Being Thorough

More is not automatically better – redundant test cases just waste time.

Testing correct things is also a waste of time – focus on test cases for what is likely to fail.

  - balance the consequences of missing a bug with the effort of unnecessary testing

- *don't* write test cases for simple things where you can confidently reason about the correctness of the code
  - but bugs can still creep in to simple things, and simple things may become less simple as development continues
- *do* write test cases if checking if something worked or not is easier than reasoning about the correctness of the code
  - but testing is not a perfect substitute for reasoning about correctness
- *do* test special cases and trouble spots where bugs often arise
  - black box tests covering cases which often require unique code paths

Being Thorough

```
public static boolean contains ( int[] array,
                                 int elt ) { … }
```

- include the typical case(s) – e.g.
  - middle element in a non-empty collection
- typical special cases – e.g.
  - empty collections or collections with only one element
  - end conditions – involving first or last element
  - off-the-end conditions – before the beginning or after the end
  - handling duplicate values
- typical bugs and trouble spots – e.g.
  - off-by-one in counting loops – 1 repetition, max repetitions
  - where `null` values can arise

Designing for Testing

Implementing test cases for class methods may require access to private instance variables to set up starting state or check the expected result.

- add what is needed, but try to grant as little extra access as possible
  - for testing code in the same package, use the default (no keyword) access modifier rather than `public`
  - add a getter method or constructor rather than making instance variables less `private`
  - consider returning a “safe” representation rather than granting direct access
    - e.g. `toString()` to return a string version of the contents instead of returning the array of elements
  - consider implementing the check (at least partially) within the class rather than in the tester

Reasoning About Correctness

- test cases only test specific inputs
  - how can we be sure we’ve covered all the cases?

Preconditions

- address correct usage
- constraints on the values the method (block) works with
  - method's parameters
  - instance variables in that class which are used by the method body
  - local variables used by the block
  - global variables used by the method body / block
- only include conditions which could be violated at runtime
  - e.g. for an integer parameter, a precondition could be that the value must be > 0
  - that the value of a parameter or variable must match the declared type is not a precondition – type mismatches won't compile
- where do they go?
  - about public concepts? → include in the method comments
  - about private concepts? → note in internal comments

Internal Preconditions

```
// suit is one of “spades”, “diamonds”, “hearts”,
//  “clubs”

if ( suit.equals(“spades”) ) {
  …
} else if ( suit.equals(“diamonds”) ) {
  …
} else if ( suit.equals(“hearts”) ) {
  …
} else {
  …
}
```


Internal Preconditions

```
// i % 3 must be 0, 1, or 2  i.e. i >= 0

if ( i % 3 == 0 ) {
  …
} else if ( i % 3 == 1 ) {
  …
} else {
  …
}
```

(why is this invariant useful to state?)

Postconditions

- address correct result
- constraints on the values resulting from the successful completion of the method's (block's) task
  - method's return value
  - instance variables in that class which are used by the method body
  - local variables used by the block
  - global variables used by the method body / blocks
- practical matters
  - focus only on that the method / block does what it should, not that it also doesn't do what it shouldn't
  - for runtime checks, easily verifiable postconditions are needed so may need to identify and check certain properties of a correct result rather than the result itself
- where do they go?
  - about public concepts? → include in the method comments
  - about private concepts? → note in internal comments

Postcondition

```
// pre: array contains only values >= 0
int max = -1;
for ( int i = 0 ; i < array.length ; i++ ) {
  if ( array[i] > max ) { max = array[i]; }
}
// post: max >= array[i] for 0 <= i < array.length
```


Invariants

- address correct behavior
- constraints on intermediate values within a method or class
  - e.g. loop invariant – property that is true before the loop and after each iteration (loop correctness)
  - e.g. class invariants – limits on state contained in the object, true before and after each class method (object consistency)
  - e.g. data structure constraints (structural consistency)
- primarily useful for formal correctness arguments, but can also be employed for runtime sanity checks
- choose invariants that are useful
  - they clarify expectations or explain operation e.g. pre/postconditions, constraints imposed by the problem domain
  - they reward with information e.g. a bug in complex code could cause invariant to be false
- where to write?
  - internal comments

Class Invariant

```
public class BankAccount {

  private double balance_;   // balance >= 0

  public BankAccount () {
    balance_ = 0;
  }

  public void withdraw ( double amount ) {
    balance_ -= amount;
  }

  public void deposit ( double amount ) {
    balance_ += amount;
  }
}
```


Class Invariant / Data Structure Constraint

```
public class SortedArray {
  private int[] array_;   // in increasing order
  private int size_;
  public SortedArray ( int capacity ) {
    array_ = new int[capacity];
    size_ = 0;
  }
  public void insert ( int elt ) {
    for ( int i = size_ ; i >= 0 ; i-- ) {
      if ( array_[i-1] > elt ) {
        array_[i] = array_[i-1];
      } else {
        array_[i] = elt;
        break;
      }
    }
  }
  public int getMin () { return array_[0]; }
}
```

Loop Invariant

```
// pre: array contains only values >= 0
int max = -1;
for ( int i = 0 ; i < array.length ; i++ ) {
  // before iteration: max is the largest value in
  //  array[0..i], not including array[i]
  if ( array[i] > max ) { max = array[i]; }
}
// post: max >= array[i] for 0 <= i < array.length
```

this invariant allows us to use proof by induction to establish that the loop computes the sum of the elements in the array

Invariants and Correctness

Why consider invariants?

- supports reasoning about correctness
  - showing that the invariant is maintained shows that the code isn't broken
- supports producing correct code
  - identifying preconditions is essential for correct usage of the module
  - reveals assumptions and expectations

Checking preconditions, postconditions, and other invariants at runtime aids in the detection of bugs during testing.

- (but not all postconditions and invariants can be easily checked)

Loop Invariant

```
// pre: array contains only values >= 0
int max = -1;
for ( int i = 0 ; i < array.length ; i++ ) {
  // before iteration: max is the largest value in
  //  array[0..i], not including array[i]
  if ( array[i] > max ) { max = array[i]; }
}
// post: max >= array[i] for 0 <= i < array.length
```

the pre- and postconditions can be checked, but not the loop invariant

Handling Violations

- a violated precondition, postcondition, or invariant means a bug in the code, and cannot be handled at runtime
  - program should terminate

Note –

- for public preconditions, violation is the fault of the caller, not the module
  - robustness dictates always checking
- for everything else, violation is the fault of the local code
  - only need to check if code is buggy

Public Preconditions

For public preconditions, the Java convention is to throw an `IllegalArgumentException` if the precondition is violated.

```
if ( precondition is violated ) {
  throw new IllegalArgumentException("detail message");
}
```

  - *detail message* should provide info to help with debugging

- typically checked first thing in the method
- exception is not caught
  - `RuntimeException`, so catch is not required
  - violated precondition is a bug, so solution is to correct the code – uncaught exception causes program termination

Public Preconditions

```
public class BankAccount {
  …

  /**
   * @param amount amount to withdraw (must be >= 0 and
   * cannot exceed balance)
   */
  public void withdraw ( double amount ) {
    if ( amount < 0 || amount > balance_ ) {
      throw new IllegalArgumentException("expect 0 <= "+
                        "amount <= balance; got "+amount);
    }
    balance_ -= amount;
    assert balance >= 0;
  }

  …
}
```


Assertions

Assertions let you state a boolean condition that should be true at that point in the program.

  - if it is, program execution continues normally
  - if it isn't, the program terminates

Syntax:

```
assert condition;
assert condition : error-message;
```

  - if the condition is true, nothing happens (program continues)
  - if the condition is false, an exception is generated (with the optional error message)

Using Assertions

```
// suit is one of "spades", "diamonds", "hearts",
//  "clubs"

if ( suit.equals("spades") ) {
  …
} else if ( suit.equals("diamonds") ) {
  …
} else if ( suit.equals("hearts") ) {
  …
} else {
  assert suit.equals("clubs");
  …
}
```

Using Assertions

```
// i % 3 must be 0, 1, or 2  i.e. i >= 0

if ( i % 3 == 0 ) {
  …
} else if ( i % 3 == 1 ) {
  …
} else {
  assert i % 3 == 2;
  …
}
```

- an alternative is `assert i >= 0` before the statement

Class Invariant

include only useful checks – class invariant should be true at beginning and end of each method, but with private instance variables, their values can't be changed between method calls

```
public class BankAccount {

  private double balance_;   // balance >= 0

  public BankAccount () {
    balance_ = 0;
    assert balance >= 0;
  }

  public void withdraw ( double amount ) {
    balance_ -= amount;
    assert balance >= 0;
  }

  public void deposit ( double amount ) {
    balance_ += amount;
    assert balance >= 0;
  }
}
```

identifying class invariant reveals need for precondition

Class Invariant / Data Structure Constraint

```
public class SortedArray {

  private int[] array_;   // in increasing order
  private int size_;

  public SortedArray ( int capacity ) {
    array_ = new int[capacity];
    size_ = 0;
    assert isSorted();
  }

  public void insert ( int elt ) {
    for ( int i = size_ ; i >= 0 ; i-- ) {
      if ( array_[i-1] > elt ) {
        array_[i] = array_[i-1];
      } else {
        array_[i] = elt;
        break;
      }
    }
    assert isSorted();
  }
  private boolean isSorted () { … }
}
```


Assertions

An advantage of assertions is that they can be turned on and off.

- can be left in production code without incurring a performance hit
  - checking assertion condition may be expensive
- can be turned on for testing and debugging

Note: assertions are disabled by default.

Enable for the whole program with the runtime argument

`-ea`

- in Eclipse, this is under Run Configurations “VM Arguments”
- can also selectively enable assertions for particular classes – see section 8.4.1 in the text