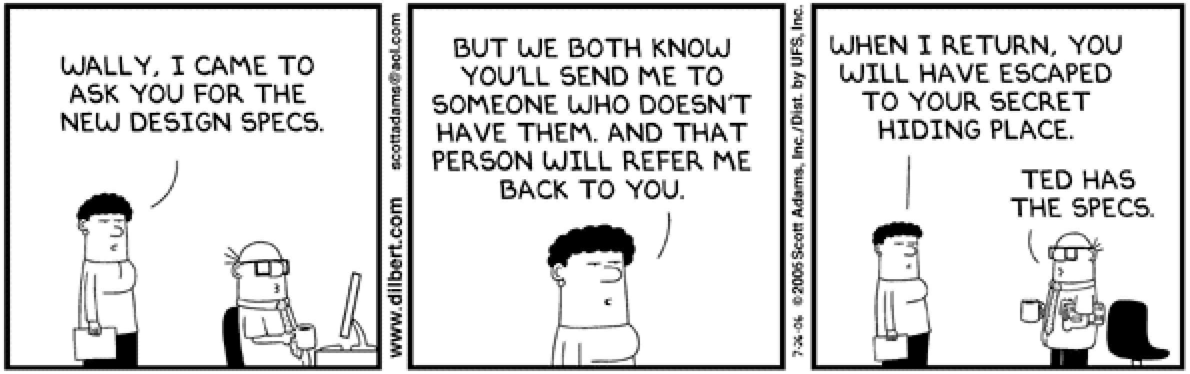
Specifications, continued

Dilbert
by Scott Adams



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 shows 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 **`value`** occurs in **`a`**
  - modifies: none
  - effects: none
  - returns: **`i`** such that **`a[i] = value`**
- Specification A:
  - requires: **`a`** is non-null // fewer conditions!
  - modifies: none // same
  - effects: none // same
  - returns: **`i`** such that **`a[i] = 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

Truth Tables for Connectives

P	Q	P∧Q	P∨Q	P→Q
True	True	True	True	True
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

S1	S2	S1=>S2
T	T	T
T	F	F
F	T	T
F	F	T

Comparing by Logical Formulas

- S1 is stronger than S2
- (x is an element of set of programs satisfying S1) => (x 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

- we say "A is a subset of B" if and only if every element of *A* also belongs to *B*

- An implementation I that satisfies S1 must satisfy S2
- if (I satisfies S1) => (I satisfies S2) is false
  - Then S1 does not imply S2
- If I does not satisfy S1, all bets are off. I might or might not satisfy S2.
  - See http://press.princeton.edu/chapters/s8898.pdf

Comparing by Logical Formulas

- The following is a sufficient condition:

  If PB => PA and QA => QB then A is stronger than B

  P_B => P_A and Q_A => Q_B

  A is stronger than B

- Too strict a requirement

Example: int find(int[] a, int val)

```
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`**

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.

But Q_A, which states "**val** occurs in **a** => returns **i** such that **a[i]=val** AND **val** does not occur in **a** => returns **-1**"

does not imply Q_B unless we take the precondition into account!

Comparing postconditions

- Q_B (postcondition of Spec B)

  i such that **a[i] == value** can be written (due to the precondition) as:

  value is in **a** => **i** such that **a[i] == value**

  && **value** is not in **a** => true

- Q_A (postcondition of Spec A)

  value is in **a** => **i** such that **a[i] == value**

  && **value** is not in **a =>** -1==i

- Which is stronger, Q_B or Q_A?

Q_B and Q_A are **NOT** those:

Q_B: {0 <= i < a.length}
Q_A: {-1 <= i < a.length}

For those, Q_B => Q_A, i.e., Q_B is stronger

Comparing postconditions

Does Q_B => Q_A?
Consider a set of all triplets (value, i, a) that make Q_B true. There are two cases.

1) **value** is in **a.** true => **i** such that **a[i] == value**. This implication is true for all such triplets (value, i, a) that make **i** such that **a[i] == value** true. false => true is true. So, this is **i** such that **a[i] == value** && true or simply **i** such that **a[i] == value**.
2) **value** is not in **a.** false => **i** such that **a[i] == value** is true (because from the implication truth table false => true is true but also false => false is true); true => true is true. So, true && true is true. This means that for all triplets (value, i, a) Q_B is true.

Now, if we take a union of triplets from 1) and 2), it would be {(value, i, a) such that **value** is in **a** and **i** such that **a[i] == value**} U {(value, i, a) such that **value** is not in **a**}. It turns out that it is not a subset of triplets that make $_{QA}$ true (see below). There are triplets that make Q_B true but Q_A false. An example would be (5, 100, [1,2,3]). For value==5, i==100, and a==[1,2,3], Q_B is true. But the same triplet makes Q_A false, so it is not in the set of triplets that make Q_A true. Clearly, true => false is false, so Q_B => Q_A is false.

Does Q_A => Q_B?
Consider a set of all triplets (value, i, a) that make Q_A true.

1) **value** is in **a.** true => **i** such that **a[i] = value**. This implication is true for all such triplets (value, i, a) that make **i** such that **a[i] = value** true. false => -1==i is true (because from the implication truth table false => true is true but also false => false is true). So, this is **i** such that **a[i] = value** && true or simply **i** such that **a[i] = value**.
2) **value** is not in **a.** false => **i** such that **a[i] = value** is true (because from the implication truth table false => true is true but also false => false is true); true => -1==i is true for i==-1. So, this is **i** such that true && -1==i is true or simply **i==-1**. This means that for all triplets (value, -1, a) such that **value** is not in **a**, Q_A is true.

Now, if we take a union of triplets from 1) and 2), it would be {(value, i, a) such that **value** is in **a** and **i** such that **a[i] == value**} U {(value, i, a) such that **value** is not in **a** and **i==-1**}. The first set in this union is the same as the first set of the union that makes Q_B true. For the second set in the union, any triplet from {(value, i, a) such that **value** is not in **a** and **i==-1**} is in {(value, i, a) such that **value** is not in **a**}. Together, it means that a set of triplets that make Q_A true is a subset of the set of triplets that make Q_B true. Therefore, Q_A => Q_B which means that Q_A is stronger than Q_B.

Comparing by Logical Formulas

$(P_A \Rightarrow Q_A) \Rightarrow (P_B \Rightarrow Q_B)$ =

$!(P_A \Rightarrow Q_A) \lor (P_B \Rightarrow Q_B)$ = [due to law $p \Rightarrow q = !p \lor q$]

$!(!P_A \lor Q_A) \lor (!P_B \lor Q_B)$ = [due to $p \Rightarrow q = !p \lor q$]

$(P_A \land !Q_A) \lor (!P_B \lor Q_B)$ = [due to $!(p \lor q) = !p \land !q$]

$(!P_B \lor Q_B) \lor (P_A \land !Q_A)$ = [due to commutativity of $\lor$]

$(!P_B \lor Q_B \lor P_A) \land (!P_B \lor Q_B \lor !Q_A)$ [distributivity]

$[\, P_B \Rightarrow (Q_B \lor P_A)\,] \land [\, (P_B \land Q_A) \Rightarrow Q_B\,]$

A is stronger than B if and only if

$P_B \Rightarrow Q_B$ is true trivially or P_B implies P_A AND

Q_A together with P_B imply Q_B (i.e., for the inputs permitted by P_B, Q_B holds)

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]**

$P_B \Rightarrow P_A$ (P_B includes P_A and one more condition)

Now, let's show $P_B \wedge Q_A \Rightarrow Q_B$.

P_B implies "**`val`** occurs in **`a`**". Q_A, states

"**`val`** occurs in **`a`** => returns **`i`** s.t. **`a[i]=val`**".

$P_B \wedge Q_A$ => "returns **`i`** s.t. **`a[i]=val`**", precisely 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_A, by itself, does not imply Q_B because A may return -1. But Q_A does imply Q_B for the inputs permitted by B. Thus, it's still OK to substitute A for B.

Converting PSoft Specs into Logical Formulas

- PSoft specification

  requires: R

  modifies: M

  effects: E

is equivalent to this logical formula

R => (E ⋀ (nothing but M is modified))

throws and returns are absorbed into effects E

⋀ means && means AND

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

- Principles of Software 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>`

`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:

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>`**

```
T set(int index, T element)
```

requires: true

modifies: this[index]

effects: E = if index < 0 || index ≥ size then
    throws IndexOutOfBoundsException
  else
    $this_{post}$[index] = element and returns $this_{pre}$[index]

Denote effects expression by E. Resulting formula is:

true => (E $\wedge$ (foreach i ≠ index, $this_{post}[i] = this_{pre}[i]$))

Stronger Specification

- S1 is stronger than S2 iff

$$(R_1 \Rightarrow E_1 \wedge (Only\, modifies\, M_1)) \Rightarrow (R_2 \Rightarrow E_2 \wedge (Only\, modifies\, M_2))$$

$$I(R_1 \Rightarrow E_1 \wedge (Only\, modifies\, M_1)) \subset I(R_2 \Rightarrow E_2 \wedge (Only\, modifies\, M_2))$$

$$The\ set\ of\ programs\ satisfying\ (R_1 \Rightarrow E_1 \wedge (Only\, modifies\, M_1))\ is\ a$$
$$subset\ of\ the\ set\ of\ programs\ satisfying\ (R_2 \Rightarrow E_2 \wedge (Only\, modifies\, M_2))$$

Exercise

- Convert Principles of Software 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

a is sorted && a is non-null => (E && (for each i, 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 AND lst2 != null AND lst1.length == lst2.length)
=> (forall i :: 0 <= i < lst1.length ==> lst1[i]_post = lst1[i]_pre + lst2[i]_pre
&& forall i :: 0 <= i < lst2.length => lst2[i]_post = lst2[i]_pre)

Exercise

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

requires: **`a`** non-null, **`0<=i,j<a.length`**

modifies: **`a[i]`** and **`a[j]`**

effects: a_{post}**`[i]=`**a_{pre}**`[j]`** and a_{post}**`[j]=`**a_{pre}**`[i]`**

returns: none

```
static void swap(int[] a, int i, int j) {
      int tmp = a[j];
      a[j] = a[i];
      a[i] = tmp;
}
```

R => (E ^ (foreach k != i,j a_{post}[k] = a_{pre}[k]))

a != null AND 0 <= i,j <=a.length
    => (a[i]_post = a[j]_pre AND a[j]_post = a[i]_pre)
        AND foreach k :: 0 <= k < a.length k != i,j -> a_{post}[k] = a_{pre}[k])

Comparison by Logical Formulas

- We often use this stricter (but simpler) test:

If P_B => P_A and Q_A => 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
  - But no new throws in domain
    - That could surprise client code
- 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)	Spec of Box.add(Ball b)
`boolean add(Ball b)`	**`boolean add(Ball b)`**
requires: **`b`** non-null	requires: **`b`** non-null
modifies: **`this`** BallContainer	modifies: **`this`** Box
effects: adds **`b`** to this BallContainer if **`b`** not already in	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	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:

```
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 <u>is not stronger</u> than BallContainer's. Thus Box <u>is not substitutable</u> 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.

```
BallContainer.add()
E = if b is_element BallContainer_pre
            return false
      else
         BallContainer_post = BallContainer_pre U b
```

```
Box.add()
E = if b is_element BallContainer_pre
            return false
    else
          if Box.volume_pre >= max_volume
             return false
          else
             Box_post = 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 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: none

(This spec is so strong, it is useless)