Softwaretechnik

Vorlesung 06: Design by Contract (Entwurf gemäß Vertrag)

Peter Thiemann

Universität Freiburg, Germany

SS 2008

Inhalt

Grundidee

Überführe das Konzept eines Vertrags zwischen Geschäftspartnern in die Softwaretechnik

Was ist ein Vertrag?

Ein bindendes Abkommen, das die *Verpflichtungen* und *Rechte* jedes Partners explizit aufführt.

Beispiel: Vertrag zwischen Bauherr und Bauunternehmer

	Verpflichtungen	**Rechte**
Bauherr	Stellt 5 ar Land; bezahlt für das Haus, falls dieses rechtzeitig fertig gestellt ist	Erhält das Haus in sechs Monaten
Bauunternehmer	Baut innerhalb von sechs Monaten das Haus auf dem bereitgestellten Land	Muss nicht arbeiten, falls das bereitgestellte Land kleiner als 5 ar ist; erhält Bezahlung falls das Haus rechtzeitig fertig gestellt ist

Wer sind die Vertragspartner in SE?

Partner können sein:

Module/Prozeduren, Objekte/Methoden, Komponenten/Operationen, ...

In einer Softwarearchitektur sind die Komponenten die Partner und jede Verbindung zwischen Komponenten kann einen Vertrag besitzen.

Verträge von prozeduralen Programmen

- Ziel: Spezifikation von Prozeduren (statischen Methoden)
- Ansatz: Mache *Zusicherungen* über die Prozeduren
  - Vorbedingung (Precondition)
    - Muss bei Eintritt in die Prozedur erfüllt sein (d.h. wahr sein)
    - Muss vom Aufrufer der Prozedur sichergestellt werden
  - Nachbedingung (Postcondition)
    - Muss beim Verlassen der Prozedur erfüllt sein
    - Muss von der Prozedur sichergestellt werden, *falls diese terminiert*
- **Vorbedingung**(*State*) $\Rightarrow$ **Nachbedingung**(**procedure**(*State*))
- Notation: {**Vorbedingung**} **procedure** {**Nachbedingung**}
- Zusicherungen in Prädikatenlogik der ersten Stufe oder OCL

Beispiel

```
/**
 * @param a an integer
 * @returns integer square root of a
 */
int root (int a) {
  int i = 0;
  int k = 1;
  int sum = 1;
  while (sum <= a) {
    k = k+2;
    i = i+1;
    sum = sum+k;
  }
  return i;
}
```

Spezifikation von `root`

- Typkorrektheit wird vom Compiler sichergestellt, $\mathtt{a} \in \mathtt{integer}$ und $\mathtt{root} \in \mathtt{integer}$ (das Ergebnis)

1. `root` als Funktion auf den natürlichen Zahlen

   Vorbedingung: $\mathtt{a} \geq 0$

   Nachbedingung: $\mathtt{root} * \mathtt{root} \leq \mathtt{a} < (\mathtt{root} + 1) * (\mathtt{root} + 1)$

2. `root` als Funktion auf den ganzen Zahlen

   Vorbedingung: **true**

   Nachbedingung:

$$(\mathtt{a} \geq 0 \quad \Rightarrow \quad \mathtt{root} * \mathtt{root} \leq \mathtt{a} < (\mathtt{root} + 1) * (\mathtt{root} + 1))$$
$$\wedge$$
$$(\mathtt{a} < 0 \quad \Rightarrow \quad \mathtt{root} = 0)$$

Abschwächen und Verstärken

Ziel: Bestmöglicher Vertrag

- Suche schwächste Vorbedingung
  - d.h., eine Vorbedingung, die durch alle anderen Vorbedingungen impliziert wird
  - größter Nutzen der Prozedur
  - größter Wertebereich der Prozedur
  - (Was bedeutet die Vorbedingung **false**?)
- Suche stärkste Nachbedingung
  - d.h.. suche eine Nachbedingung, die alle anderen Nachbedingungen impliziert
  - kleinstmögliche Wertemenge der Prozedur
  - (Was bedeutet die Nachbedingung **true**?)

Beispiel (schwächste Vorbedingung / stärkste Nachbedingung)

Betrachte `root` als Funktion auf den ganzen Zahlen

Vorbedingung: **true**

Nachbedingung:

$$
\begin{array}{l}
(\mathtt{a} \geq 0 \quad \Rightarrow \quad \mathtt{root} * \mathtt{root} \leq \mathtt{a} < (\mathtt{root} + 1) * (\mathtt{root} + 1)) \\
\wedge \\
(\mathtt{a} < 0 \quad \Rightarrow \quad \mathtt{root} = 0)
\end{array}
$$

- **true** ist die schwächste Vorbedingung
- Die Nachbedingung kann verstärkt werden zu

$$
\begin{array}{l}
(\mathtt{root} \geq 0) \ \wedge \\
(\mathtt{a} \geq 0 \quad \Rightarrow \quad \mathtt{root} * \mathtt{root} \leq \mathtt{a} < (\mathtt{root} + 1) * (\mathtt{root} + 1)) \ \wedge \\
(\mathtt{a} < 0 \quad \Rightarrow \quad \mathtt{root} = 0)
\end{array}
$$

Partielle Korrektheit vs Totale Korrektheit

. . . einer Prozedur f mit Vorbedingung P und Nachbedingung Q

- f ist *partiell korrekt*:
  für alle Zustände S:
  Falls Vorbedingung P für S gilt *und* f terminiert ausgehend vom Zustand S im Zustand S', *dann* gilt die Nachbedingung Q für S'.
- f ist *total korrekt*:
  für alle Zustände S:
  Falls Vorbedingung P für S gilt, *dann* terminiert f ausgehend vom Zustand S im Zustand S' *und* die Nachbedingung Q ist erfüllt für S'.

$\Rightarrow$ Totale Korrektheit verlangt Beweis der Terminierung

$\Rightarrow$ Totale Korrektheit impliziert partielle Korrektheit

Ein Beispiel

Füge ein Element in eine Tabelle fester Größe ein

```
int capacity; // size of table
int count; // number of elements in table
T get (String key) {...}
void put (String key, T element);
```

Vorbedingung: Tabelle ist nicht voll

$$\texttt{count} < \texttt{capacity}$$

Nachbedingung: neues Element in der Tabelle, `count` ist angepasst

$$\begin{aligned} & \texttt{count} \leq \texttt{capacity} \\ \wedge \quad & \texttt{get}(\texttt{key}) = \texttt{element} \\ \wedge \quad & \texttt{count} = \textbf{old}\ \texttt{count} + 1 \end{aligned}$$

Ein Beispiel

Vertrag

	Verpflichtungen	**Rechte**
Aufrufer	Aufruf von `put` nur auf nicht voller Tabelle	Erhält modifizierte Tabelle, in der das Element `element` dem Schlüssel `key` zugeordnet ist
Prozedur	Fügt `element` in Tabelle ein, so dass Zugriff mit Hilfe des Schlüssels `key` möglich ist	Der Fall "Tabelle ist voll" kann ignoriert werden

Weitere Element eines Vertrages

- Typsignatur (minimaler Vertrag)
- Fehler, die geworfen werden (Exceptions)
- Zeitliche Eigenschaften (Invarianten des Typs)
  - Die Kapazität der Tabelle verändert sich nicht über die Zeit
  - eine Menge, die nur anwachsen kann

Verträge für objekt-orientierte Programme

Verträge für Methoden haben zusätzliche Probleme zu behandeln

- lokaler Zustand
  Zustand des Empfängerobjekts muss spezifiziert werden
- Vererbung und dynamischer Methodenaufruf
  Empfängerobjekt hat zur Laufzeit einen Subtyp des statisch erwarten Typs; Methode kann überschrieben worden sein

Lokaler Zustand $\Rightarrow$ Klassen Invarianten

- Eine Klasseninvariante *INV* ist ein Prädikat, das für alle Objekte der Klasse gilt

$\Rightarrow$ Es muss durch alle Konstruktoren sichergestellt werden

$\Rightarrow$ Muss von allen sichtbaren Methoden erhalten werden

Vor- und Nachbedingungen für Methoden

- Konstruktoren c

$$\{\mathbf{Pre}_c\}\ c\ \{INV\}$$

- sichtbare Methoden m

$$\{\mathbf{Pre}_m \wedge INV\}\ m\ \{\mathbf{Post}_m \wedge INV\}$$

Tabellen Beispiel, die 2.

- `count` und `capacity` sind Instanzvariablen der Klasse `Table`
- Klasseninvariante $INV_{\texttt{Table}}$ ist $\texttt{count} \leq \texttt{capacity}$
- Spezifikation von `void put (String key, T element)`

  Vorbedingung:

  $$\texttt{count} < \texttt{capacity}$$

  Nachbedingung:

  $$\texttt{get(key)} = \texttt{element} \wedge \texttt{count} = \textbf{old}\ \texttt{count} + 1$$

Vererbung und dynamische Bindung

- Subklassen können Methoden überschreiben
- Auswirkung auf die Spezifikation:
  - Subklassen haben unterschiedliche Invarianten
  - Überschriebene Methoden können
    - unterschiedliche Vor- und Nachbedingungen haben
    - unterschiedliche Fehler werfen

    $\Rightarrow$ *Methodenspezialisierung*
- Relation zu den Vor- und Nachbedingungen der Basisklasse?
- Richtlinie: *No surprises requirement* (Wing, FMOODS 1997)
  (keine überraschenden Rechte/Auflagen/Forderungen)

  Eigenschaften, die von einem Objekt des Typs T erwartet werden, sollten auch für ein Objekt eines Subtypen S von T gelten.

Invarianten einer Subklasse

Angenommen

```
class Mytable extends Table ...
```

- jede Eigenschaft, die von `Table` erwartet wird, müssen Objekte von Type `Mytable` ebenfalls erfüllen.
- Falls o den Typ `Mytable` hat, dann muss $INV_{\texttt{Table}}$ für o gelten

$\Rightarrow$ $INV_{\texttt{Mytable}} \Rightarrow INV_{\texttt{Table}}$

- Beispiel: `Mytable` kann eine Hashtabelle sein, mit Invariante

$$INV_{\texttt{Mytable}} \equiv \texttt{count} \leq \texttt{capacity}/3$$

Methodenspezialisierung

Falls `Mytable` die Methode `put` neu definiert, dann muss ...

- die neue *Vorbedingung schwächer* sein und
- die neue *Nachbedingung stärker* sein

denn der Aufrufer

- garantiert nur $\mathbf{Pre}_{\texttt{put,Table}}$
- und erwartet $\mathbf{Post}_{\texttt{put,Table}}$

```
Table cast = new Mytable (150);
...
cast.put ("Arnie", new Terminator (3));
```

Anforderung der Methodenspezialisierung

Angenommen eine Klasse T definiert eine Methode m unter den Annahmen $\mathbf{Pre}_{T,m}$ und $\mathbf{Post}_{T,m}$, und wirft die Fehler $\mathbf{Exc}_{T,m}$. Falls die Klasse S die Klasse T erweitert und m überschreibt, so ist diese neue Definition eine *korrekte Methodenspezialisierung*, falls

- $\mathbf{Pre}_{T,m} \Rightarrow \mathbf{Pre}_{S,m}$ und (Index!)
- $\mathbf{Post}_{S,m} \Rightarrow \mathbf{Post}_{T,m}$ und (Index!)
- $\mathbf{Exc}_{S,m} \subseteq \mathbf{Exc}_{T,m}$
  jeder Fehler, der von $S.m$ geworfen wird, konnte auch von $T.m$ geworfen werden

Beispiel: `Mytable.put`

- $\textbf{Pre}_{\texttt{Mytable,put}} \equiv \texttt{count} < \texttt{capacity}/3$
  ist *keine* korrekte Methodenspezialisierung, da es nicht von $\texttt{count} < \texttt{capacity}$ impliziert wird.
- `Mytable` kann automatisch die Größe der Tabelle verändern, so dass $\textbf{Pre}_{\texttt{Mytable,put}} \equiv \textbf{true}$
  Dies wäre eine korrekte Methodenspezialisierung, da $\texttt{count} < \texttt{capacity} \Rightarrow \textbf{true}$!
- Angenommen, `Mytable` fügt eine neue Instanzvariable `T lastInserted` hinzu, die den Wert des letzten eingefügten Elements beinhaltet.

$$\begin{aligned} \textbf{Post}_{\texttt{Mytable,put}} \equiv \quad & \texttt{get(key)} = \texttt{element} \\ \wedge \quad & \texttt{count} = \textbf{old}\ \texttt{count} + 1 \\ \wedge \quad & \texttt{lastInserted} = \texttt{element} \end{aligned}$$

ist eine korrekte Methodenspezialisierung, da $\textbf{Post}_{\texttt{Mytable,put}} \Rightarrow \textbf{Post}_{\texttt{Table,put}}$

Methodenspezialisierung in Java 5

- In Java 5 darf beim Überschreiben einer Methode nur der Rückgabetyp spezialisiert werden (d.h. durch einen Subtyp ersetzt werden).
- Die Parametertypen müssen unverändert bleiben. (Warum?)

Beispiel: Angenommen A **extends** B

```
class C {
  A m () {
    return new A();
  }
}
class D extends C {
  B m () { // overrides A.m
    return new B();
  }
}
```

Contract Monitoring

- Was passiert, falls der Vertrag während der Ausführung verletzt wird?
- Eine solche Ausführung läuft außerhalb der Systemspezifikation.
- Das Verhalten des Systems kann *beliebig* sein.
  - Absturz
  - Weiterlaufen
  - *Contract Monitoring:* Wertet die Zusicherungen zur Laufzeit aus und wirft einen Fehler, falls eine Vertragsverletzung festgestellt wird.
- Wieso Beobachten?
  - Debugging (mit unterschiedlicher Genauigkeit des Monitorings)
  - Softwarefehlertoleranz (*e.g.*, α und β Releases)

Welche Fehler können auftreten?

Vorbedingung: Werte Bedingung bei Eintritt aus
Stelle Problem bei Aufrufer fest

Nachbedingung: Werte Bedingung bei Ende der Methode aus
Findet Fehler in der aufgerufenen Methode, d.h. in der Methode selbst

Invariante: Werte Bedingungen beim Eintritt und Austritt aus
Problem der aufgerufenden Klasse

Hierarchie: inkorrekte Methodenspezialisierung
Muss für alle Superklassen T von S geprüft werden

- $\mathbf{Pre}_{T,m} \Rightarrow \mathbf{Pre}_{S,m}$ bei Eintritt und
- $\mathbf{Post}_{S,m} \Rightarrow \mathbf{Post}_{T,m}$ bei Austritt

Wie?

Hierarchieprüfung

Angenommen `class` S `extends` T und überschreibt die Methode m.
Sei $T\ x = \texttt{new}\ S()$ und betrachte $x.m()$

- beim Eintritt
  - Falls $\mathbf{Pre}_{T,m}$ gilt, so muss $\mathbf{Pre}_{S,m}$ gelten
  - Es muss $\mathbf{Pre}_{S,m}$ gelten
- Beim Austritt
  - $\mathbf{Post}_{S,m}$ muss gelten
  - Falls $\mathbf{Post}_{S,m}$ gilt, so muss $\mathbf{Post}_{T,m}$ gelten
- Allgemein: Kaskade von Implikationen zwischen S und T
- Vor- und Nachbedingung werden nur für S geprüft!
- Wenn Vorbedingung von S nicht gilt, aber die Vorbedingung von T, dann liegt eine inkorrekte Methodenspezialisierung vor

Beispiel

```
interface IConsole {
  int getMaxSize();
    @post { getMaxSize > 0 }
  void display (String s);
    @pre { s.length () < this.getMaxSize() }
}

class Console implements IConsole {
  int getMaxSize () { ... }
    @post { getMaxSize > 0 }
  void display (String s) { ... }
    @pre { s.length () < this.getMaxSize() }
```

Eine korrekte Erweiterung

```
class RunningConsole extends Console {
  void display (String s) {
    ...
    super.display(s.substring ( n, n + getMaxSize() − 1));
    ...
  }
    @pre { true }
}
```

Eine fehlerhafte Erweiterung

```
class PrefixedConsole extends Console {
  String getPrefix() {
    return ">> ";
  }
  void display (String s) {
    super.display (this.getPrefix() + s);
  }
    @pre { s.length() < this.getMaxSize() − this.getPrefix().length() }
}
```

- Der Aufrufer muss nur die Vorbedingung von `IConsole`'s erfüllen
- Console.display kann mit zu langem Argument aufgerufen werden
- Der Programmierer von `PrefixedConsole` ist schuld!

Beispiel 2: Fehlerhafte Erweiterung des Interface

Programmierer Jim

```
interface I {
  void m (int a);
    @pre { a > 0 }
}

interface J extends I {
  void m (int a);
    @pre { a > 10 }
}
```

Programmierer Don

```
class C implements J {
  void m (int a) { ... };
    @pre { a > 10 }

  public static void
    main (String av[]) {
    I i = new C ();
    i.m (5);
  }
}
```

Eigenschaften des Monitoring

- Zusicherungen können beliebige seiteneffektfreie Ausdrücke sein
- Code zur automatischen Prüfung kann aus den Zusicherungen generiert werden
- Monitoring kann nur das Vorhandensein von Fehlern entdecken, nicht aber ihre prinzipielle Abwesenheit
- Fehlerfreiheit kann nur durch formale Verifikation sichergestellt werden

Verifikation von Verträgen

- Gegeben: Spezifikation von imperativen **Prozeduren** durch **Vorbedingung** und **Nachbedingung**
- Ziel: Formaler Beweis, dass **Vorbedingung**(*State*) $\Rightarrow$ **Nachbedingung**(**procedure**(*State*))
- Methode: *Hoare Logik*, *z.B.*, durch ein Deduktionssystem für *Hoare Tripel* der Form

$$\{\textbf{Vorbedingung}\}\ \textbf{procedure}\ \{\textbf{Nachbedingung}\}$$

- benannt nach C.A.R. Hoare, dem Erfinder von Quicksort, CSP, und vielen anderen