Specifications and Exceptions

Spring 2012
Today

Specifications

➢ What is a specification

➢ Why specifications

➢ Judging a specification
  • Strength
  • Declarative vs Operational
  • je ne sais quoi

Exceptions
What is a specification

A **contract** between the **user** of a component and its **creators**

- Grants the user **rights** and **responsibilities**

**Rights**

- What can the user expect when using a component
  - what will the function return?
  - will the method modify the state of an object?
  - can calling a function block the execution?

**Responsibilities**

- What can a component expect from the user
  - does the component expect methods to be called in a particular sequence?
  - does it expect parameters to satisfy a particular property?
  - does it have any expectations about the state of the program?

**Can exist at multiple levels of abstraction**

- Functions, Objects, Packages
Example 1:

```java
public static int find(int x, int[] a) {
...
}
```

finds the value x in array a

does it contain x?

Is there a complexity guarantee?

what if a contains multiple copies of x?

will the function allocate new objects?

is the function deterministic?

what does it mean to find?

what if the array is empty?
public static int find(int x, int[] a) {
    a[0] = x;
    return 0;
}

for (int i = 0; i < a.length; ++i) {
    if (a[i] == x) {
        return i;
    }
}

return a.length;

public static int find(int x, int[] a) {
    for (int i = a.length-1; i >= 0; --i) {
        if (a[i] == x) {
            return i;
        }
    }

    return -1;
}

public static int findC(int x, int[] a) {
    Random rnd = new Random();
    int t=0;
    while(a[t] != x){
        t = rnd.nextInt(a.length);
    }

    return t;
}
Example 1:

finds the value x in array a

Returns an index r such that a[r] == x or r=-1 if none exists

Returns the first index r such that a[r] == x or r=-1 if none exists

for(i=0; i<a.length; ++i){ if(a[i]==x){ return i; } return -1;

public static int find(int x, int[] a){
  ...
}

a[r] = x ∧ ∀ y. 0 ≤ y < r ⇒ a[y] ≠ x

return r s.t.

r = −1 ∧ ∀ y. 0 ≤ y < a.length ⇒ a[y] ≠ x
Elements of a specification

**Preconditions (requires)**
- These are your obligations as the user of the function
- Describe what the function expects from the environment
- If the preconditions don’t hold, all bets are off

**Postconditions (effects)**
- These are part of your rights as a caller
- Describe the result of calling this function

**Frame conditions (modifies)**
- These are really a special kind of post-condition
- Bound the changes that the function may produce in the environment
- In general, the function will describe what it will change, and the assumption is that everything else remains unchanged
Example 2

```
public static int binsearch(int x, int[] a){
    int h = a.length;
    int l = 0;
    while(h!=l){
        t = (h+l)/2;
        if(a[t]>x){
            h=t;
        }
        if(a[t]<x){
            l=t;
        }
        if(a[t]==x){
            return t;
        }
    }
    return 999;  //Joe Programmer: This should never happen!!
}

requires:  a must be a sorted array containing the value x
effects:    returns result such that a[result] = x
modifies:  nothing
```
Example 3

static boolean addAll (List l1, List l2)

requires: l1 != l2, l1!=null, l2!=null
modifies: l1
effects: adds the elements of l2 to the end of l1, and returns true if l1 changed as a result of call
Additional elements

**Resource requirements**
- Will the function open files, allocate memory, request special privileges?

**Complexity guarantees**
- e.g. “Table insertion is guaranteed to be logarithmic in the size of the table”

**Timing guarantees**
- Important for real time systems
- You won’t see these in standard Java

**Consistency guarantees**
- What happens if many threads call this function at the same time?
Why Specifications

Some good reasons to have specifications

- Assign blame
- Preserve freedom of implementation
- Isolate complexity
public static void replaceAll(int vBefore, int vAfter, int[] a) {
    int x = find(vBefore, a);
    while (x > 0) {
        a[x] = vAfter;
        x = find(vBefore, a);
    }
}

public static int find(int x, int[] a) {
    for (int i = 0; i < a.length; ++i) {
        if (a[i] == x) {
            return i;
        }
    }
    return a.length;
}
public static int find (int x, int[] a) {
    for (int i = 0; i < a.length; ++i) {
        if (a[i] == x) {
            return i;
        }
    }
    return a.length;
}

public static int find (int x, int[] a) {
    for (int i = a.length-1; i >= 0; --i) {
        if (a[i] == x) {
            return i;
        }
    }
    return -1;
}
Preserve freedom of implementation

**Crucial for more complex functions**

- Allows developers to upgrade algorithms and data-structures without having to modify clients

**Use with care**

- Unfortunately, just because you wrote the specification doesn’t mean your clients read it.
Common Scenario

public static void replaceAll(int vBefore, int vAfter, int[] a) {  
    int x = find(vBefore, a);  
    while(x > 0) {  
        a[x] = vAfter;  
        x = find(vBefore, a);  
    }  
}

2: Bob writes a function without reading the documentation carefully, and it happens to work

public static int find (int x, int[] a) {  
    for (int i = 0; i < a.length; ++i) {  
        if (a[i] == x) {  
            return i;  
        }  
    }  
    throw new MyException();  
}

1: Alice purposely writes the spec to give herself freedom of implementation

requires: a is an array containing the value x
effects: returns result such that a[result] = x
modifies: nothing

3: Alice changes the implementation
Isolate Complexity

Well designed interfaces hide complex implementations behind simple specifications

This process is what makes multi-million line software possible
Properties of specifications

Strength of a specification

If Specification A is stronger than Specification B

What does that mean?

➤ Any client that worked correctly under Spec B will work correctly under A

Spec A

requirements: a is an array

effects: returns the smallest value result such that a[result] = x, and -1 if x is not in a

Spec B

requirements: a is an array

effects: returns result such that a[result] = x

weaker

stronger
Potential Pitfalls

**Specification is too weak**
- Too many preconditions that are hard to satisfy
- Not sufficient guarantees about the output
- Makes life easy for implementers, but makes function impossible to use

```java
static voidaddAll(List<T> l1, List<T> l2)
effects: adds the elements of l2 to l1,
unless it encounters a null element,
at which point it throws a NullPointerException
```

**Specification too strong**
- Makes the function unimplementable
- Ex: 
  ```java
  static void open(String filename)
effects: opens a file named filename
  ```
- You can’t possibly guarantee that file opening is going to succeed!
Properties of a Specification

**Completeness**

- Does the specification describe all possible scenarios?
- When a spec is incomplete it can also be said to be under-determined

**Ex: under-determined**

- requires: a is an array containing the value x
- effects: returns result such that a[result] = x

**Ex: complete**

- requires: a is an array containing the value x
- effects: returns result such that a[result] = x
  - throws a NoSuchElement exception when a does not contain x.
Enforcing Preconditions

requires:  a is an array containing the value x  
effects:  returns result such that a[result] = x  
         throws a NoSuchElementException when a does not contain x.

There are two ways of viewing the above spec

➢ There is a pre-condition that forces a to contain the value x. When you violate it, the routine is kind enough to tell you.
➢ There is no pre-condition. When a does not contain x, the normal behavior of the routine is to throw an exception, and clients can rely on that.
public static int binsearch(int x, int[] a){
    int h = a.length;
    int l = 0;
    while(h!=l){
        t = (h+l)/2;
        if(a[t]>x){
            h=t;
        }else{
            l=t;
        }
        if(a[t]==x){
            return t;
        }
    }
    return 999; //Joe Programmer: This should never happen!!!
}
public static int binsearch(int x, int[] a) {
    assert checkSorted(a);
    int h = a.length;
    int l = 0;
    while (h != l) {
        t = (h + l) / 2;
        if (a[t] > x) {
            h = t;
        } else if (a[t] < x) {
            l = t;
        } else if (a[t] == x) {
            return t;
        }
    }
    throw new RuntimeException("Element not found!");
}

Check could be more expensive than the algorithm

requires: a must be a sorted array containing the value x
effects: returns result such that a[result] = x
modifies: nothing

Avoid silent failure!! In this case it costs you nothing
Assertions

Benefits of assertions

- Turned off by default during production runs
- Easy to turn on when testing and debugging
Assertions

Benefits of assertions
- Turned off by default during production runs
- Easy to turn on when testing and debugging
- Make checks explicit

Possible pitfalls
- Assertion bodies should not have side effects!
- Just like comments, they need to be maintained
EXCEPTIONS
Exceptions

Exceptions are abnormal return conditions from a method

- Instead of returning a value normally, the method throws an exception
- Exceptions usually indicate error conditions, but not necessarily
- Exceptions are objects. Usually just have a message, but can carry other data as well

Throwing an exception

- throw statement throws an exception object
  ```java
  throw new MalformedURLException("bad URL: " + urlString);
  ```
- throw is like return – the method immediately stops, but instead of returning a value, it propagates the exception

Catching an exception

- catch block catches exceptions of a given type
Example
Exceptions are normally **checked** at compile time – Java requires them to be either caught or declared.

But **RuntimeExceptions** are **unchecked** at compile time. You can catch or declare them, but Java doesn’t require it.
Example
Exceptions vs. Assertions

Ask yourself:

- Is the exceptional situation something that is expected to happen? (e.g. files fail to open) Is it something that would happen frequently? Or is it something that would only happen due to a programming error?

- Can the check be done cheaply, or would it change the asymptotic behavior of the function?