CITS5501 Software Testing and Quality Assurance
Input Space Partition Testing

Unit coordinator: Arran Stewart

Highlights

- How we choose values for tests
- Approaches to testing
- Model-based testing
- Input space partitioning

Input Space Partitioning

Preliminaries

Problem – choosing test values

Suppose we have some Java method we want to test:

```
/** return true if <code>elem</code> is
  * in <code>list</code>, otherwise return false.
  */
public static boolean findElement (List<Integer> list, Integer elem)
```

- What tests should we write?

Problem – choosing test values

Suppose we have some Java method we want to test:

```
/** return true if <code>elem</code> is
  * in <code>list</code>, otherwise return false.
  */
public static boolean findElement (List<Integer> list, Integer elem)
```

- What tests should we write?
- When can we stop writing tests?

Function-like things

The technique of *Input Space Partitioning* (ISP) helps us answer these questions.

If something we are trying to test can be modelled as a mathematical *function*, then we can apply the ISP technique to devise tests for it (and to check how thorough our testing currently is).

Before we use it, we'll go through a few mathematical preliminaries.

Functions – abs

In mathematics, a *function* is a mapping from one set (called the *domain*) to another (called the *codomain*).

For example, the function *abs* gives the absolute value of an integer.[1] Its *domain* is the integers, and its *codomain* is the non-negative integers.

$$abs(n) = \begin{cases} n & \text{when } n \geq 0 \\ -n & \text{when } n < 0 \end{cases}$$

[1] Actually, we could think of *abs* as being a *family* of functions. Another member of the family maps from real numbers to non-negative reals.

java.lang.Math.abs

$$abs(n) = \begin{cases} n & \text{when } n \geq 0 \\ -n & \text{when } n < 0 \end{cases}$$

Java has a method `java.lang.Math.abs` with signature

```
int abs(int a);
```

which is intended to implement that mathematical function (though for a smaller range of possible inputs).

Functions

If we want to indicate the domain and codomain of a function, we do it like this:[2]

$$abs : \mathbb{Z} \to \mathbb{Z}_{\geq 0}$$

[2] Hopefully you're familiar with standard mathematical notation for sets like the integers, rational numbers and reals. $\mathbb{Z}$ is from the German word "Zahlen", meaning "numbers" (German mathematicians were responsible for formalising a great deal of modern set theory in the early 19th century).

If not, there's a short list of them here.

isLeapYear

Some Java methods can be very naturally modelled as mathematical functions. `java.lang.Math.abs` was *intended* to implement tha mathematical *abs* function, so the mathematical function is a natural model.

As another example, a Java method `isLeapYear`, with the following signature and description

```
/** Returns a Boolean, indicating whether
 * <code>year</code> is a leap year or not.
 */
static boolean isLeapYear(int year);
```

can be modelled as the mathematical function

$$isLeapYear : \mathbb{Z} \rightarrow \{true, false\}$$

Mathematics and models

Models are always simplifications – they abstract from the real world, and leave some details out.

When we abstract the Java method

```
  static boolean isLeapYear(int year)
```

as the mathematical function

$$\mathit{isLeapYear} : \mathbb{Z} \to \{\mathit{true}, \mathit{false}\}$$

we're ignoring the fact that a Java `int` can't *actually* hold every possible integer value – it's limited to the range of values from -2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31}-1$).

Mathematics and models

Java vs model

```
static boolean isLeapYear(int year)
```

$isLeapYear : \mathbb{Z} \rightarrow \{true, false\}$

But for most purposes, that model will be good enough.

How much detail we put into our models – how "true to life" they must be – will depend on what the consequences are if our software goes wrong, and how much we want to avoid those consequences.

If we are writing budgeting software in Python for our own use, then the model above is probably fine.

If we are working with numbers of very large magnitude (or perhaps are writing a compiler), then we might want to make our model more precise.

Testing functions

We know that we can't test most Java methods exhaustively (for instance, it would be impractical to test the `java.lang.Math.abs` method with every one of the 4294967296 values it can take).

So when modelling something as a function, we rely on two important principles:

- We don't have to test all the inputs to a software component, but can choose representative samples, and
- Programmers tend to make mistakes on or around the *boundaries* of things.

Equivalence classes

Instead of writing 4294967296 tests for `java.lang.Math.abs`, we might instead try to ensure that

- We have tested it with a positive `int`
- We have tested it with a negative `int`
- We have tested it with 0

Why? Because in all likelihood, `java.lang.Math.abs` will treat all positive `int`s the same – once we've tested a few positive `int`s, chances are that testing more isn't go to make any difference.

We have grouped the possible inputs into what are called *equivalence classes* – sets of values which (for some property we choose) can be treated as *equivalent*.

Boundaries

So having divided up the `int`s into positive, negative and 0, we might decide to test `java.lang.Math.abs` with, say, the numbers 32, -4059, and 0.

Knowing that when programmers make mistakes, it is often around the boundaries of things, we might also test the numbers 1 and -1 (and perhaps the numbers -2^{31} and $2^{31}-1$) – why those numbers?

Tuples

What about if our method has more than one argument?

Then we represent it as a mapping that takes in a *pair* of things. The Java method

```
/** Returns the smaller of two int values.
  */
static int min(int a, int b)
```

is modelled as the mathematical function

$$min : (\mathbb{Z}, \mathbb{Z}) \to \mathbb{Z}$$

Tuples

Java vs model

```
static int min(int a, int b)
```

$min : (\mathbb{Z}, \mathbb{Z}) \to \mathbb{Z}$

And a method that took *three* arguments would be modelled as function taking in a triple, of type $(\mathbb{Z}, \mathbb{Z}, \mathbb{Z}) \to \mathbb{Z}$

And more generally, a method that takes *n* arguments is modelled by a function mapping from an *n*-tuple.

The “this” parameter

How would we model the getValue() method in the following class?

```
class Counter {
  private int c;
  public Counter() { this.c = 0; }
  public void increment() { c += 1; }
  public int getValue() { return c; }
}
```

The “this” parameter

How would we model the `getValue()` method in the following class?

```
class Counter {
  private int c;
  public Counter() { this.c = 0; }
  public void increment() { c += 1; }
  public int getValue() { return c; }
}
```

It takes *no* arguments at all.

The “this” parameter

Can we model it as a mathematical function something like this?

$$getValue : () \to \mathbb{Z}$$

The “this” parameter

Can we model it as a mathematical function something like this?

$$\mathit{getValue} : () \to \mathbb{Z}$$

We cannot. Mathematical functions *always* return the same result for a given parameter or parameters.

The “this” parameter

Can we model it as a mathematical function something like this?

$$getValue : () \to \mathbb{Z}$$

We cannot. Mathematical functions *always* return the same result for a given parameter or parameters.

The result of `public int getValue()` depends not only on the arguments (. . . of which there are none), but also on the *state* of the object it’s being called on.

The “this” parameter

Can we model it as a mathematical function something like this?

$$\mathit{getValue} : () \to \mathbb{Z}$$

We cannot. Mathematical functions *always* return the same result for a given parameter or parameters.

The result of `public int getValue()` depends not only on the arguments (. . . of which there are none), but also on the *state* of the object it's being called on.

If we're going to model it as a mathematical function, we *also* have to include the object state.

The “this” parameter

So our model is a mapping from a `Counter` object’s *state* (an `int`) to a return value.

$$getValue : \mathbb{Z} \rightarrow \mathbb{Z}$$

In effect, we can think of the `getValue` method as taking an “invisible” extra parameter (sometimes called the “receiver” parameter), representing the state of the object.

```
public int getValue( /* invisible "this" parameter */ )
```

In Java, whenever we call a non-static method, the Java virtual machine takes care of passing an invisible “extra” argument to the method.

```
myCounter.getValue() // <— the JVM includes myCounter as
                     //     a sort of extra argument
```

Other languages

The Python runtime does the same, when we invoke a method; but when we *write* the method, we have to explicitly put in the receiver parameter (conventionally called “self”):

```
def getValue(self):
  return self.c
```

And in non-OO languages like C, there is no language support for receiver objects at all. If we want to emulate object state, we must pass it around explicitly.

Mutating state

```
class Counter {
  private int c;
  public Counter() { this.c = 0; }
  public void increment() { c += 1; }
  public int getValue() { return c; }
}
```

What about the `increment` method? How do we model that?

Mutating state

```
class Counter {
  private int c;
  public Counter() { this.c = 0; }
  public void increment() { c += 1; }
  public int getValue() { return c; }
}
```

What about the `increment` method? How do we model that?

It takes no arguments, but *also* has no return value.

Mutating object state

```
public void increment() { c += 1; }
```

We model it as a mathematical function that takes *in* the state of a `Counter` object, and returns a *new* state:

$$\mathit{increment} : \overset{\text{input state}}{\mathbb{Z}} \to \overset{\text{output state}}{\mathbb{Z}}$$

Mutating other state

What about the `System.out.println` method, which prints things to the screen – how do we model that?

Mutating other state

What about the `System.out.println` method, which prints things to the screen – how do we model that?

Well it “mutates” the state of the screen (and possibly our eyes?), so we could model it as

$$\mathit{println} : (\mathit{String}, \overset{\text{input state}}{\mathit{SetOfPossibleScreenStates}}) \to \overset{\text{output state}}{\mathit{SetOfPossibleScreenStates}}$$

Mutating other state

If we have methods that read or write to the file system or a database, we could model them as having domains and/or codomains which include sets of things like *SetOfPossibleFileSystemStates* or *SetOfPossibleDatabaseStates*.

It doesn't matter too much what we call those sets, but we have to remember they are there.

We have to remember that the return value of a Java method that *looks* from its signature like it has no parameters –

```
  public int returnMostRecentDatabaseRecordId();
```

may very well depend on other things; so when we're coming up with **test values**, we have to include those "other things" in our test values.

Further examples

Suppose we have a static method `divide`, with the signature:

```
int divide(int m, int n);
```

It returns the value of `m` divided by `n`, but with the *precondition* that `n` can't be zero.

- How would we model this as a function?

Further examples

Suppose we have a static method `divide`, with the signature:

```
int divide(int m, int n);
```

It returns the value of `m` divided by `n`, but with the *precondition* that `n` can't be zero.

- How would we model this as a function?
- Suppose instead that we specify that in most cases, `divide` returns the value of `m` divided by `n`, *unless* `n` equals zero; in that case, it throws a `DivisionByZeroException`. How does our model change?