# Software Engineering
# Testing and Debugging — Testing

Prof. Dr. Peter Thiemann

Universität Freiburg

02.07.2009

# Recap

# Recap

- Testing – detect the presence of bugs by observing failures

# Recap

- Testing – detect the presence of bugs by observing failures
- Debugging – find the bug causing a certain failure

# Recap

- Testing – detect the presence of bugs by observing failures
- Debugging – find the bug causing a certain failure
- In order to know when a program fails we must have a specification (except for obvious failure such as a program crash)

# Recap

- Testing – detect the presence of bugs by observing failures
- Debugging – find the bug causing a certain failure
- In order to know when a program fails we must have a specification (except for obvious failure such as a program crash)
- A great deal of care is needed when writing specifications

# Recap

- Testing – detect the presence of bugs by observing failures
- Debugging – find the bug causing a certain failure
- In order to know when a program fails we must have a specification (except for obvious failure such as a program crash)
- A great deal of care is needed when writing specifications
- Finding a good set of test cases is often difficult

# Recap

- Testing – detect the presence of bugs by observing failures
- Debugging – find the bug causing a certain failure
- In order to know when a program fails we must have a specification (except for obvious failure such as a program crash)
- A great deal of care is needed when writing specifications
- Finding a good set of test cases is often difficult
- Even with good test cases, absence of failures does not imply absence of bugs (the number of ways to use the program is huge or infinite)

# Recap

- Testing – detect the presence of bugs by observing failures
- Debugging – find the bug causing a certain failure
- In order to know when a program fails we must have a specification (except for obvious failure such as a program crash)
- A great deal of care is needed when writing specifications
- Finding a good set of test cases is often difficult
- Even with good test cases, absence of failures does not imply absence of bugs (the number of ways to use the program is huge or infinite)
- Program Verification is about making sure there are no bugs

# Recap

- Testing – detect the presence of bugs by observing failures
- Debugging – find the bug causing a certain failure
- In order to know when a program fails we must have a specification (except for obvious failure such as a program crash)
- A great deal of care is needed when writing specifications
- Finding a good set of test cases is often difficult
- Even with good test cases, absence of failures does not imply absence of bugs (the number of ways to use the program is huge or infinite)
- Program Verification is about making sure there are no bugs
- This seems preferable, but software verification is in most cases economically impossible

# Recap

- Testing – detect the presence of bugs by observing failures
- Debugging – find the bug causing a certain failure
- In order to know when a program fails we must have a specification (except for obvious failure such as a program crash)
- A great deal of care is needed when writing specifications
- Finding a good set of test cases is often difficult
- Even with good test cases, absence of failures does not imply absence of bugs (the number of ways to use the program is huge or infinite)
- Program Verification is about making sure there are no bugs
- This seems preferable, but software verification is in most cases economically impossible
- On the other hand, testing is not just a way of finding bugs during the development –

# Recap

- Testing – detect the presence of bugs by observing failures
- Debugging – find the bug causing a certain failure
- In order to know when a program fails we must have a specification (except for obvious failure such as a program crash)
- A great deal of care is needed when writing specifications
- Finding a good set of test cases is often difficult
- Even with good test cases, absence of failures does not imply absence of bugs (the number of ways to use the program is huge or infinite)
- Program Verification is about making sure there are no bugs
- This seems preferable, but software verification is in most cases economically impossible
- On the other hand, testing is not just a way of finding bugs during the development –
- Making testing a part of the development makes your claims about the software more credible

# Contents of Testing part

# Contents of Testing part

- Specifications (informal)

# Contents of Testing part

- Specifications (informal)
- Test Cases

# Contents of Testing part

- Specifications (informal)
- Test Cases
  - How to write a test case

# Contents of Testing part

- Specifications (informal)
- Test Cases
  - How to write a test case
  - How to come up with a good test suite (collection of test cases)

# Specifications

- Program specifications tell what a piece of code should do
- But also what it requires to be able to do its job
- A specification can be seen as contract between the implementor and the user of the implemented code.
- A specification consists of two parts:
  - Requires (precondition) – what the user should fulfill before calling the code
  - Ensures (postcondition) – what the implementor promises about the result of the execution (provided requires were fulfilled)

# Specification Example

```
public static int find_min(int[] a) { ... }
```

# Specification Example

```
public static int find_min(int[] a) { ... }
```

**Specification**

*Requires:*

*Ensures:* *Result is the minimum element in* `a`

# Specification Example

```
public static int find_min(int[] a) { ... }
```

**Specification**

*Requires:* `a` *is non-null*

*Ensures:* *Result is the minimum element in* `a`

# Specification Example

```
public static int find_min(int[] a) { ... }
```

**Specification**

*Requires:* `a` *is non-null*

*Ensures:* *Result is a minimum element in* `a`

# Specification Example

```
public static int find_min(int[] a) { ... }
```

**Specification**

*Requires:* `a` *is non-null*

*Ensures:* *Result is equal to the minimum element in* `a`

# Specification Example

```
public static int find_min(int[] a) { ... }
```

**Specification**

*Requires:* `a` *is non-null*

*Ensures:* *Result is less than or equal to all elements in* `a`

# Specification Example

```
public static int find_min(int[] a) { ... }
```

**Specification**

*Requires:* `a` *is non-null*

*Ensures:* *Result is less than or equal to all elements in* `a` *and equal to at least one element in* `a`

# Specification Example

```
public static int find_min(int[] a) {
 int x, i;
 x = a[0];
 for (i = 1; i < a.length;i ++) {
  if (a[i] < x) x = a[i];
 }
 return x;
}
```

## Specification

*Requires:* a *is non-null*

*Ensures:* *Result is less than or equal to all elements in* a *and equal to at least one element in* a

# Specification Example

```
public static int find_min(int[] a) {
 int x, i;
 x = a[0];
 for (i = 1; i < a.length;i ++) {
  if (a[i] < x) x = a[i];
 }
 return x;
}
```

## Specification

*Requires:* `a` *is non-null and contains at least one element*

*Ensures:* *Result is less than or equal to all elements in* `a` *and equal to at least one element in* `a`

# What can be wrong about a specification?

# What can be wrong about a specification?

- Badly stated – does not make sense

**Specification**

*Requires:* `a` *is non-null*

*Ensures:* *Result is the minimum element in* `a`

# What can be wrong about a specification?

- Badly stated – does not make sense
- Vague – unclear what is meant

**Specification**

*Requires:* `a` *is non-null*

*Ensures:* *Result is a minimum element in* `a`

# What can be wrong about a specification?

- Badly stated – does not make sense
- Vague – unclear what is meant
- Imprecise – more can be said about the behaviour

# What can be wrong about a specification?

- Badly stated – does not make sense
- Vague – unclear what is meant
- Imprecise – more can be said about the behaviour
  - Postcondition is too weak

**Specification**

*Requires:* `a` *is non-null*

*Ensures:* *Result is less than or equal to all elements in* `a`

# What can be wrong about a specification?

- Badly stated – does not make sense
- Vague – unclear what is meant
- Imprecise – more can be said about the behaviour
  - Postcondition is too weak
  - Precondition is too strong

**Specification**

*Requires:* `a` *is non-null and contains at exactly one element*

*Ensures:* *Result is less than or equal to all elements in* `a` *and equal to at least one element in* `a`

# What can be wrong about a specification?

- Badly stated – does not make sense
- Vague – unclear what is meant
- Imprecise – more can be said about the behaviour
  - Postcondition is too weak
  - Precondition is too strong
- Incorrect – too much is said

# What can be wrong about a specification?

- Badly stated – does not make sense
- Vague – unclear what is meant
- Imprecise – more can be said about the behaviour
  - Postcondition is too weak
  - Precondition is too strong
- Incorrect – too much is said
  - Precondition is too weak

**Specification**

*Requires:* `a` *is non-null*

*Ensures:* *Result is less than or equal to all elements in* `a` *and equal to at least one element in* `a`

# What can be wrong about a specification?

- Badly stated – does not make sense
- Vague – unclear what is meant
- Imprecise – more can be said about the behaviour
  - Postcondition is too weak
  - Precondition is too strong
- Incorrect – too much is said
  - Precondition is too weak
  - Postcondition is too strong

**Specification**

*Requires:* `a` *is non-null and contains at least one element*

*Ensures:* *Result is less than or equal to all elements in* `a` *and equal to at least one element in* `a`, *and result is greater than 0*

# What can go wrong when writing specifications?

Are all these cases of "bad" specifications?

- It's clear that we don't invalid or incorrect specifications.
- Vague specifications is a matter of whether they can be misunderstood.
- But imprecise specifications is not such a bad thing

# Example: Strong or Weak Precondition

## Example

What does this method do?

```
public static int[] insert(int[] x, int n)
{
  int[] y = new int[x.length + 1];
  int i;
  for (i = 0; i < x.length; i++) {
    if (n >= x[i]) break;
    y[i] = x[i];
  }
  y[i] = n;
  for (; i < x.length; i++) {
    y[i+1] = x[i];
  }
  return y;
}
```

# Example, cont'd

## Example

What does this method do?

```
public static int[] insert(int[] x, int n)
{ ... }
```

## Specification

*Requires:*

*Ensures:*

# Example, cont'd

## Example

What does this method do?

```
public static int[] insert(int[] x, int n)
{ ... }
```

## Specification

*Requires:* `x` *is non-null.*

*Ensures:* *Result is equal to* `x` *with* `n` *inserted in it.*

# Example, cont'd

## Example

What does this method do?

```
public static int[] insert(int[] x, int n)
{ ... }
```

## Specification

*Requires:* `x` *is non-null.*

*Ensures:* *Result is equal to* `x` *with* `n` *inserted in it and result is sorted in ascending order.*

# Example, cont'd

## Example

What does this method do?

```
public static int[] insert(int[] x, int n)
{ ... }
```

## Specification

*Requires:* `x` *is non-null and sorted in ascending order.*

*Ensures:* *Result is equal to* `x` *with* `n` *inserted in it and result is sorted in ascending order.*

# Specification of a Class

## Class invariant

- A class invariant is a condition about the state of each class instance that should be maintain throughout its existence
- We will focus on weak invariants
  - It should hold between calls to methods of the class,
  - but not during the execution of such methods

# Specification of a Class

## Class invariant

- A class invariant is a condition about the state of each class instance that should be maintain throughout its existence
- We will focus on weak invariants
  - It should hold between calls to methods of the class,
  - but not during the execution of such methods

## Class specification consists of

- Class invariant
- Requires and ensures of the methods

# Example, class invariant

```
public class HashSet {
  private Object[] arr;
  int nobj;

  public void insert(Object o) { ... }
  ...
}
```

# Example, class invariant

```
public class HashSet {
  private Object[] arr;
  int nobj;

  public void insert(Object o) { ... }
  ...
}
```

**Class Invariant**

# Example, class invariant

```
public class HashSet {
  private Object[] arr;
  int nobj;

  public void insert(Object o) { ... }
  ...
}
```

## Class Invariant

- `nobj` should be equal to the number of non-null elements in `arr`, and

# Example, class invariant

```
public class HashSet {
  private Object[] arr;
  int nobj;

  public void insert(Object o) { ... }
  ...
}
```

**Class Invariant**

- `nobj` should be equal to the number of non-null elements in `arr`, and
- for each index $i$ in range of `arr` suc that `arr`$[i]$ is non-null, all elements between indices `arr`$[i]$.`hash`() and $i$ are non-null, and

# Example, class invariant

```
public class HashSet {
  private Object[] arr;
  int nobj;

  public void insert(Object o) { ... }
  ...
}
```

**Class Invariant**

- `nobj` should be equal to the number of non-null elements in `arr`, and
- for each index $i$ in range of `arr` suc that `arr`$[i]$ is non-null, all elements between indices `arr`$[i]$.`hash()` and $i$ are non-null, and
- there are no two non-null elements of `arr` that are equal

# Testing

# Software testing

**What we look at here**

- Systematic – general rules for how to write test cases
- Repeatable – be able to run tests over and over again

# Software testing

## What we look at here

- Systematic – general rules for how to write test cases
- Repeatable – be able to run tests over and over again

## What we don't look at here

- Running your program to see if anything goes wrong
- Letting a lot of people run your program to see if anything goes wrong (Beta-testing)

# Testing on Different Levels

- Unit Testing – testing a small unit of a system
  Requires that the behaviour of the unit has been specificied.
  In JAVA this often corresponds to testing a method or a class.

# Testing on Different Levels

- Unit Testing – testing a small unit of a system
  Requires that the behaviour of the unit has been specificied.
  In JAVA this often corresponds to testing a method or a class.
- Integration Testing – testing the interaction between two or more units

# Testing on Different Levels

- Unit Testing – testing a small unit of a system
  Requires that the behaviour of the unit has been specificied.
  In JAVA this often corresponds to testing a method or a class.
- Integration Testing – testing the interaction between two or more units
- System Testing – testing a whole system against the specification of its externally observable behaviour
  System testing is mostly useful for convincing about the correctness. Less useful for finding bugs because the infection phase going from defect to failure is usually complex and difficult to unwind.

# Testing on Different Levels

- Unit Testing – testing a small unit of a system
  Requires that the behaviour of the unit has been specificied.
  In JAVA this often corresponds to testing a method or a class.
- Integration Testing – testing the interaction between two or more units
- System Testing – testing a whole system against the specification of its externally observable behaviour
  System testing is mostly useful for convincing about the correctness. Less useful for finding bugs because the infection phase going from defect to failure is usually complex and difficult to unwind.

The code that is being tested is called the IUT (implementation under test).

# Testing Non-Functional Requirements

Not considered further

- Performance testing or load testing
- Stability testing
- Usability testing
- Security testing

# When Should Testing Take Place?

**The sooner a bug is found, the better.**

So, testing should start early.
Extreme case: Test-driven program development

# When Should Testing Take Place?

The sooner a bug is found, the better.

So, testing should start early.
Extreme case: Test-driven program development

but

Testing early means a lot of unit testing which requires a lot of specifications.

But writing specifications for the units of a system is already needed for a large project when programming by contract.

# When Should Testing Take Place?

**The sooner a bug is found, the better.**

So, testing should start early.
Extreme case: Test-driven program development

but

**Testing early means a lot of unit testing which requires a lot of specifications.**

But writing specifications for the units of a system is already needed for a large project when programming by contract.

**Tested units may be replaced later on, making the tests useless.**

On the other hand, writing and running tests often gives a deep understanding of the program. The need to replace the unit may have been realized during the testing activities.

# Systematic testing

- Use precise methods to design correct tests

# Systematic testing

- Use precise methods to design correct tests
- Each individual test is called a test case

# Systematic testing

- Use precise methods to design correct tests
- Each individual test is called a test case
- Organize collections of related test cases in test suites

# Systematic testing

- Use precise methods to design correct tests
- Each individual test is called a test case
- Organize collections of related test cases in test suites
- Use precise methods to make sure that a test suite has a good coverage of the different cases of usage

# Repeatable testing

The basic idea is to write code that performs the tests.

# Repeatable testing

The basic idea is to write code that performs the tests.

- A tool can automatically run a large collection of tests

# Repeatable testing

The basic idea is to write code that performs the tests.

- A tool can automatically run a large collection of tests
- The testing code can be integrated into the actual code, thus stored in a organised way

# Repeatable testing

The basic idea is to write code that performs the tests.

- A tool can automatically run a large collection of tests
- The testing code can be integrated into the actual code, thus stored in a organised way
- When a bug has been fixed, the tests can be rerun to check if the failure is gone

# Repeatable testing

The basic idea is to write code that performs the tests.

- A tool can automatically run a large collection of tests
- The testing code can be integrated into the actual code, thus stored in a organised way
- When a bug has been fixed, the tests can be rerun to check if the failure is gone
- Whenever the code is extended, all old test cases can be rerun to check that nothing is broken (regression testing)

# Repeatable testing

The basic idea is to write code that performs the tests.

- A tool can automatically run a large collection of tests
- The testing code can be integrated into the actual code, thus stored in a organised way
- When a bug has been fixed, the tests can be rerun to check if the failure is gone
- Whenever the code is extended, all old test cases can be rerun to check that nothing is broken (regression testing)

JUnit is a small tool for writing and running test cases. It provides:

- Some functionality that is repeatedly needed when writing test cases
- A way to annotate methods as being test cases
- A way to run test cases automatically in a batch

# Testing influences code

Apart the obvious – that testing should result in removal of bugs

# Testing influences code

Apart the obvious – that testing should result in removal of bugs

- When you write specifications and test cases for units, the responsibilities of the different parts become clearer, which promotes good OO programming style (low coupling)

# Testing influences code

Apart the obvious – that testing should result in removal of bugs

- When you write specifications and test cases for units, the responsibilities of the different parts become clearer, which promotes good OO programming style (low coupling)
- In order to be able to test programs automatically, separating the IO and functionality becomes important

# What does a test case consists of?

**Test case**

- Initialisation (of class instance and input arguments)
- Call to the method of the IUT
- A test oracle which decides if the test succeeded or failed

# What does a test case consists of?

**Test case**

- Initialisation (of class instance and input arguments)
- Call to the method of the IUT
- A test oracle which decides if the test succeeded or failed

- The test oracle is vital to run tests automatically

# Demo

Small demo showing basics of how to use JUnit

# Summary, and what's next?

## Summary

- Specifications (motivation, contracts, pre- and postconditions, what to think about)
- Testing (motivation, different kinds of testing, role in software development, junit)

## What's next?

- More examples of test cases, presenting aspects of writing test cases and features of JUnit
- How to write a good test case?
- How to construct a good collection of test cases (test suite)?