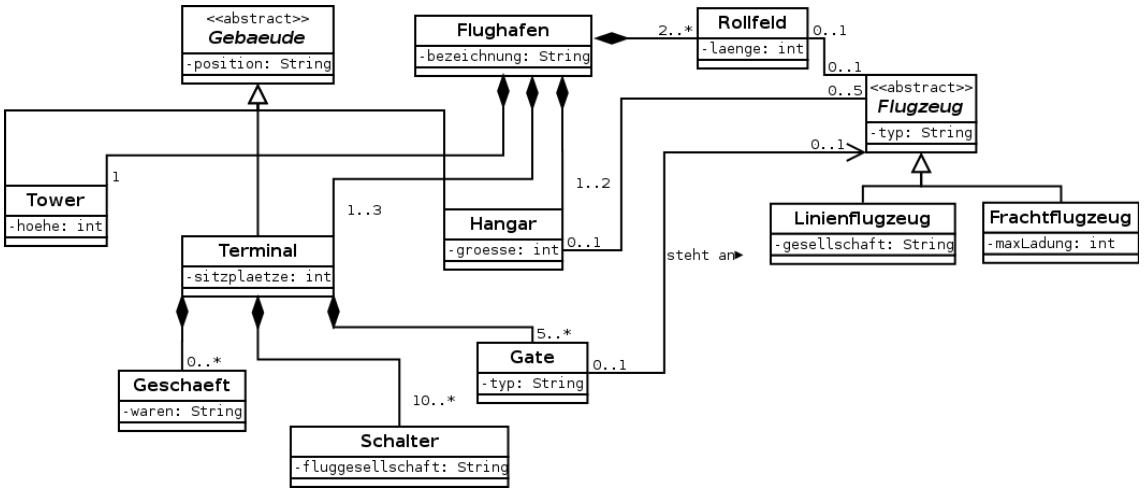
- Gleich benannte Enumwerte in unterschiedlichen Enumtypen sind inkompatibel
- Enum-Literal = enumtype.enumelement
- Definition einer Variablen

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

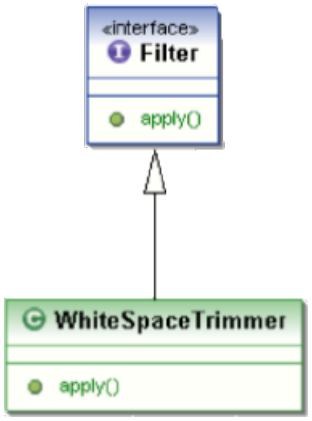
- Enumtypen = Referenztypen, Enumwerte = Klassenvariablen
- Vordefinierte Methoden in allen Enumtypen E:
  - static E valueOf(String s) = Enumwert mit dem Namen s
  - static E[] values () = Array mit allen Enumelementen
- Begriffe Enumelement = Enumwert = Enumobjekt
  - Enumobjekte sind eindeutig
  -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

1.2 Unified Modelling Language (UML)

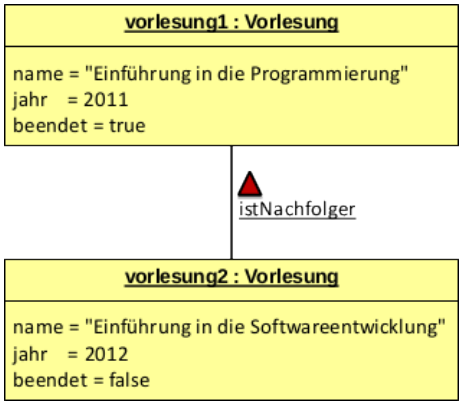
1.2.1 Basics

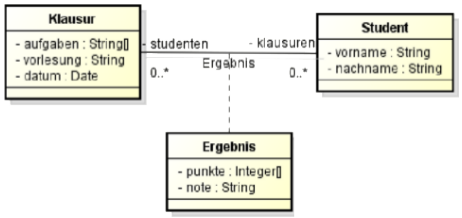


1.2.2 Interfaces



1.2.3 Objektdiagramme





1.3 Design Pattern

1.3.1 Observer-Pattern

1.3.2 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 Collections

- Java bietet eine Klassenhierachie für dynamsiche Container an, die sogenannten Collections
- Collections sollen zwar Objekte beliebiger Klassen speicher können, müssen dabei aber typsicher sein
- In jede Collection "passt"jeweils nur ein Typ von Objekten (bzw. natürlcih auch alle Instanzen von ihm abgeleiteter Typen)
- nur durch Generics möglich

1.6 Generics

Beobachtung: oft ist die Formulierung einer bestimmten Klasse oder die Arbeitsweise eines bestimmten Algorithmus unabhängig von den exakten beteiligten Typen

Beispiele:
- Container sind Klassen zum Speichern meherer Objekte als Elemente. Der Aufbau des Containers ist oft unabhängig vom gespeicherten Typ, dieser muss aber zur Implementierung angegeben werden
- Sortieralgorithmen werden üblicherweise unabhängig vom zu sortieren Typ entwickelt wobei dei Existenz einer Vergelcihsfunktion bekannten Namens für dne Typ vorausgesetzt wird. Zur Implementierung des Algorithmus muss aber ein konkreter Typ angegeben werden.

- Generics = Platzhalter

```
1 class Node<T>
2 {
3 private T info;
4 private Node<T> left;
5 private Node<T> right;
6 }
```

- Generics mit Restriktionen

```
1     class Node<T extends Number>
```

- Kompatibilität zwischen Typen
  - Implizite Typkonversion
    für bestimmte primitive Typen (nicht für Arrays)

```
1     int i = 1; double d  =i;
```

  - Implementierung
    Klasse => Interface (einschließlich arrays)

```
1     Cartesian ct = new Cartesian(1,2); Complex cx = ct;
```

  - Vererbung
    Abgeleitete KLasse => Basisklasse (einschließlich Arrays)

```
1     String s = "foo", Object o = s;
```

  - Autoboxing
    Primitiver Typ => Wrapperklasse (nicht für Arrays)

```
1     int i = 1; Integer it = i;
```

  - Auto-Unboxing
    Wrapperklasse => primitiver Typ (nicht für Arrays)

```
1     Integer it = new Integer(2); int i = it;
```

- Generic Invarianz

```
1 Node<Number> nn = new Node<Integer>(23); /7 Fehler
```

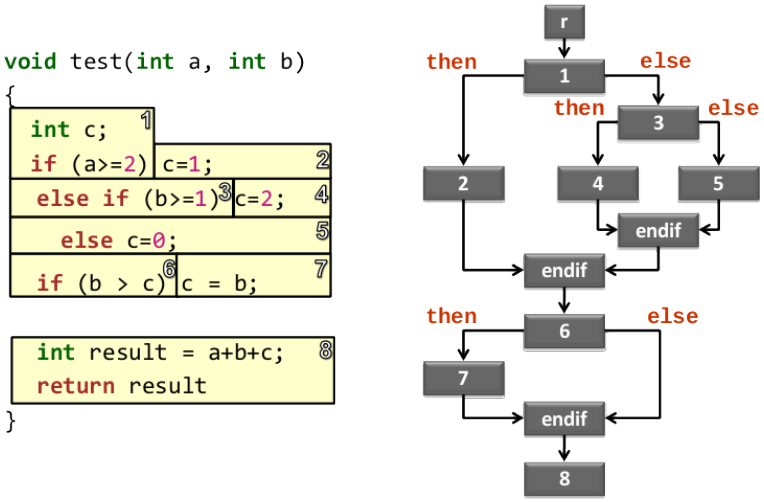
- Generic Wildcartypen

```
Node<?> nx
//============================
nx = new Node<?>(); // Fehler
//============================
Node<?> nx;
nx = new Node<String>("foo");// ok
nx = new Node<Integer>(1);// ok
nx = new Node<Double>(3.14);// ok
```

- Mögliche Varianzen

Varianz	Syntax	Lesen	Schreiben	Typargumente kompatibler Typen
Invarianz	C<T>	erlaubt	erlaubt	T
Bivarianz	C<?>	verboten	verboten	alle
Covarianz	C<? extend B>	erlaubt	verboten	B und abgeleitete Typen
Contravarianz	C<? super B>	verboten	erlaubt	B und Basistypen

1.7 Kontrollflussgraph



1.8 Testfälle

- Testfall C_0
  Finde Testfälle, durch die alle Knoten des Kontrollflussgraphen einmal besucht werden!
  Anweisungsüberdeckung

- Testfall C_1
  Finde Testfälle, durch die alle Kanten des Kontrollflussgraphen einmal verwendet werden!
  Zweigüberdeckung

- Testfall C_2
  Finde Testfälle, die zusammen alle Pfade durch den Kontrollflussgraphen durchlaufen
  Pfadüberdeckung

- Testfall C_2b
  teste alle Pfade, aber bei Schleifen nur "durchlaufenünd "übersprungen"

- Testfall C_2c
  teste alle PFade aber Schleifen nur bis n $\in \mathbb{N}$ Durchläufen

- Testfall C_3
  Finde Testfälle, die alle Bedingungen des Programms testen!
  Bedingungsüberdeckung

- Testfall C_3a
  Jede atomare Bedinung soll einmal true und einmal false annehmen

- Testfall C_3b
  jede Kombination atomarer Bedingungen soll getestet werden.

- Testfall C_3c
  jede atomare Bedingung und jede kombinierte Bedingung soll einaml true und einmal falls annehmen

1.9 Testen mit JUnit

Annotationen:

- @Test
  Testmethode (kann mehrfach vorkommen)
- @Before
  Wird vor jedem einzelnen Test aufgerufen
- @After
  Wird nach jedem einzelnen Test aufgerufen
- @BeforeClass
  Wird einmal vor allen Tests aufgerufen
- @AfterClass
  Wird einmal nach allen Tests aufgerufen
- Testfall für gcd(1,1) = :

```
1 @Test public void test11()
2 {
3     int want = 1
4     int have = new GCD().gcd(1,1);
5     assertEquals(want,have)
6 }
```

- Test mit Exception Erwartung

```
1 @Test(expected=ArithmeticException.class)
2 public void test10()
3 {...}
```

1.10 GUI

1.11 Threads