Software Engineering
Lecture 13: Design by Contract

Peter Thiemann

University of Freiburg, Germany

17.06.2013
Table of Contents

Design by Contract

- Contracts for Procedural Programs
- Contracts for Object-Oriented Programs
- Contract Monitoring
- Verification of Contracts
Contracts for Procedural Programs
Underlying 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

<table>
<thead>
<tr>
<th></th>
<th>Obligations</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Landowner</strong></td>
<td>Provide 5 acres of land; pay for building if completed in time</td>
<td>Get building in less than six months</td>
</tr>
<tr>
<td><strong>Builder</strong></td>
<td>Build house on provided land in less than six month</td>
<td>No need to do anything if provided land is smaller than 5 acres; Receive payment if house finished in time</td>
</tr>
</tbody>
</table>
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*

\[
\text{Precondition}(State) \Rightarrow \text{Postcondition}(\text{procedure}(State))
\]

- **Notation**: \{Precondition\} procedure \{Postcondition\}
- **Assertions** stated in first-order predicate logic
Example

Consider the following procedure:

```c
/**
 * @param a an integer
 * @return 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: \( a \in \text{integer} \) and \( \text{root} \in \text{integer} \) (the result)

1. root as a partial function
   
   **Precondition**: \( a \geq 0 \)
   
   **Postcondition**: \( \text{root} \times \text{root} \leq a < (\text{root} + 1) \times (\text{root} + 1) \)

2. root as a total function
   
   **Precondition**: \textbf{true}
   
   **Postcondition**:
   
   \[
   (a \geq 0 \implies \text{root} \times \text{root} \leq a < (\text{root} + 1) \times (\text{root} + 1)) \wedge \\
   (a < 0 \implies \text{root} = 0)
   \]
Weakness and Strength

Goal:

- find weakest precondition
  a precondition that is implied by all other preconditions
  highest demand on procedure
  largest domain of procedure
  (Q: what if precondition = false?)

- find strongest postcondition
  a postcondition that implies all other postconditions
  smallest range of procedure
  (Q: what if postcondition = true?)

Met by “root as a total function”:

- true is weakest possible precondition
- “defensive programming”
Example (Weakness and Strength)

Consider root as a function over integers

**Precondition:** true

**Postcondition:**

\[(a \geq 0 \implies root \times root \leq a < (root + 1) \times (root + 1)) \wedge (a < 0 \implies root = 0)\]

▶ true is the weakest precondition

▶ The postcondition can be strengthened to

\[(root \geq 0) \wedge (a \geq 0 \implies root \times root \leq a < (root + 1) \times (root + 1)) \wedge (a < 0 \implies root = 0)\]
An Example

Insert an element in a table of fixed size

```java
class TABLE<T> {
    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

\[ \text{count} < \text{capacity} \]

**Postcondition:** new element in table, count updated

\[ \text{count} \leq \text{capacity} \]
\[ \land \text{get(key)} = \text{element} \]
\[ \land \text{count} = \text{old count} + 1 \]
<table>
<thead>
<tr>
<th></th>
<th><strong>Obligations</strong></th>
<th><strong>Benefits</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Caller</strong></td>
<td>Call put only on non-full table</td>
<td>Get modified table in which element is associated with key</td>
</tr>
<tr>
<td><strong>Procedure</strong></td>
<td>Insert element in table so that it may be retrieved through key</td>
<td>No need to deal with the case where table is full before insertion</td>
</tr>
</tbody>
</table>
Contracts for Object-Oriented Programs
Contracts for Object-Oriented Programs

Contracts for methods have additional features

- 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: Class Invariant

- class invariant $INV$ is predicate that holds for all objects of the class
  - must be established by all constructors
  - must be maintained by all public methods
Pre- and Postconditions for Methods

- constructor methods $c$

  $$\{\text{Pre}_c\} \; c \; \{\text{INV}\}$$

- visible methods $m$

  $$\{\text{Pre}_m \land \text{INV}\} \; m \; \{\text{Post}_m \land \text{INV}\}$$
Table example revisited

- count and capacity are instance variables of class TABLE
- $INV_{TABLE}$ is $count \leq capacity$
- specification of $void$ put ($T$ element, $String$ key)
  
  **Precondition:**
  
  $count < capacity$

  **Postcondition:**

  $get(key) = element \land count = old \: count + 1$
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
  \[\Rightarrow \text{method specialization}\]
- Relation to invariant and pre-, postconditions in base class?
- 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$_{TABLE}$ must hold for o

⇒ INV$_{MYTABLE}$ ⇒ INV$_{TABLE}$

► Example: MYTABLE might be a hash table with invariant

$$ INV_{MYTABLE} \equiv \text{count} \leq \frac{\text{capacity}}{3} $$
Method Specialization

If MYTABLE redefines put then . . .

- the **precondition** in the subclass **must be weaker** and
- the **postcondition** in the subclass **must be stronger**

than in the superclass because in

```java
TABLE personnel = new MYTABLE (150);
...
personnel.put (new Terminator (3), "Arnie");
```

the caller

- only guarantees $\text{Pre}_{\text{put},\text{Table}}$
- and expects $\text{Post}_{\text{put},\text{Table}}$
Requirements for Method Specialization

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

- $\text{Pre}_{T,m} \Rightarrow \text{Pre}_{S,m}$ and
- $\text{Post}_{S,m} \Rightarrow \text{Post}_{T,m}$ and
- $\text{Exc}_{S,m} \subseteq \text{Exc}_{T,m}$

each exception thrown by $S.m$ may also be thrown by $T.m$
Example: MYTABLE.put

- \( \text{Pre}_{\text{MYTABLE,put}} \equiv \text{count} < \text{capacity}/3 \)
  - not a sound method specialization because it is not implied by \( \text{count} < \text{capacity} \).

- MYTABLE may automatically resize the table, so that \( \text{Pre}_{\text{MYTABLE,put}} \equiv \text{true} \)
  - is a sound method specialization because \( \text{count} < \text{capacity} \Rightarrow \text{true}! \)

- Suppose MYTABLE adds a new instance variable \( T \) \( \text{lastInserted} \) that holds the last value inserted into the table.
  
\[
\text{Post}_{\text{MYTABLE,put}} \equiv \begin{align*}
\text{item(key)} &= \text{element} \\
\land \quad \text{count} &= \text{old count} + 1 \\
\land \quad \text{lastInserted} &= \text{element}
\end{align*}
\]

- is a sound method specialization because
  \( \text{Post}_{\text{MYTABLE,put}} \Rightarrow \text{Post}_{\text{TABLE,put}} \)
Interlude: Method Specialization since Java 5

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

Example: Assume B extends A

```java
class Original {
    A m () {
        return new A();
    }
}
class Specialization extends Original {
    B m () { // overrides method Original.m()
        return new B();
    }
}
```
Interlude: NO Specialization

- Method specialization interferes with overloading in Java
- Class Specialization has two different methods

Example: Assume B extends A

```java
class Original {
    void m (B x) {
        return;
    }
}
class Specialization extends Original {
    void m (A x) {
        // does NOT override method Original.m()
        return;
    }
}
```
Contract Monitoring
Contract Monitoring

- What happens if a system’s execution violates an assertion at runtime?
- A violating execution runs outside the system’s specification.
- The system’s reaction may be \textit{arbitrary}
  - crash
  - continue
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

- evaluates assertions at run time
- raises an exception indicating any violation
- assign blame for the violation
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

- evaluates assertions at run time
- raises an exception indicating any violation
- assign blame for the 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 in class $S$
need to check (for all superclasses $T$ of $S$)

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

how?
Hierarchy Checking

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

- **on entry**
  - if $\text{Pre}_{T,m}$ holds, then $\text{Pre}_{S,m}$ must hold, too
  - $\text{Pre}_{S,m}$ must hold

- **If the precondition of $S$ is not fulfilled, but the one of $T$ is, then this is a wrong method specialization.**

- **on exit**
  - $\text{Post}_{S,m}$ must hold
  - if $\text{Post}_{S,m}$ holds, then $\text{Post}_{T,m}$ must hold, too

- **In general, with more than two classes:**
  - Cascade of implications between $S$ and $T$ must be checked.
  - All intermediate pre- and postconditions must be checked.
Examples

```java
interface IConsole {
    @post { getMaxSize > 0 }
    int getMaxSize();

    @pre { s.length () < this.getMaxSize() }
    void display (String s);
}

class Console implements IConsole {
    @post { getMaxSize > 0 }
    int getMaxSize () { ... }

    @pre { s.length () < this.getMaxSize() }
    void display (String s) { ... }
}
```
class RunningConsole extends Console {
  @pre { true }
  void display (String s) {
    ... super.display(String. substring (s, ..., ... + getMaxSize()))
    ...
  }
}
A Bad Extension

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

▶ caller may only guarantee IConsole’s precondition
▶ Console.display can be called with argument that is too long
▶ blame the programmer of PrefixedConsole!
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
Verification of Contracts

- Given: Specification of imperative procedure by **Precondition** and **Postcondition**
- Goal: Formal proof for
  \[ \text{Precondition}(State) \Rightarrow \text{Postcondition}(\text{procedure}(State)) \]
  if \( \text{procedure}(State) \) terminates
- Method: **Hoare Logic**, i.e., a proof system for **Hoare triples** of the form
  \{\text{Precondition}\} \text{ procedure } \{\text{Postcondition}\}
- named after C.A.R. Hoare, inventor of Quicksort, CSP, and many other
- here: method bodies, no recursion, no pointers (extensions exist)
Syntax of While
A small language to illustrate verification

\[
E ::= c \mid x \mid E + E \mid \ldots \quad \text{expressions}
\]

\[
B, P, Q ::= \neg B \mid P \land Q \mid P \lor Q \mid E = E \mid E \leq E \mid \ldots \quad \text{boolean expressions}
\]

\[
C, D ::= x = E \quad \text{assignment}
\]

\[
| C; D \quad \text{sequence}
\]

\[
| \text{if } B \text{ then } C \text{ else } D \quad \text{conditional}
\]

\[
| \text{while } B \text{ do } C \quad \text{iteration}
\]

\[
\mathcal{H} ::= \{P\}C\{Q\} \quad \text{Hoare triples}
\]

- (boolean) expressions are free of side effects
Proof Rules for Hoare Triples

- Instead: define axioms and inferences rules
- Construct a derivation to prove the triple
- Choice of axioms and rules guided by structure of $C$
Skip Axiom

\{ P \} \text{skip} \{ P \}
Assignment Axiom

\[ \{ P[x \mapsto E] \} \ x = E \ \{ P \} \]

Examples:

- \( \{ 1 == 1 \} \ x = 1 \ \{ x == 1 \} \)
- \( \{ odd(1) \} \ x = 1 \ \{ odd(x) \} \)
- \( \{ x == 2 \ast y + 1 \} \ y = 2 \ast y \ \{ x == y + 1 \} \)
Sequence Rule

\[
\begin{array}{c}
\{P\} \mathcal{C} \{R\} \quad \{R\} \mathcal{D} \{Q\} \\
\{P\} \mathcal{C};\mathcal{D} \{Q\}
\end{array}
\]

Example:

\[
\{x \equiv 2 \cdot y + 1\} \quad y = 2 \cdot y \quad \{x \equiv y + 1\} \quad \{x \equiv y + 1\} \quad y = y + 1 \quad \{x \equiv y\}
\]

\[
\{x \equiv 2 \cdot y + 1\} \quad y = 2 \cdot y; \quad y = y + 1 \quad \{x \equiv y\}
\]
Conditional Rule

\[
\begin{align*}
\{ P \land B \} & \quad C \quad \{ Q \} \quad \{ P \land \neg B \} & \quad D \quad \{ Q \} \\
\{ P \} & \quad \text{if} \quad B \quad \text{then} \quad C \quad \text{else} \quad D \quad \{ Q \}
\end{align*}
\]
Conditional Rule — Issues

Examples:

\[
\begin{align*}
\{ P \land x < 0 \} & \quad z = -x \quad \{ z == |x| \} \\
\{ P \land x \geq 0 \} & \quad z = x \quad \{ z == |x| \}
\end{align*}
\]

\[
\{ P \} \quad \text{if } x < 0 \text{ then } z = -x \text{ else } z = x \quad \{ z == |x| \}
\]

▶ incomplete!
▶ precondition for \( z = -x \) should be \((z == |x|)[z \mapsto -x] \equiv -x == |x|\)
⇒ need logical rules
Logical Rules

- **weaken precondition**

\[
\frac{P' \Rightarrow P}{\{P'\} C \{Q\}}
\]

- **strengthen postcondition**

\[
\frac{\{P\} C \{Q\} \quad Q \Rightarrow Q'}{\{P\} C \{Q'\}}
\]

- Example needs strengthening: \( P \land x < 0 \Rightarrow -x == |x| \)
- holds if \( P \equiv \text{true} \)!
- similarly: \( P \land x \geq 0 \Rightarrow x == |x| \)
Completed example:

\[ D_1 = \frac{x < 0 \Rightarrow -x == |x| \quad \{ -x == |x| \} \quad z = -x \quad \{ z == |x| \}} {x < 0 \quad z = -x \quad \{ z == |x| \}} \]

\[ D_2 = \frac{x \geq 0 \Rightarrow x == |x| \quad \{ x == |x| \} \quad z = x \quad \{ z == |x| \}} {x \geq 0 \quad z = x \quad \{ z == |x| \}} \]

\[ \{ x < 0 \} \quad z = -x \quad \{ z == |x| \} \quad \{ x \geq 0 \} \quad z = x \quad \{ z == |x| \} \]

\[ \{ \text{true} \} \text{ if } x < 0 \text{ then } z = -x \text{ else } z = x \quad \{ z == |x| \} \]
While Rule

\[
\begin{align*}
\{P \land B\} & C \{P\} \\
\{P\} \text{ while } B \text{ do } C \{P \land \neg B\}
\end{align*}
\]

▶ \( P \) is loop invariant

Example: try to prove

\[
\{ a\geq 0 \land i==0 \land k==1 \land \text{sum}==1 \}
\]
while \( \text{sum} \leq a \) do

\begin{align*}
& k = k+2; \\
& i = i+1; \\
& \text{sum} = \text{sum}+k
\end{align*}

\[
\{ i*i \leq a \land a < (i+1)*(i+1) \}
\]

⇒ while rule not directly applicable …
While Rule

Step 1: Find the loop invariant

\[ a \geq 0 \land i = 0 \land k = 1 \land \text{sum} = 1 \]
\[ \Rightarrow \]
\[ i \cdot i \leq a \land i \geq 0 \land k = 2 \cdot i + 1 \land \text{sum} = (i+1) \cdot (i+1) \]

\[ P \equiv i \cdot i \leq a \land i \geq 0 \land k = 2 \cdot i + 1 \land \text{sum} = (i+1) \cdot (i+1) \]
holds on entry to the loop

To prove that \( P \) is an invariant, requires to prove that

\[ \{ P \land \text{sum} \leq a \} \ k = k + 2 ; \ i = i + 1 ; \ \text{sum} = \text{sum} + k \ \{ P \} \]

It follows by the sequence rule and weakening:
Proof of loop invariance

\[
\begin{align*}
\{ & \ i\cdot i \leq a \ \land \ i \geq 0 \ \land \ k = 2\cdot i + 1 \ \land \ \text{sum} = (i+1)^2 \ \land \ \text{sum} \leq a \} \\
\{ & \ i \geq 0 \ \land \ k+2 = 2+2\cdot i+1 \ \land \ \text{sum} = (i+1)^2 \ \land \ \text{sum} \leq a \} \\
\{ & \ i \geq 0 \ \land \ k+2 \cdot (i+1) + 1 \ \land \ \text{sum} = (i+1)^2 \ \land \ \text{sum} \leq a \} \\
\{ & \ i+1 \geq 1 \ \land \ k = 2\cdot (i+1) + 1 \ \land \ \text{sum} = (i+1)^2 \ \land \ \text{sum} \leq a \} \\
\{ & \ \text{i = i+1} \\
\{ & \ i \geq 1 \ \land \ k = 2\cdot i + 1 \ \land \ \text{sum} = i\cdot i \ \land \ \text{sum} \leq a \} \\
\{ & \ i\cdot i \leq a \ \land \ i \geq 1 \ \land \ k=2\cdot i+1 \ \land \ \text{sum}+k = i\cdot i+k \ \land \ \text{sum}+k \leq a+k \} \\
\{ & \ \text{sum = sum+k} \\
\{ & \ i\cdot i \leq a \ \land \ i \geq 1 \ \land \ k = 2\cdot i + 1 \ \land \ \text{sum} = i\cdot i+k \ \land \ \text{sum} \leq a+k \} \\
\{ & \ i\cdot i \leq a \ \land \ i \geq 1 \ \land \ k = 2\cdot i + 1 \ \land \ \text{sum} = i\cdot i+2\cdot i+1 \ \land \ \text{sum} \leq a+k \} \\
\{ & \ i\cdot i \leq a \ \land \ i \geq 1 \ \land \ k = 2\cdot i + 1 \ \land \ \text{sum} = (i+1)^2 \ \land \ \text{sum} \leq a+k \} \\
\{ & \ i\cdot i \leq a \ \land \ i \geq 0 \ \land \ k = 2\cdot i + 1 \ \land \ \text{sum} = (i+1)^2 \} 
\end{align*}
\]
Step 2: Apply the while rule

\[
\begin{align*}
\{ P \land sum \leq a \} & \quad k = k + 2; \quad i = i + 1; \quad sum = sum + k \quad \{ P \} \\
\{ P \} & \quad while \quad sum \leq a \quad do \quad k = k + 2; \quad i = i + 1; \quad sum = sum + k \quad \{ P \land sum > a \} 
\end{align*}
\]

Now, \( P \land sum > a \) is

\[
\{ \text{i*i<=a } \land \text{i>=0 } \land \text{k==2*i+1 } \land \text{sum==(i+1)*(i+1) } \land \text{sum>a } \}
\]

implies

\[
\{ \text{i*i<=a } \land \text{a<(i+1)*(i+1) } \}
\]
Soundness of the Rules

- Intuitively, the proof rules are ok.
- But are they sound?
- Is there a definition from which $\{P\} C \{Q\}$ can be proved directly?
- Answer: Yes!
- Each rule can be proved correct from this definition.
Semantics — Domains and Types

- \( BValue = \text{true} \mid \text{false} \)
- \( IValue = 0 \mid 1 \mid \ldots \)
- \( \sigma \in State = \text{Variable} \rightarrow \text{Value} \)

- \( E[] : Expression \times State \rightarrow IValue \)
- \( B[] : \text{BoolExpression} \times State \rightarrow BValue \)
- \( S[] : State_\bot \rightarrow State_\bot \)

- \( State_\bot := State \cup \{\bot\} \)
- result \(\bot\) indicates non-termination
Semantics — Expressions

\[ E[\llbracket c \rrbracket \sigma] = c \]
\[ E[\llbracket x \rrbracket \sigma] = \sigma(x) \]
\[ E[\llbracket E + F \rrbracket \sigma] = E[\llbracket E \rrbracket \sigma] + E[\llbracket F \rrbracket \sigma] \]

\[ \ldots \]
\[ B[\llbracket E = F \rrbracket \sigma] = E[\llbracket E \rrbracket \sigma] = E[\llbracket F \rrbracket \sigma] \]
\[ B[\llbracket \neg B \rrbracket \sigma] = \neg B[\llbracket B \rrbracket \sigma] \]

\[ \ldots \]
Semantics — Statements

\[
\begin{align*}
S[C] \parallel & = \bot \\
S[\text{skip}] \sigma & = \sigma \\
S[x = E] \sigma & = \sigma[x \mapsto E[\sigma]] \\
S[C; D] \sigma & = S[D](S[C] \sigma) \\
S[\text{if } B \text{ then } C \text{ else } D] \sigma & = B[B] \sigma = \text{true} \rightarrow S[C] \sigma , S[D] \sigma \\
S[\text{while } B \text{ do } C] \sigma & = F(\sigma) \\
\text{where } & F(\sigma) = B[B] \sigma = \text{true} \rightarrow F(S[C] \sigma) , \sigma
\end{align*}
\]

- McCarthy conditional: \( b \rightarrow e_1, e_2 \)
Proving a Hoare triple

Theorem

\[ \{ P \} \ C \ { Q \} \]

- holds if \((\forall \sigma \in State)\ P(\sigma) \Rightarrow (Q(S\llbracket C \rrbracket \sigma) \lor S\llbracket C \rrbracket \sigma = \bot)\) (partial correctness)

- alternative reading/notation: \(P, Q \subseteq State\)
  \[ \{ P \} \ C \ { Q \} \equiv S\llbracket C \rrbracket P \subseteq Q \cup \bot \]

- reading predicates as boolean expressions
  \(B\llbracket P \rrbracket \sigma = \text{true} \Rightarrow (B\llbracket Q \rrbracket (S\llbracket C \rrbracket \sigma) = \text{true} \lor S\llbracket C \rrbracket \sigma = \bot)\)

Proof

By induction on the derivation of \(\{ P \} \ C \ { Q \} :\)
For each Hoare rule, if the above hypothesis holds for the assumptions, then it holds for the conclusion.
Skip Axiom — Correctness

\[ \{ P \} \text{skip} \{ P \} \]

Correctness

- \( S[\text{skip}]\sigma = \sigma \)
- Assume \( B[P]\sigma = \text{true} \). Then \( B[P](S[\text{skip}]\sigma) = B[P]\sigma = \text{true} \)
Assignment Axiom — Correctness

\[
\{ P[x \mapsto E] \} \ x = E \ {P}
\]

- Semantics: \( S[\![x=E]\!] \sigma = \sigma[\![x \mapsto E]\!] \)

- Under assumption \( B[\![P[x \mapsto E]\!] \sigma = \text{true} \) show that
  \( (B[\![P]\!] (S[\![x = E]\!] \sigma) = \text{true} \lor S[\![x = E]\!] \sigma = \bot) \)
  \( \leftrightarrow (B[\![P]\!] (\sigma[\![x \mapsto E]\!] \sigma)] = \text{true} \lor S[\![x = E]\!] \sigma = \bot) \)

- Requires induction on \( P \):
Assignment Axiom — Correctness II

- Prove $B[P[x \mapsto E]]\sigma = B[P](\sigma[x \mapsto E][E]\sigma)$ by induction on $P$.
- Case $P \equiv \neg Q$:
  $$B[\neg Q[x \mapsto E]]\sigma \overset{\text{def}}{=} \neg B[Q[x \mapsto E]]\sigma \overset{IH}{=} \neg B[Q](\sigma[x \mapsto E][E]\sigma) \overset{\text{def}}{=} B[\neg Q](\sigma[x \mapsto E][E]\sigma)$$
- Cases $P \equiv Q \land Q'$ and $P \equiv Q \lor Q'$ analogously.
- Case $P \equiv E' = E''$:
  $$B[(E' = E'')[x \mapsto E]]\sigma \overset{\text{def}}{=} (E'[E'][x \mapsto E])\sigma = E''[E'][x \mapsto E]\sigma$$
  Need another lemma:
  $$E'[E'][x \mapsto E]\sigma = E'[E']\sigma[x \mapsto E][E]\sigma$$
  $$= (E'E')\sigma[x \mapsto E][E]\sigma = E'E'\sigma[x \mapsto E][E]\sigma$$
  $$\overset{\text{def}}{=} E'[E' = E'']\sigma[x \mapsto E][E]\sigma$$
- Case $P \equiv E' \leq E''$ etc: analogously.
Assignment Axiom — Correctness III

Remains to show that $\mathcal{E}[E'[x \mapsto E]] \sigma = \mathcal{E}[E'] \sigma[x \mapsto \mathcal{E}[E] \sigma]$ by induction on $E'$.

- **Case $E' \equiv x$:**
  \[ \mathcal{E}[x[x \mapsto E]] \sigma = \mathcal{E}[E] \sigma = \mathcal{E}[x] \sigma[x \mapsto \mathcal{E}[E] \sigma] \]

- **Case $E' \equiv y$, $y \neq x$:**
  \[ \mathcal{E}[y[x \mapsto E]] \sigma = \mathcal{E}[y] \sigma = \sigma(y) = \sigma[x \mapsto \mathcal{E}[E] \sigma](y) = \mathcal{E}[y] \sigma[x \mapsto \mathcal{E}[E] \sigma] \]

- **Case $E' \equiv -E'':$ Immediate by induction.**
  \[ \mathcal{E}[-E''[x \mapsto E]] \sigma \overset{\text{def}}{=} -\mathcal{E}[E''[x \mapsto E]] \sigma \overset{\text{IH}}{=} -\mathcal{E}[E''] \sigma[x \mapsto \mathcal{E}[E] \sigma] \overset{\text{def}}{=} \mathcal{E}[-E''] \sigma[x \mapsto \mathcal{E}[E] \sigma] \]

- **Case $E' \equiv E'' + E'''$ etc: analogously.**
Sequence Rule — Correctness

\[
\begin{array}{c}
\{P\} \ C \ \{R\} \\
\{R\} \ D \ \{Q\}
\end{array}
\]

\[
\{P\} \ C;D \ \{Q\}
\]

Proof

- Assume \( B[P] \sigma = \text{true} \)
- Induction on \( \{P\} \ C \ \{R\} \) yields
  \( B[R](S[C] \sigma) = \text{true} \lor S[C] \sigma = \bot \)
- If \( S[C] \sigma = \bot \) then the rule is correct because \( S[C;D] \sigma = \bot \).
- Otherwise: induction on \( \{R\} \ C \ \{Q\} \) yields
  \( B[Q](S[D](S[C] \sigma)) = \text{true} \lor S[D](S[C] \sigma) = \bot \)
- Recall that \( S[D](S[C] \sigma) \overset{\text{def}}{=} S[C;D] \sigma \)
- If \( S[D](S[C] \sigma) = \bot \) then the rule is correct because \( S[C;D] \sigma = \bot \).
- Otherwise: \( B[Q](S[C;D] \sigma) = \text{true} \) QED
Conditional Rule — Correctness

\[
\{ P \land B \} \text{ } C \{ Q \} \quad \{ P \land \neg B \} \text{ } D \{ Q \}
\]

\[
\{ P \} \text{ if } B \text{ then } C \text{ else } D \{ Q \}
\]

Correctness

- Show: \( \sigma \in P \) implies \( S[\text{if } B \text{ then } C \text{ else } D] \in Q \cup \{ \bot \} \)
- Exercise
Logical Rules — Correctness

- **weaken precondition**

\[
P' \Rightarrow P \quad \{P\} C \{Q\} \quad \{P'\} C \{Q\}
\]

- **strengthen postcondition**

\[
\{P\} C \{Q\} \quad Q \Rightarrow Q' \quad \{P\} C \{Q'\}
\]

**Correctness**

\[
P' \Rightarrow P \text{ iff } P' \subseteq P \text{ (as set of states)}
\]
While-Rule — Correctness

\[
\{P \land B\} \quad C \quad \{P\} \\
\{P\} \text{ while } B \text{ do } C \quad \{P \land \neg B\}
\]

- Consider the semantics of while: 
  \[S[\text{while } B \text{ do } C]\sigma = F(\sigma)\]
  where \(F(\bot) = \bot\) and \(F(\sigma) = B[B]\sigma = \text{true} \rightarrow F(S[C]\sigma), \sigma\)

- It is sufficient to show (fixpoint induction):
  \[
  \text{If } (\forall \sigma \in P), \ F(\sigma) \in P \land \neg B \lor \{\bot\} \\
  \text{then } (\forall \sigma \in P), \ B[B]\sigma = \text{true} \rightarrow F(S[C]\sigma), \sigma \in P \land \neg B \lor \{\bot\}
  \]
  - Case \(B[B]\sigma = \text{true}\):
    By induction on \(\{P \land B\} \quad C \quad \{P\}\),
    either \(S[C]\sigma = \bot\) (then \(F(S[C]\sigma) = F(\bot) = \bot\) completes the proof),
    or \(S[C]\sigma \in P\) (then \(F(S[C]\sigma) \in P \land \neg B \lor \{\bot\}\) completes the proof)
  - Case \(B[B]\sigma = \text{false}\):
    Then \(\sigma \in P \land \neg B\). QED
Properties of Formal Verification

- requires more restrictions on assertions (e.g., use a certain logic) than monitoring
- full compliance of code with specification can be guaranteed
- scalability and expressivity are challenging research topics:
  - full automatization
  - manageable for small/medium examples
  - large examples require manual interaction
  - real programs use dynamic datastructures (pointers, objects) and concurrency