Specifications, continued

**Dilbert**
by Scott Adams

---

WALLY: I CAME TO ASK YOU FOR THE NEW DESIGN Specs.

BUT WE BOTH KNOW YOU'LL SEND ME TO SOMEONE WHO DOESN'T HAVE THEM, AND THAT PERSON WILL REFER ME BACK TO YOU.

WHEN I RETURN, YOU WILL HAVE ESCAPED TO YOUR SECRET HIDING PLACE.

TED HAS THE SPECS.

© Scott Adams, Inc./Dist. by UFS, Inc.
Review

• Spec “A is stronger than B” means
  • For every implementation 𝐈
    • “𝐈 satisfies A” implies “𝐈 satisfies B”
    • If the implementation satisfies the stronger spec (A), it satisfies the weaker (B)
    • The opposite is not necessarily true!
  • For every client 𝐂
    • “𝐂 meets the obligations of B” implies “𝐂 meets the obligations of A”
    • If C meets the weaker spec (B), it meets the stronger spec (A)
    • The opposite is not necessarily true

• A **larger world** of implementations satisfy the weaker spec B than the stronger spec A

• Consequently, it is easier to implement a weaker spec!
  • Weaker specs require *more* AND/OR Weaker specs guarantee (promise) *less*
Satisfaction of Specifications

• I is an implementation and S is a specification
• I satisfies S if
  • Every behavior of I is permitted by S
  • No behavior of I violates S
• The statement “I is correct” is meaningless, but often used
• If I does not satisfy S, either or both could be wrong
  • I does something that S doesn’t specify
  • S expects a result that I doesn’t produce
• When I doesn’t satisfy S, it’s usually better to change the program rather than the spec.
• If spec is too complex modify spec
Why Compare Specs?

- Liskov Substitution Principle
  - We want to use a subclass method in place of superclass method
  - Spec of subclass method must be stronger
    - Or at least equally strong
- Which spec is stronger?
  - A procedure satisfying a stronger spec can be used anywhere a weaker spec is required.

- Does the implementation satisfy the specification?
Comparing Specifications

• One way: by hand, examine each clause
• Another way: logical formulas representing the spec
• Use whichever is most convenient

• Comparing specs enables reasoning about substitutability
Exercise

- Specification A:
  requires: \(a\) is non-null and \(value\) occurs in \(a\)
  modifies: none
  effects: none
  returns: the smallest index \(i\) such that \(a[i] = value\)

- Specification B:
  requires: \(a\) is non-null and \(value\) occurs in \(a\) // same as A
  modifies: none // same as A
  effects: none // same as A
  returns: \(i\) such that \(a[i] = value\) // fewer guarantees

• Therefore, A is stronger.
• In fact, A’s postcondition implies B’s postcondition
Example

• Specification B:
  • requires: \( a \) is non-null and \( \text{value} \) occurs in \( a \)
  • modifies: none
  • effects: none
  • returns: \( i \) such that \( a[i] = \text{value} \)

• Specification A:
  • requires: \( a \) is non-null // fewer conditions!
  • modifies: none // same
  • effects: none // same
  • returns: \( i \) such that \( a[i] = \text{value} \) if value occurs in \( a \) and \( i = -1 \) if value is not in \( a \) // guarantees more!

• Therefore, A is stronger!
Strong Versus Weak Specifications

• double sqrt(double x)
  
  A. @requires x>= 0  
     @return y such that |y^2 – x| <= 1
  
  B. @requires none  
     @return y such that |y^2 – x| <= 1  
     @throws IllegalArgumentException if x < 0
  
  C. @requires x>= 0  
     @return y such that |y^2 – x| <= 0.1
  
• Which are stronger?
Comparing Specifications

Most of our specification comparisons will be informal

A is stronger than B if

A’s precondition is weaker than B’s
(keeping postcondition the same)
  - Requires less of client

Or

A’s postcondition is stronger than B’s
(keeping precondition the same)
  - Guarantees more to client

Or

A’s precondition is weaker than B’s
AND
A’s postcondition is stronger than B’s
Comparing by Logical Formulas

• Specification S1 is stronger than S2 iff
  • For all implementations I, (I satisfies S1) => (I satisfies S2)
  • The set of implementations that satisfy S1 is a *subset* of the set of implementations satisfying S2.

• If each specification is a logical formula
  • S1 => S2

• Comparison using logical formulas is precise but can be difficult to carry out.

• It is often difficult to express all preconditions and postconditions with precise logical formulas!
# Implication Truth Table

<table>
<thead>
<tr>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_1 \Rightarrow S_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

### Truth Tables for Connectives

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>P \land Q</th>
<th>P \lor Q</th>
<th>P \Rightarrow Q</th>
<th>P \iff Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>
Comparing by Logical Formulas

- S1 is stronger than S2
- \((x \text{ is an element of set of programs satisfying } S1) \implies (x \text{ is an element of the set of programs satisfying } S2)\)
  - the set of programs satisfying S1 is a subset of the set of programs satisfying S2
  - "A is a subset of B" if and only if every element of A also belongs to B

- An implementation I that satisfies S1 also satisfies S2

- If \((I \text{ satisfies } S1) \implies (I \text{ satisfies } S2)\) is false
  - Then S1 does not imply S2, or S1 is not stronger than S2.
  - If I does not satisfy S1, all bets are off. I might or might not satisfy S2.
  - See \(\text{http://press.princeton.edu/chapters/s8898.pdf}\)
Comparing by Logical Formulas

• Let $\text{Spec } A: \{P_A\} \text{ code } \{Q_A\}$,
  
  $\text{Spec } B: \{P_B\} \text{ code } \{Q_B\}$.

We say code satisfies a specification with precondition $P$ and postcondition $Q$ iff $\{P\} \text{ code } \{Q\}$ Hoare triple is true.

Do not confuse it with $P \Rightarrow Q$.

e.g., $\{ \text{true} \} x = 1 ; \{ x = 1 \}$ is true, but

true $\Rightarrow x = 1$ is false.
Comparing by Logical Formulas

• Let Spec A: \{P_A\} code \{Q_A\},
  Spec B: \{P_B\} code \{Q_B\}.

The following are equivalent:
  • P_B => P_A and Q_A => Q_B
  • A is stronger than B
  • A => B
Example Revisited: int find(int[] a, int val)

```java
int find(int[] a, int value) {
    for (int i = 0; i < a.length; i++) {
        if (a[i] == value) return i;
    }
    return -1;
}
```

• Specification B:
  • requires: a is non-null and value occurs in a
  • returns: i such that a[i] = value

• Specification A:
  • requires: a is non-null
  • returns: i such that a[i] = value or i = -1 if value is not in a
Be careful with specifications!

returns: i such that a[i] = value or i = -1 if value is not in a

Let P = “val occurs in a”,
Q = “return i s.t. a[i] = val”
R = “return -1”

(P => Q) v (!P => R)

= (!P v Q) v (P v R)

= Q v R

“or” would allow us to write a method that always returns -1!
Be careful with specifications!

\[\text{returns: } i \text{ such that } a[i] = \text{value} \text{ or } i = -1 \text{ if value is not in } a\]

We really mean: “\(i \text{ such that } a[i] = \text{value} \text{ if value is in } a, \text{ AND } i = -1 \text{ if value is not in } a\).”

\[P \Rightarrow Q \land \neg P \Rightarrow R \text{ is equivalent to: } (P \land Q) \lor (\neg P \land R) \lor (Q \land R)\]

In our case, “\(P \Rightarrow Q \land \neg P \Rightarrow R\)” and “\((P \land Q) \lor (\neg P \land R)\)” are equivalent since “\(Q \land R\)” is false (return -1 and return a value \(\geq 0\) cannot both be true.)

So, we could also say: “\((i \text{ such that } a[i] = \text{value} \text{ and value is in } a) \text{ or } (i = -1 \text{ and value is not in } a)\).”
Example: int find(int[] a, int val)

• Specification B:
  requires: a is non-null and val occurs in a [P_B]
  returns: i such that a[i] = val [Q_B]

• Specification A:
  requires: a is non-null [P_A]
  returns: i such that a[i] = val if value val occurs in a and -1 if value val does not occur in a [Q_A]

Clearly, P_B => P_A.

Q_A states “val occurs in a => returns i such that a[i]=val AND val does not occur in a => returns -1“

Q_B can be logically rewritten as: “val occurs in a => returns i such that a[i]=val AND val does not occur in a => returns anything.” (violated precondition allows anything.)
Comparing postconditions

• $Q_B$ (postcondition of Spec B)
  
i such that $a[i] == \text{value}$ can be written (due to the precondition) as:
  
  $\text{value is in a} \Rightarrow i$ such that $a[i] == \text{value}$
  
  $\&\& \text{value is not in a} \Rightarrow \text{true}$

• $Q_A$ (postcondition of Spec A)
  
  $\text{value is in a} \Rightarrow i$ such that $a[i] == \text{value}$
  
  $\&\& \text{value is not in a} \Rightarrow -1=i$

Q_B and Q_A are NOT:

$Q_{B2}: \{0 \leq i < a.\text{length}\}$

$Q_{A2}: \{-1 \leq i < a.\text{length}\}$

For these, $Q_{B2} \Rightarrow Q_{A2}$, i.e., $Q_{B2}$ is stronger

• Which is stronger, $Q_B$ or $Q_A$?
Comparing by Logical Formulas

Let \( A = \{P_A\} \) code \( \{Q_A\} \),
\[ B = \{P_B\} \text{ code } \{Q_B\} \]
be Hoare triples.

\( A \) is stronger than \( B \) if and only if \( P_A \) is weaker than \( P_B \) and \( Q_A \) is stronger than \( Q_B \), i.e.,

- \( A \Rightarrow B \iff (P_B \Rightarrow P_A \land Q_A \Rightarrow Q_B) \).

\( A \Rightarrow B \) means that any code satisfying \( A \) also satisfies \( B \).
Example: int find(int[] a, int val)

• Specification B:
  requires: \( a \) is non-null and \( \text{val} \) occurs in \( a \) \( [P_B] \)
  returns: \( i \) such that \( a[i] = \text{val} \) \( [Q_B] \)

• Specification A:
  requires: \( a \) is non-null \( [P_A] \)
  returns: \( i \) such that \( a[i] = \text{val} \) if value \( \text{val} \) occurs in \( a \) and \(-1\) if value \( \text{val} \) does not occur in \( a \) \( [Q_A] \)

• \( P_B \) requires more of the caller than \( P_A \). That is, \( P_B \Rightarrow P_A \).

• \( Q_A \) promises more to the caller than \( Q_B \) (\( Q_B \) does not promise anything if \( \text{val} \) does not occur in \( a \); e.g., code satisfying B could return \(-99\)). That is, \( Q_A \Rightarrow Q_B \).
Example: int find(int[] a, int val)

• Specification B:
  requires: a is non-null and val occurs in a \([P_B]\)
  returns: i such that a[i] = val \([Q_B]\)

• Specification A:
  requires: a is non-null \([P_A]\)
  returns: i such that a[i] = val if val occurs in a and \(-1\) if val does not occur in a \([Q_A]\)

Intuition: \(Q_B\) should really be thought of as:
  i such that a[i] = val if val occurs in a

Thus, it’s still OK to substitute A for B.
Exercise: int find(int[] a, int val)
Sort specifications in order of strength

• Specification B:
  requires: a is non-null and val occurs in a \([P_B]\)
  returns: i such that a[i] = val \([Q_B]\)

• Specification A:
  requires: a is non-null \([P_A]\)
  returns: i such that a[i] = val if val occurs in a and -1 if val does not occur in a \([Q_A]\)

• Specification C:
  requires: none \([P_C]\)
  returns: i such that a[i] = val if val occurs in a and -1 if val does not occur in a \([Q_C]\)
  throws: NullPointerException if a is null \([Q_C]\)
Converting PSoft Specs into Logical Formulas

- PSoft specification
  
  requires: R
  modifies: M
  effects: E

is equivalent to this logical formula

\{R\} code \{E ^ (nothing but M is modified)\}

throws and returns are absorbed into effects E
Convert Spec to Formula, step 1: absorb throws and returns into effects

• PSoft specification convention
  
  requires: (unchanged)
  modifies: (unchanged)
  effects:
  returns: absorbed into “effects”
  throws:
Convert Spec to Formula, step 1: absorb throws and returns into effects

- `set` method from `java.util.ArrayList<T>`
  
  ```java
  T set(int index, T element)
  ```

  requires: true
  modifies: this[index]
  effects: this_post[index] = element
  throws: IndexOutOfBoundsException if index < 0 || index ≥ size
  returns: this_pre[index]

  Absorb effects, returns and throws into new `effects`:

  ```java
  E = if index < 0 || index ≥ size then
      throws IndexOutOfBoundsException
    else
      this_post[index] = element and returns this_pre[index]
  ```
Convert Spec to Formula, step 2: Convert into Formula

- `set from java.util.ArrayList<T>`
  
  ```java
  T set(int index, T element)
  ```

  `requires: true`
  
  `modifies: this[index]`
  
  `effects: E = if index < 0 || index ≥ size then`  
  throws IndexOutOfBoundsException  
  else
  ```java
  this_post[index] = element and returns this_pre[index]
  ```

Denote `effects` expression by `E`. Resulting formula is:

```latex
\{true\} code \{ (E \land (\text{forall } i \neq \text{index}, \text{this}_\text{post}[i] = \text{this}_\text{pre}[i])) \}\}
```
Stronger Specification

• S1 is stronger than S2 iff

\{R_1\} \text{ code } \{E_1 \wedge (\text{only } M_1 \text{ is modified})\}
=>
\{R_2\} \text{ code } \{E_2 \wedge (\text{only } M_2 \text{ is modified})\}

iff \ R_2 => R_1 \wedge (E_1 \wedge (\text{only } M_1 \text{ is modified}) => (E_2 \wedge (\text{only } M_2 \text{ is modified}))

iff \ R_2 => R_1 \wedge E_1 => E_2 \wedge (\text{only } M_1 \text{ is modified}) => (\text{only } M_2 \text{ is modified})

iff \ R_2 => R_1 \wedge E_1 => E_2 \wedge (M_1 \subseteq M_2)
Stronger Specification

• $S_1$ is stronger than $S_2$ if $R_2 \Rightarrow R_1 \Rightarrow E_1 \Rightarrow E_2 \Rightarrow (M_1 \subseteq M_2)$

• A stronger specification:
  • Requires less
  • Guarantees more
  • Modifies less
Exercise

• Convert PSoft spec into logical formula

public static int binarySearch(int[] a, int key)

requires: a is sorted in ascending order and a is non-null

modifies: none

effects: none

returns: i such that a[i] = key if such an i exists; -1 otherwise

effects: E: if key occurs in a then returns i such that a[i] = key else returns -1.

E more formally:

E = 0 <= index => index < a.Length && a[index] = value
    ^ index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != value

{ sorted(a) ^ a != null } code { E ^ (forall i :: 0 <= i < a.Length, a_pre[i] = a_post[i]) }
Exercise

static void listAdd2(List<Integer> lst1, List<Integer> lst2)

requires: lst1, lst2 are non-null. lst1 and lst2 are same size.
modifies: lst1
effects: i-th element of lst1 is replaced with the sum of
i-th elements of lst1 and lst2
returns: none

{ (lst1 != null ^ lst2 != null ^ lst1.length = lst2.length) } code
{ (forall i :: 0 <= i < lst1.length => lst1_post[i] = lst1_pre[i] + lst2_pre[i])
^ (forall i :: 0 <= i < lst2.length => lst2_post[i] = lst2_pre[i]) }
private static void swap(int[] a, int i, int j) {

    int tmp = a[j];
    a[j] = a[i];
    a[i] = tmp;
}

static void swap(int[] a, int i, int j) {

    int tmp = a[j];
    a[j] = a[i];
    a[i] = tmp;
}

{ R } code { ( E ^ (forall k :: k != i,j a_post[k] = a_pre[k]) ) }

{ a != null ^ 0 <= i,j < a.length } code
{ (a_post[i] = a_pre[i] ^ a_post[j] = a_pre[i])
 ^ (forall k :: (0 <= k < a.length ^ k != i ^ k != j) ==> a_post[k] = a_pre[k]) }
Comparison by Logical Formulas

• We often use this equivalence direction:

If $P_B \implies P_A$ and $Q_A \implies Q_B$ then $A$ is stronger than $B$
Comparing Specifications, Review

• It is not easy to compare specifications

• Comparison by hand
  • Easier but can be imprecise
  • It may be difficult to see which of two conditions is stronger

• Comparison by logical formulas
  • Accurate
  • Sometimes, it is difficult to express behaviors with precise logical formulas!
Comparing by Hand

• **Requires** clause
  • Stronger spec has **fewer** conditions in requires
  • Requires less

• **Modifies/effects** clause
  • Stronger spec modifies **fewer** objects. Stronger spec guarantees more objects stay unmodified!

• **Returns** and **throws** clauses
  • Stronger spec guarantees **more** in returns and throws clauses. They are harder to implement, but easier to use by client
  • When pre-conditions are the same: no new throws in domain
  • When pre-conditions are weaker, it may guarantee more by specific throws. (See e.g., Spec C of `find`.)

• Bottom line: Client code should not be “surprised” by behavior
BallContainer and Box

• Suppose Box is a subclass of BallContainer

Spec of BallContainer.add(Ball b)

```java
boolean add(Ball b)

requires: b non-null
modifies: this BallContainer
effects: adds b to this BallContainer if b not already in
returns: true if b is added false otherwise
```

Spec of Box.add(Ball b)

```java
boolean add(Ball b)

requires: b non-null
modifies: this Box
effects: adds b to this Box if b is not already in and Box is not full
returns: true if b is added false otherwise
```
BallContainer and Box

• A client honoring BallContainer’s spec is justified to expect that this will work:

```java
BallContainer c = new Box(100);
...
for(int i = 0; i < 20; i++) {
    Ball b = new Ball(10);
    c.add(b)
}
```
• This will fail, but if c is a BallContainer we expect it to work

• Box’ spec is not stronger than BallContainer’s. Thus Box is not substitutable for BallContainer!

• Implementation that satisfies Box specs doesn't satisfy BallContainer specs
BallContainer and Box

• BallContainer.add unconditionally adds the Balls. Box has a condition --- the Box is not full.
• Could a client coding against BallContainer expect to work on Box?
• Is Box guaranteeing more than BallContainer?
  • Box effects are weaker. Box’s effects guarantee less.

\[
\text{BallContainer.add}() \\
E = \text{if } b \text{ is_element BallContainer_pre} \\
\quad \text{return false} \\
\text{else} \\
\quad \text{BallContainer_post} = \text{BallContainer_pre U b}
\]

\[
\text{Box.add}() \\
E = \text{if } b \text{ is_element BallContainer_pre} \\
\quad \text{return false} \\
\text{else} \\
\quad \text{if } \text{Box.volume_pre} \geq \text{max_volume} \\
\quad \quad \text{return false} \\
\text{else} \\
\quad \text{Box_post} = \text{Box_pre U b}
\]
Substitutability

• Box is not what we call a true subtype of BallContainer
  • It is more limited than BallContainer.
  • A Box can only hold a limited amount;
  • A user who uses a BallContainer in their code cannot simply substitute a BallContainer with a Box and assume the same behavior in the program.
  • The code may cause the Box to fill up, but they did not have this concern when using a BallContainer.
  • For this reason, it is not a good idea to make Box extend BallContainer.

• Therefore, it is wrong to make Box a subclass of BallContainer

• An object of a true subtype should be able to do everything the superclass object can do and possibly more
Substitutability

• Box is not a true subtype (also called behavioral subtype) of BallContainer

• Bottom line:
  • Box.add() guarantees less

• Therefore, it is wrong to make Box a subclass of BallContainer

• More on substitutability, Java subtypes and true subtypes later
The Weakest Specification

requires: false
// Remember, false is the strongest condition of all

modifies: anything

effects: true
// true is the weakest condition of all

returns: true

throws: true

(This spec is so weak, it is trivial to implement, but impossible to use.)
The Strongest Specification

requires: true
// Remember, true is the weakest condition of all
modifies: none
effects: false
// false is the strongest condition of all
returns: false
throws: false

(This spec is so strong, it is impossible to implement with a terminating program.)