Softwaretechnik

Lecture 06: Design by Contract

Peter Thiemann

Universität Freiburg, Germany

SS 2009

Table of Contents

Basic Idea

Transfer the notion of contract between business partners to software engineering

What is a contract?

A binding agreement that explicitly states the *obligations* and the *benefits* of each partner

Example: Contract between Builder and Landowner

	Obligations	**Benefits**
Landowner	Provide 5 acres of land; pay for building if completed in time	Get building in less than six months
Builder	Build house on provided land in less than six month	No need to do anything if provided land is smaller than 5 acres; Receive payment if house finished in time

Who are the contract partners in SE?

Partners can be modules/procedures, objects/methods, components/operations, . . .

In terms of software architecture, the partners are the components and each connector may carry a contract.

Contracts for Procedural Programs

- Goal: Specification of imperative procedures
- Approach: give *assertions* about the procedure
  - Precondition
    - must be true on entry
    - ensured by caller of procedure
  - Postcondition
    - must be true on exit
    - ensured by procedure *if it terminates*
- $\mathbf{Precondition}(\mathit{State}) \Rightarrow \mathbf{Postcondition}(\mathbf{procedure}(\mathit{State}))$
- Notation: $\{\mathbf{Precondition}\}$ **procedure** $\{\mathbf{Postcondition}\}$
- Assertions stated in first-order predicate logic
- May also be used to specify the semantics of imperative programs

Example

Recall the following procedure:

```
/**
 * @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;
}
```

Specification of `root`

- types guaranteed by compiler: $\mathtt{a} \in \mathtt{integer}$ and $\mathtt{root} \in \mathtt{integer}$ (the result)

1. `root` as a partial function

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

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

2. `root` as a total function

   Precondition: **true**

   Postcondition:

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

Weakness and Strength

Goal:

- find weakest precondition
  i.e. a precondition that is implied by all other preconditions
  highest demand on procedure
  greatest domain of procedure
  (meaning of precondition **false**?)
- find strongest postcondition
  i.e. a postcondition that implies all other postconditions
  smallest range of procedure
  (meaning of postcondition **true**?)

Met by "root as a total function":

- **true** is weakest possible precondition
- "defensive programming"

Example (Weakness and Strength)

Look at `root` as a function over integers

Precondition: **true**

Postcondition:

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

- **true** is the weakest precondition
- The postcondition can be strengthened to

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

Partial Correctness vs Total Correctness

. . . of a procedure f with precondition P and postcondition Q

- f is *partially correct*:
  for all states S:
  if precondition P holds for S *and* f terminates from state S, *then* postcondition Q holds.
- f is *totally correct*:
  for all states S:
  if precondition P holds for S, *then* f terminates from state S, *and* postcondition Q holds.

$\Rightarrow$ Total correctness requires proof of termination

$\Rightarrow$ Total correctness implies partial correctness

An Example

Insert an element in a table of fixed size

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

Precondition: table is not full

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

Postcondition: new element in table, count updated

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

	Obligations	**Benefits**
Caller	Call `put` only on non-full table	Get modified table in which `element` is associated with `key`
Procedure	Insert `element` in table so that it may be retrieved through `key`	No need to deal with the case where table is full before insertion

Further elements of a contract

- type signature (minimal contract)
- exceptions raised
- temporal properties (type invariant)
  - the capacity of the table does not change over time
  - a set that is only supposed to grow

Contracts for Object-Oriented Programs

Contracts for methods have additional complications

- local state
  receiving object's state must be specified
- inheritance and dynamic method dispatch
  receiving object's type may be different than statically expected;
  method may be overridden

Local State $\Rightarrow$ Class Invariant

- class invariant *INV* is predicate that holds for all objects of the class

$\Rightarrow$ must be established by all constructors

$\Rightarrow$ must be maintained by all visible methods

Pre- and Postconditions for Methods

- constructor methods c

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

- visible methods m

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

Table example revisited

- `count` and `capacity` are instance variables of class `TABLE`
- $\mathit{INV}_{\texttt{TABLE}}$ is $\texttt{count} \le \texttt{capacity}$
- specification of `void put (T element, String key)`

  Precondition:

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

  Postcondition:

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

Inheritance and Dynamic Binding

- Subclass may override a method definition
- Effect on specification:
  - Subclass may have different invariant
  - Redefined methods may
    - have different pre- and postconditions
    - raise different exceptions

    $\Rightarrow$ *method specialization*
- Relation to invariant and pre-, postconditions in base class?
- Main guideline: *No surprises requirement* (Wing, FMOODS 1997)
  Properties that users rely on to hold of an object of type T should hold even if the object is actually a member of a subtype S of T.

Invariant of a Subclass

Suppose

class MYTABLE **extends** TABLE ...

- each property expected of a `TABLE` object should also be granted by a `MYTABLE` object
- if o has type `MYTABLE` then $INV_{\texttt{TABLE}}$ must hold for o

$\Rightarrow$ $INV_{\texttt{MYTABLE}} \Rightarrow INV_{\texttt{TABLE}}$

- Example: `MYTABLE` might be a hash table with invariant

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

Method Specialization

If `MYTABLE` redefines `put` then ...

- the new *precondition must be weaker* and
- the new *postcondition must be stronger*

because the caller

- guaranties only $\mathbf{Pre}_{\texttt{put},\texttt{Table}}$
- and expects $\mathbf{Post}_{\texttt{put},\texttt{Table}}$

```
TABLE cast = new MYTABLE (150);
...
cast.put (new Terminator (3), "Arnie");
cast.put (new Terminator (4), "Arnie-puppet");
```

Requirements for Method Specialization

Suppose class T defines method m with assertions $\mathbf{Pre}_{T,m}$ and $\mathbf{Post}_{T,m}$ throwing exceptions $\mathbf{Exc}_{T,m}$. If class S extends class T and redefines m then the redefinition is a sound method specialization if

- $\mathbf{Pre}_{T,m} \Rightarrow \mathbf{Pre}_{S,m}$ and
- $\mathbf{Post}_{S,m} \Rightarrow \mathbf{Post}_{T,m}$ and
- $\mathbf{Exc}_{S,m} \subseteq \mathbf{Exc}_{T,m}$
  each exception thrown by $S.m$ may also be thrown by $T.m$

Example: MYTABLE.put

- $\textbf{Pre}_{\texttt{MYTABLE,put}} \equiv \texttt{count} < \texttt{capacity}/3$
  not a sound method specialization because it is not implied by
  $\texttt{count} < \texttt{capacity}$.
- MYTABLE may automatically resize the table, so that $\textbf{Pre}_{\texttt{MYTABLE,put}} \equiv \textbf{true}$
  a sound method specialization because $\texttt{count} < \texttt{capacity} \Rightarrow \textbf{true}$!
- Suppose MYTABLE adds a new instance variable `T lastInserted` that holds the last value inserted into the table.

$$\begin{array}{rcl} \textbf{Post}_{\texttt{MYTABLE,put}} \equiv & & \texttt{item(key)} = \texttt{element} \\ & \wedge & \texttt{count} = \textbf{old}\ \texttt{count} + 1 \\ & \wedge & \texttt{lastInserted} = \texttt{element} \end{array}$$

is sound method specialization because $\textbf{Post}_{\texttt{MYTABLE,put}} \Rightarrow \textbf{Post}_{\texttt{TABLE,insert}}$

Method Specialization in Java 5

- Overriding methods in Java 5 only allows specialization of the result type. (It can be replaced by a subtype).
- The parameter types muss stay unchanged (why?)

Example : Assume A **extends** B

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

Contract Monitoring

- What happens if a system's execution violates an assertion at run time?
- A violating execution runs outside the system's specification.
- The system's reaction may be *arbitrary*
  - crash
  - continue
  - *contract monitoring:* evaluate assertions at run time and raise an exception indicating any violation
- Why monitor?
  - Debugging (with different levels of monitoring)
  - Software fault tolerance (*e.g.*, α and β releases)

What can go wrong

precondition: evaluate assertion on entry
identifies problem in the caller

postcondition: evaluate assertion on exit
identifies problem in the callee

invariant: evaluate assertion on entry and exit
problem in the callee's class

hierarchy: unsound method specialization
need to check (for all superclasses T of S)

- $\mathbf{Pre}_{T,m} \Rightarrow \mathbf{Pre}_{S,m}$ on entry and
- $\mathbf{Post}_{S,m} \Rightarrow \mathbf{Post}_{T,m}$ on exit

how?

Hierarchy Checking

Suppose `class` S `extends` T and overrides a method m.
Let $T\ x = \texttt{new}\ S()$ and consider $x.m()$

- on entry
  - if $\mathbf{Pre}_{T,m}$ holds, then $\mathbf{Pre}_{S,m}$ must hold, too
  - $\mathbf{Pre}_{S,m}$ must hold
- on exit
  - $\mathbf{Post}_{S,m}$ must hold
  - if $\mathbf{Post}_{S,m}$ holds, then $\mathbf{Post}_{T,m}$ must hold, too
- in general: cascade of implications between S and T
- pre- and postcondition only checked for S!
- If the precondition of S is not fulfilled, but the one of T is, then this is a wrong method specialization.

Examples

```
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() }
```

A Good Extension

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

A Bad Extension

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

- caller may only guarantee `IConsole`'s precondition
- Console.display can be called with to long argument
- blame the programmer of `PrefixedConsole`!

Example 2: Bad Interface Extension

Programmer Jim

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

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

Programmer 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);
  }
}
```

Properties of Monitoring

- Assertions can be arbitrary side effect-free boolean expressions
- Instrumentation for monitoring can be generated from the assertions
- Monitoring can only prove the presence of violations, not their absence
- Absence of violations can only be guaranteed by formal verification

Verification of Contracts

- Given: Specification of imperative **procedure** by **Precondition** and **Postcondition**
- Goal: Formal proof for **Precondition**(*State*) $\Rightarrow$ **Postcondition**(**procedure**(*State*))
- Method: *Hoare Logic*, *i.e.*, a proof system for *Hoare triples* of the form

$$\{\textbf{Precondition}\}\ \textbf{procedure}\ \{\textbf{Postcondition}\}$$

- named after C.A.R. Hoare, the inventor of Quicksort, CSP, and many other
- here: method bodies, no recursion, no pointers (extensions exist)

Syntax

$$
\begin{array}{llll}
E & ::= & c \mid x \mid E + E \mid \dots & \text{expressions} \\
B, P, Q & ::= & \neg B \mid P \wedge Q \mid P \vee Q & \text{boolean expressions} \\
 & \mid & E = E \mid E \leq E \mid \dots & \\
C, D & ::= & x{=}E & \text{assignment} \\
 & \mid & C;D & \text{sequence} \\
 & \mid & \texttt{if } B \texttt{ then } C \texttt{ else } D & \text{conditional} \\
 & \mid & \texttt{while } B \texttt{ do } C & \text{iteration} \\
 & & & \\
\mathcal{H} & ::= & \{P\}C\{Q\} & \text{Hoare triples}
\end{array}
$$

- (boolean) expressions are free of side effects