Lecture 19:
Verification, Validation and Testing

Test automation · Test coverage · Bug reporting · Alternatives to testing

INF2-SEPP

Adriana Sejfia

School of Informatics, University of Edinburgh

This Lecture

- Test automation with JUnit 5
  - Main components of a JUnit 5 test class
  - Assertion statements in JUnit 5
- Java inline assertions and their alternatives
- Test coverage and tools for it (IntelliJ IDEA)
- Bug reporting and tools for it (Trac, JIRA)
- Alternatives to testing:
  - Reviews / walkthroughs / inspections
  - Static Analysis and tools for it

Test Automation and JUnit

- Automation of tests is essential, particularly when tests must be re-run frequently.

Test Automation and JUnit

- Automation of tests is essential, particularly when tests must be re-run frequently.
- JUnit is a framework for automated testing of Java programs.

Test Automation and JUnit

- Automation of tests is essential, particularly when tests must be re-run frequently.
- JUnit is a framework for automated testing of Java programs.
- You will use JUnit 5 in Coursework 3.

Test Automation and JUnit

- Automation of tests is essential, particularly when tests must be re-run frequently.
- JUnit is a framework for automated testing of Java programs.
- You will use JUnit 5 in Coursework 3.
- Similar frameworks are now available for most programming languages.

Main Components of a JUnit 5 Test Class

- A JUnit test is a method annotated with @Test in a test class.

Main Components of a JUnit 5 Test Class

- A JUnit test is a method annotated with @Test in a test class.
- Contents: code to execute the code under test, assert method(s), usually an informative failure message.

Main Components of a JUnit 5 Test Class

Test methods can have additional annotations:

- @DisplayName("<Name>") — readable description of the test name (spaces allowed).

Main Components of a JUnit 5 Test Class

Test methods can have additional annotations:

- @DisplayName("<Name>") — readable description of the test name (spaces allowed).
- @Disabled("reason") — makes the test inactive.

Main Components of a JUnit 5 Test Class

Test methods can have additional annotations:

- @DisplayName("<Name>") — readable description of the test name (spaces allowed).
- @Disabled("reason") — makes the test inactive.
- @Tag("<TagName>") — tag the test; JUnit 5 allows running tests with a chosen tag.

Main Components of a JUnit 5 Test Class

Test methods can have additional annotations:

- @DisplayName("<Name>") — readable description of the test name (spaces allowed).
- @Disabled("reason") — makes the test inactive.
- @Tag("<TagName>") — tag the test; JUnit 5 allows running tests with a chosen tag.
- @RepeatedTest(<Number>) — repeat a test a specified number of times.

Main Components of a JUnit 5 Test Class

- @BeforeEach — executed before each test to prepare the test environment.

Main Components of a JUnit 5 Test Class

- @BeforeEach — executed before each test to prepare the test environment.
- @AfterEach — executed after each test to clean up the environment.

Main Components of a JUnit 5 Test Class

- @BeforeEach — executed before each test to prepare the test environment.
- @AfterEach — executed after each test to clean up the environment.
- @BeforeAll — executed once before all tests (e.g. connect to a database).

Main Components of a JUnit 5 Test Class

- @BeforeEach — executed before each test to prepare the test environment.
- @AfterEach — executed after each test to clean up the environment.
- @BeforeAll — executed once before all tests (e.g. connect to a database).
- @AfterAll — executed once after all tests (e.g. disconnect from a database).

Assertions in JUnit 5

- JUnit provides a library of assert methods to check output of the program under test.

Assertions in JUnit 5

- JUnit provides a library of assert methods to check output of the program under test.
- Typical use: assertEquals(expectedResult, obj.yourMethod());

Assertions in JUnit 5

- JUnit provides a library of assert methods to check output of the program under test.
- Typical use: assertEquals(expectedResult, obj.yourMethod());
- assertTrue(condition) / assertFalse(condition) — boolean checks.

Assertions in JUnit 5

- JUnit provides a library of assert methods to check output of the program under test.
- Typical use: assertEquals(expectedResult, obj.yourMethod());
- assertTrue(condition) / assertFalse(condition) — boolean checks.
- assertNull(actual) / assertNotNull(actual) — null checks.

Assertions in JUnit 5

- JUnit provides a library of assert methods to check output of the program under test.
- Typical use: assertEquals(expectedResult, obj.yourMethod());
- assertTrue(condition) / assertFalse(condition) — boolean checks.
- assertNull(actual) / assertNotNull(actual) — null checks.
- assertNotEquals(unexpected, actual) — values are not equal.

Assertions in JUnit 5

- JUnit provides a library of assert methods to check output of the program under test.
- Typical use: assertEquals(expectedResult, obj.yourMethod());
- assertTrue(condition) / assertFalse(condition) — boolean checks.
- assertNull(actual) / assertNotNull(actual) — null checks.
- assertNotEquals(unexpected, actual) — values are not equal.
- assertSame(expected, actual) / assertNotSame(unexpected, actual) — reference equality checks.

Assertions in JUnit 5

- JUnit provides a library of assert methods to check output of the program under test.
- Typical use: assertEquals(expectedResult, obj.yourMethod());
- assertTrue(condition) / assertFalse(condition) — boolean checks.
- assertNull(actual) / assertNotNull(actual) — null checks.
- assertNotEquals(unexpected, actual) — values are not equal.
- assertSame(expected, actual) / assertNotSame(unexpected, actual) — reference equality checks.
- assertArrayEquals(expected[], actual[]) — array element equality.

Assertions in JUnit 5

- JUnit provides a library of assert methods to check output of the program under test.
- Typical use: assertEquals(expectedResult, obj.yourMethod());
- assertTrue(condition) / assertFalse(condition) — boolean checks.
- assertNull(actual) / assertNotNull(actual) — null checks.
- assertNotEquals(unexpected, actual) — values are not equal.
- assertSame(expected, actual) / assertNotSame(unexpected, actual) — reference equality checks.
- assertArrayEquals(expected[], actual[]) — array element equality.
- All methods accept an optional custom error message String.

Inline Assertions

- Checks can be spread throughout the program code itself.

Inline Assertions

- Checks can be spread throughout the program code itself.
- Goals: record assumptions; sanity-check during execution; turn faults into failures (AssertionError).

Inline Assertions

- Checks can be spread throughout the program code itself.
- Goals: record assumptions; sanity-check during execution; turn faults into failures (AssertionError).

```
public static double sqrt (double x){
    assert x>=0;
   /*..*/
}
```

Inline Assertions

- Checks can be spread throughout the program code itself.
- Goals: record assumptions; sanity-check during execution; turn faults into failures (AssertionError).

```
int numWheels = 2*numBikes;
/*. . .*/
numWheels += 2;
/*. . .*/
assert numWheels%2 ==0;
```

Inline Assertions

- Checks can be spread throughout the program code itself.
- Goals: record assumptions; sanity-check during execution; turn faults into failures (AssertionError).

```
int numWheels = 2*numBikes;
/*. . .*/
numWheels += 2;
/*. . .*/
assert numWheels%2 ==0;
```

Assertions checking can be switched on/off.

Alternatives to Inline Assertions

- Overly defensive programming (returning NaN or 0 silently) hides faults — usually a bad idea.

Alternatives to Inline Assertions

- Overly defensive programming (returning NaN or 0 silently) hides faults — usually a bad idea.
- Proper exception-based error handling (e.g. throw NegativeArgumentException) is another option.

```
public static double sqrt (double x){
   if (x <0){
     throw NegativeArgumentException(”Cannot take the square
     root of a negative number!”);
   }
  /*..*/
}
```

Alternatives to Inline Assertions

- Overly defensive programming (returning NaN or 0 silently) hides faults — usually a bad idea.
- Proper exception-based error handling (e.g. throw NegativeArgumentException) is another option.
- Pro: program fails gracefully rather than crashing with AssertionError.

Alternatives to Inline Assertions

- Overly defensive programming (returning NaN or 0 silently) hides faults — usually a bad idea.
- Proper exception-based error handling (e.g. throw NegativeArgumentException) is another option.
- Pro: program fails gracefully rather than crashing with AssertionError.
- Con: checking inputs everywhere leads to duplicated code across classes.

Alternatives to Inline Assertions

- Overly defensive programming (returning NaN or 0 silently) hides faults — usually a bad idea.
- Proper exception-based error handling (e.g. throw NegativeArgumentException) is another option.
- Pro: program fails gracefully rather than crashing with AssertionError.
- Con: checking inputs everywhere leads to duplicated code across classes.
- Recommended: validate at the UI boundary; use inline assertions elsewhere to catch internal faults.

Test Coverage

- Test coverage measures the degree to which source code is executed by its tests.

Test Coverage

- Test coverage measures the degree to which source code is executed by its tests.
- Typically used during white-box testing.

Test Coverage

- Test coverage measures the degree to which source code is executed by its tests.
- Typically used during white-box testing.
- Statement coverage: % of lines executed by at least one test.

Test Coverage

- Test coverage measures the degree to which source code is executed by its tests.
- Typically used during white-box testing.
- Statement coverage: % of lines executed by at least one test.
- Branch coverage: % of branches executed by at least one test.

Test Coverage

- Test coverage measures the degree to which source code is executed by its tests.
- Typically used during white-box testing.
- Statement coverage: % of lines executed by at least one test.
- Branch coverage: % of branches executed by at least one test.
- Other types: basic condition coverage, MC/DC, path coverage …

Test Coverage

- Test coverage measures the degree to which source code is executed by its tests.
- Typically used during white-box testing.
- Statement coverage: % of lines executed by at least one test.
- Branch coverage: % of branches executed by at least one test.
- Other types: basic condition coverage, MC/DC, path coverage …
- This course focuses on statement and branch coverage.

Statement and branch coverage

```
void decideGreater(int a, int b){
  if (a>b)
    System,out.println(“a is greater than b”);
}
```

a=5; b=4

Statement and branch coverage

```
void decideGreater(int a, int b){
  if (a>b)
    System,out.println(“a is greater than b”);
}
```

a=5; b=4

- 100% statement coverage

Statement and branch coverage

```
void decideGreater(int a, int b){
  if (a>b)
    System,out.println(“a is greater than b”);
}
```

a=5; b=4

- 100% statement coverage
- 50% branch coverage

Statement and branch coverage

```
void decideGreater(int a, int b){
  if (a>b)
    System,out.println(“a is greater than b”);
}
```

a=5; b=4

- 100% statement coverage
- 50% branch coverage

a=1; b=3

Statement and branch coverage

```
void decideGreater(int a, int b){
  if (a>b)
    System,out.println(“a is greater than b”);
}
```

a=5; b=4

- 100% statement coverage
- 100% branch coverage

a=1; b=3

Statement and branch coverage

```
void decideGreater(int a, int b){
  if (a>b)
    System,out.println(“a is greater than b”);
  else
    System.out.println(“b is greater or equal to a”);
}
```

a=5; b=4

- 100% statement coverage
- 100% branch coverage

a=1; b=3

Statement and branch coverage

```
void decideGreater(int a, int b){
  if (a>b)
    System,out.println(“a is greater than b”);
  else
    System.out.println(“b is greater or equal to a”);
}
```

a=5; b=4

- 100% statement coverage
- 100% branch coverage

a=1; b=3

- Things can get complicated with nested or complex conditions

Statement vs Branch Coverage

- 100% branch coverage guarantees 100% statement coverage.

Statement vs Branch Coverage

- 100% branch coverage guarantees 100% statement coverage.
- 100% statement coverage does NOT guarantee 100% branch coverage.

Statement vs Branch Coverage

- 100% branch coverage guarantees 100% statement coverage.
- 100% statement coverage does NOT guarantee 100% branch coverage.
- High coverage does not mean the code is bug-free (not all input classes may be considered).

Statement vs Branch Coverage

- 100% branch coverage guarantees 100% statement coverage.
- 100% statement coverage does NOT guarantee 100% branch coverage.
- High coverage does not mean the code is bug-free (not all input classes may be considered).
- Easy to achieve high coverage without proper testing.

Statement vs Branch Coverage

- 100% branch coverage guarantees 100% statement coverage.
- 100% statement coverage does NOT guarantee 100% branch coverage.
- High coverage does not mean the code is bug-free (not all input classes may be considered).
- Easy to achieve high coverage without proper testing.
- May be impossible to reach 100% — dead code, unreachable branches.

Statement vs Branch Coverage

- 100% branch coverage guarantees 100% statement coverage.
- 100% statement coverage does NOT guarantee 100% branch coverage.
- High coverage does not mean the code is bug-free (not all input classes may be considered).
- Easy to achieve high coverage without proper testing.
- May be impossible to reach 100% — dead code, unreachable branches.
- Path coverage is exponentially more complex.

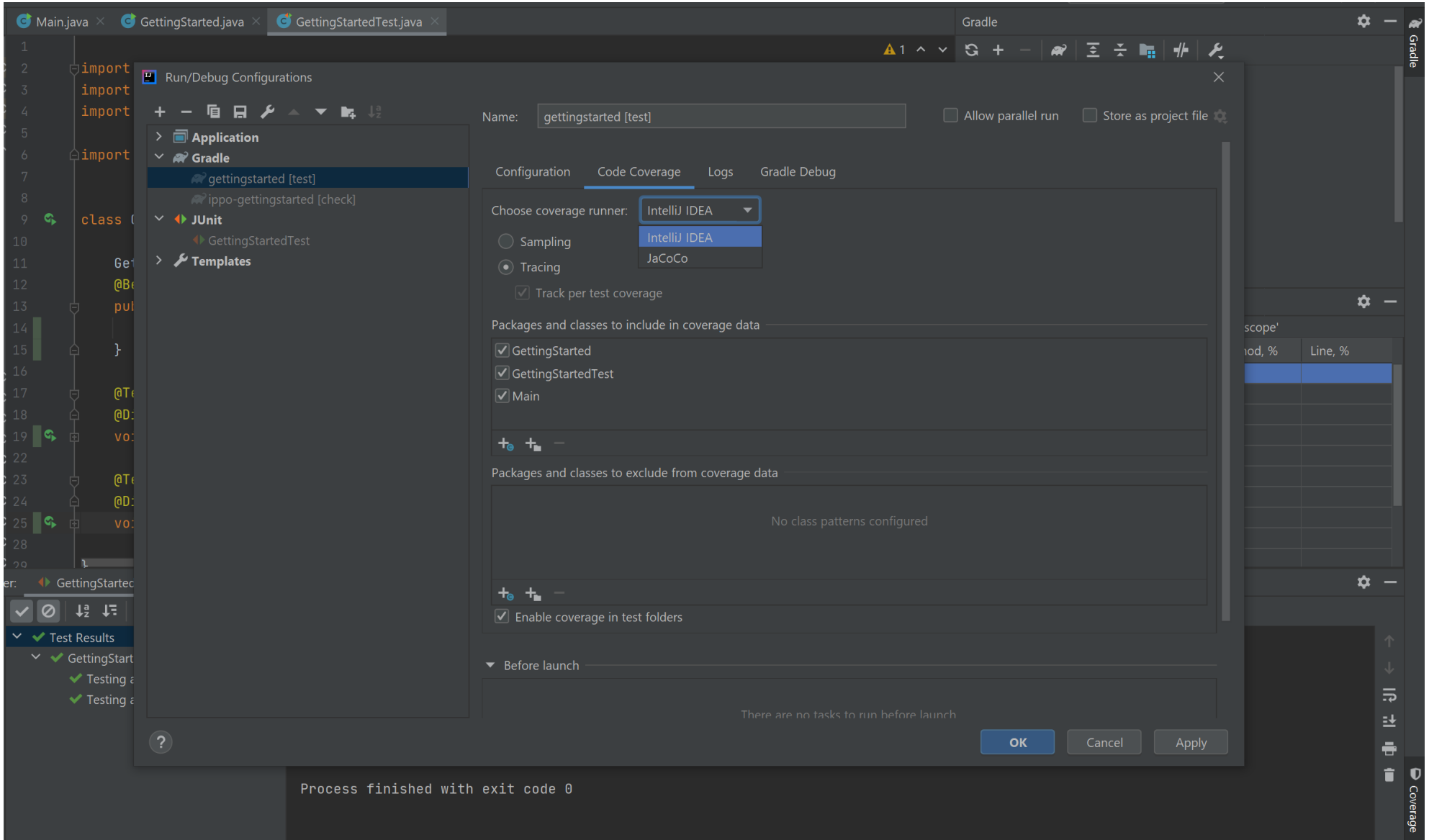
Statement vs Branch Coverage

- 100% branch coverage guarantees 100% statement coverage.
- 100% statement coverage does NOT guarantee 100% branch coverage.
- High coverage does not mean the code is bug-free (not all input classes may be considered).
- Easy to achieve high coverage without proper testing.
- May be impossible to reach 100% — dead code, unreachable branches.
- Path coverage is exponentially more complex.
- Coverage-based (white-box) testing should be combined with black-box testing.

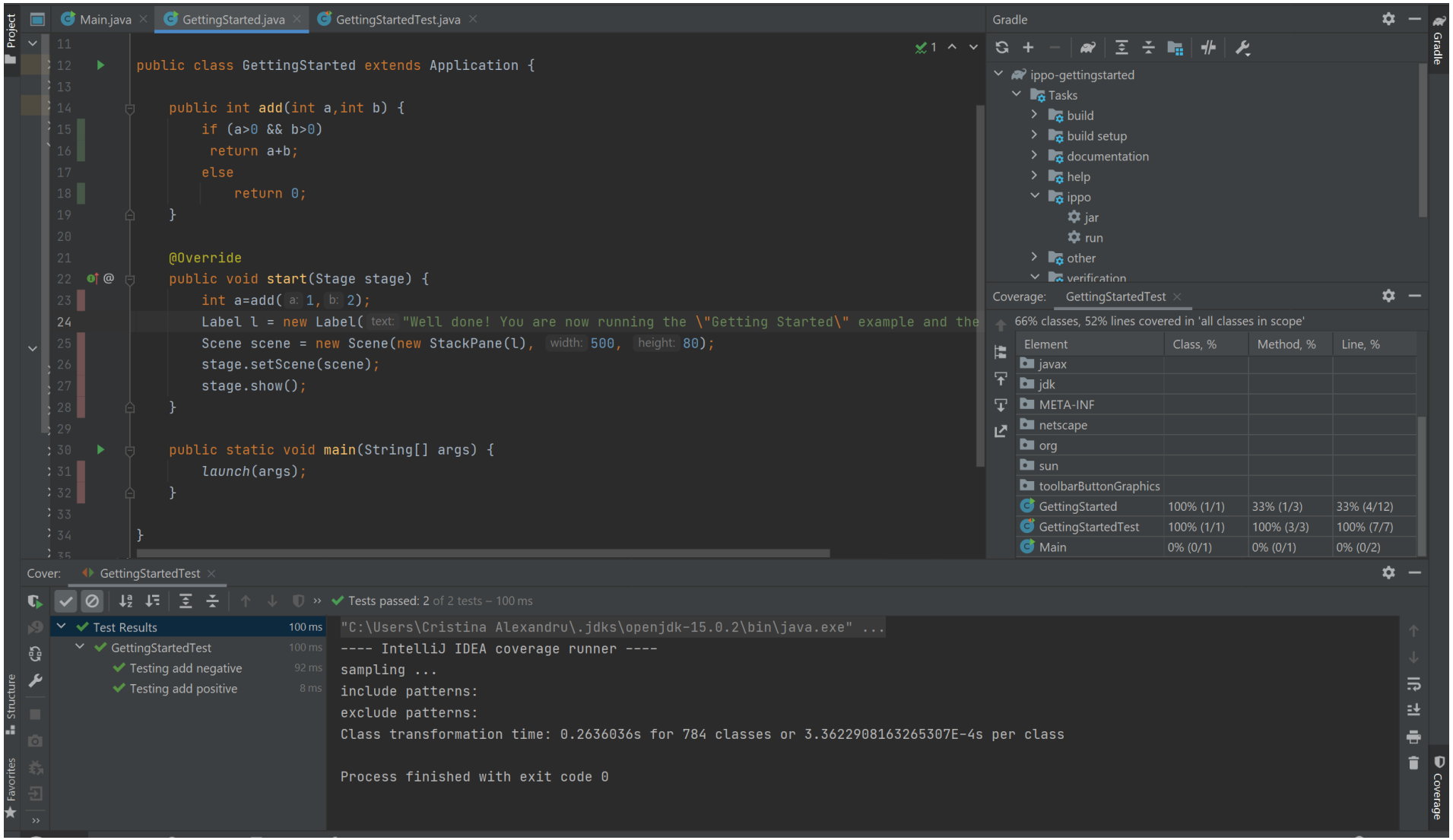
Coverage Tools — General & IntelliJ IDEA

- Open-source tools for Java/JUnit: CodeCover, EMMA, Gretel, JaCoCo, Quilt.
- Commercial tools: Atlassian Clover, Testwell.

Code Coverage in Intellij IDEA



Code Coverage in Intellij IDEA



Coverage Tools — General & IntelliJ IDEA

- Open-source tools for Java/JUnit: CodeCover, EMMA, Gretel, JaCoCo, Quilt.
- Commercial tools: Atlassian Clover, Testwell.
- IntelliJ IDEA has its own coverage runner and integrates JaCoCo and EMMA.
- Run with coverage: Edit Configurations → Code Coverage tab → choose runner (IntelliJ IDEA / JaCoCo).
- Results shown inline (green/red gutter) and in a Coverage summary panel showing Class %, Method %, Line %.

Example

Emma is working on building a library system (contains classes such as Book, BookCopy, User, LibraryMember etc). She wants to check if her code handles borrowing books well. Specifically, she wants to test the following situation: if a LibraryMember has already borrowed 12 items, they cannot borrow more, unless 2 or more of the borrowed items are journals. This functionality should only be available for a LibraryMember, not a regular user.

Bug Tracking

- Bug tracking systems manage bug reports and feature requests across a team.

Bug Tracking

- Bug tracking systems manage bug reports and feature requests across a team.
- Open-source tools: Bugzilla, Gnats, Trac, RT, MantisBT, Redmine.

Bug Tracking

- Bug tracking systems manage bug reports and feature requests across a team.
- Open-source tools: Bugzilla, Gnats, Trac, RT, MantisBT, Redmine.
- Commercial tools: JIRA, Axosoft, HP ALM/Quality Center, BugHost, IBM Rational ClearQuest.

Bug Tracking

- Bug tracking systems manage bug reports and feature requests across a team.
- Open-source tools: Bugzilla, Gnats, Trac, RT, MantisBT, Redmine.
- Commercial tools: JIRA, Axosoft, HP ALM/Quality Center, BugHost, IBM Rational ClearQuest.
- Features: receiving, tracking, notifying, monitoring, prioritising issues.

Bug Tracking

- Bug tracking systems manage bug reports and feature requests across a team.
- Open-source tools: Bugzilla, Gnats, Trac, RT, MantisBT, Redmine.
- Commercial tools: JIRA, Axosoft, HP ALM/Quality Center, BugHost, IBM Rational ClearQuest.
- Features: receiving, tracking, notifying, monitoring, prioritising issues.
- JIRA is very popular; free for teams smaller than 10.

Bug Tracking: Trac

- Open-source, web-based project management and bug tracking tool.

Bug Tracking: Trac

- Open-source, web-based project management and bug tracking tool.
- Issues (tickets) have: summary, component, version, milestone, type, owner, status, created date.

Bug Tracking: Trac

- Open-source, web-based project management and bug tracking tool.
- Issues (tickets) have: summary, component, version, milestone, type, owner, status, created date.
- Each ticket has a detail page: priority, keywords, description, workarounds, attachments.

Bug Tracking: Trac

Ticket	Summary	Component	Version	Milestone	Type	Owner	Status	Created
#3292	OOS on rejoin with new pathfinder	Core engine		Alpha 19	defect		new	Jun 13, 2015
#3471	Units not detecting invalid path.	Core engine		Alpha 19	defect		new	Sep 30, 2015
#3505	Pathfinder - Units in formation stuck frequently	Core engine		Alpha 19	defect		new	Oct 8, 2015
#3551	[PATCH] Prohibit developer overlay cheats in rated games	UI & Simulation		Alpha 19	defect		new	Oct 26, 2015
#3549	Secure authentication - prevent joins as a different player	Core engine		Alpha 20	defect		new	Oct 24, 2015
#3255	[PATCH] Prevent replay overwrites by using date and sequential ID	Core engine		Alpha 19	defect	elexis	new	May 20, 2015
#3271	OOS on rejoin - different mirage order	Core engine		Alpha 19	defect		new	May 27, 2015
#3526	Build a tower in enemy territory	UI & Simulation		Alpha 19	defect		new	Oct 14, 2015
#3545	[PATCH] Crash the game using cheats	UI & Simulation		Alpha 19	defect	stanislas69	assigned	Oct 23, 2015
#3241	[PATCH] Kick / ban players from a match	UI & Simulation		Alpha 19	enhancement	elexis	new	May 10, 2015
#1791	Units command queue is reset when they enter new formation	UI & Simulation		Alpha 20	defect		new	Dec 19, 2012
#2001	Melee units with big maximum range can attack through walls	UI & Simulation		Alpha 20	defect		new	Jun 24, 2013
#2303	Update tutorials and increase their visibility	UI & Simulation		Alpha 20	defect		new	Dec 8, 2013
#2427	[PATCH] AtlasUI does not open on on commandline Mavericks 10.9	Atlas editor		Alpha 20	defect	trompetin17	new	Feb 8, 2014

Bug Tracking: Trac

#3471 new defect

Opened 5 weeks ago
Last modified 2 days ago

Units not detecting invalid path.

Reported by:	stanislas69	Owned by:	
Priority:	Release Blocker	Milestone:	Alpha 19
Component:	Core engine	Keywords:	pathfinding
Cc:	Itms		

Description (last modified by elexis) Δ

We played a match today with elexis and ffm, and we noticed that units would often try to go from a point 'a' to a point 'b' without realising they wouldn't be able to reach it. So they just walk into the void.

The only we could workaround it, is by cancelling orders, and removing formations (setting it to none)

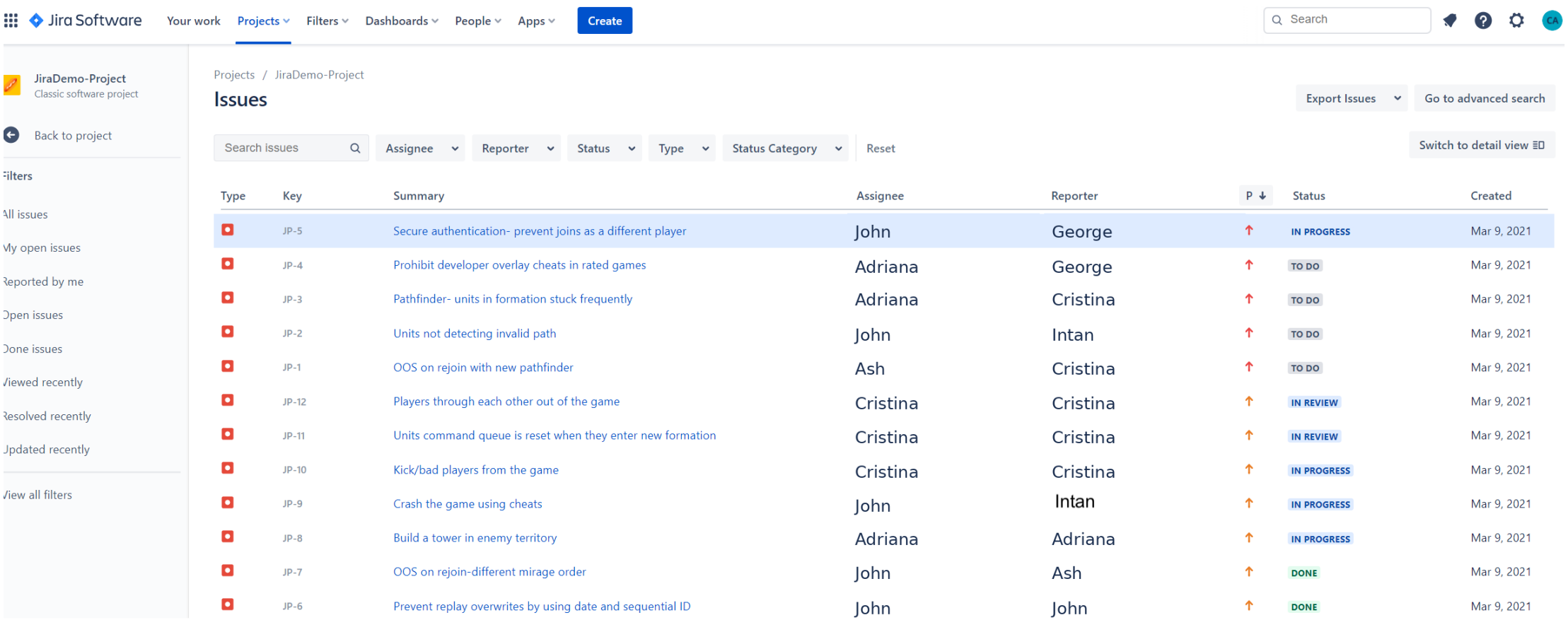
FFM stated formation should be disabled whereas I think it should be set to none by default.

▶ **Attachments** (6)

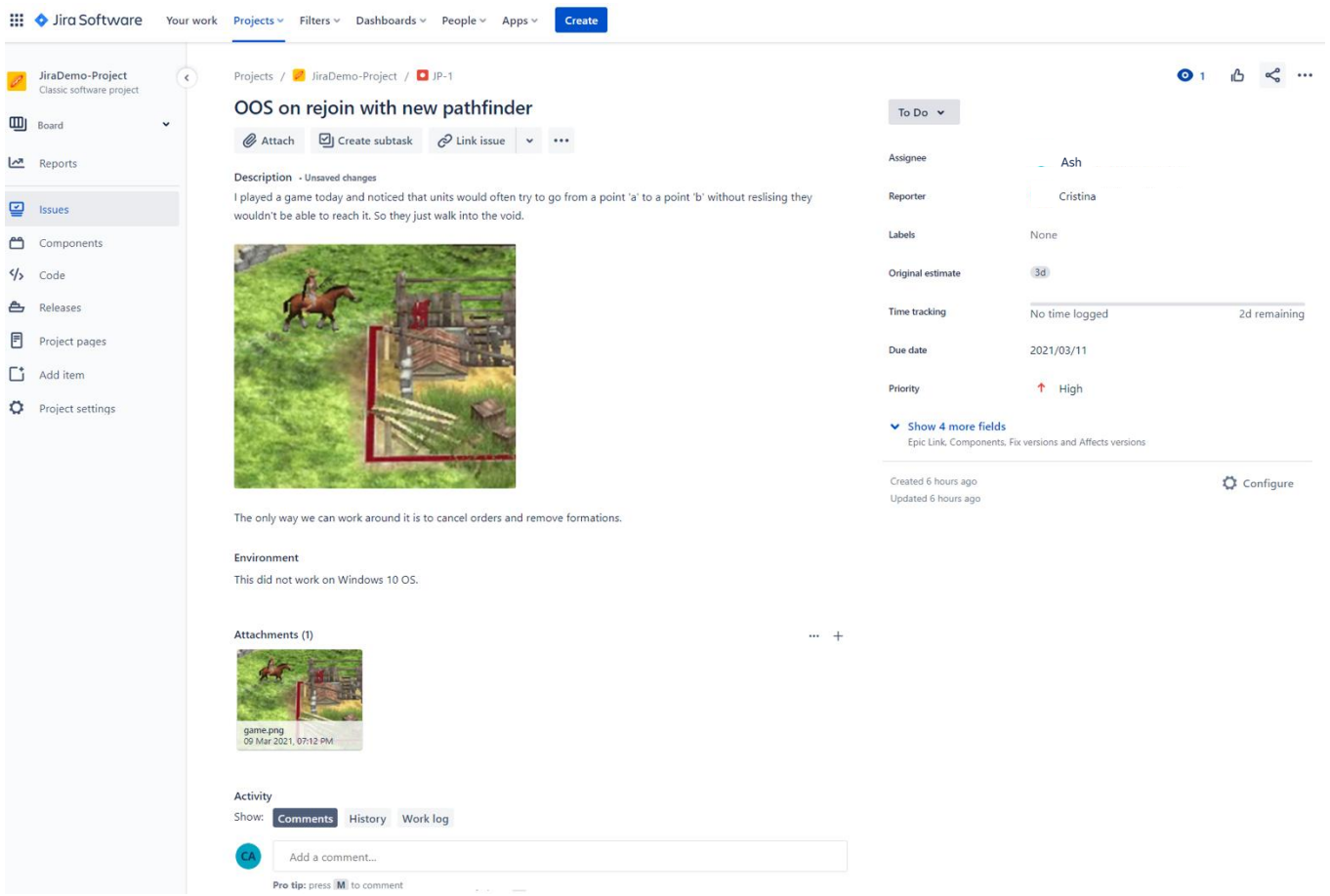
Bug Tracking: JIRA

- Widely-used commercial issue tracker by Atlassian.
- Issue list view shows: type, key, summary, assignee, reporter, priority, status, created date.
- Individual issue view: description, attachments, environment, original estimate, time tracking, due date, priority.
- Supports comments, history, work log, sub-tasks, and linked issues.

Bug Tracking: JIRA



Bug Tracking: JIRA



Alternatives to Testing: Reviews

- Complementary approach: a group of people systematically looking for problems.

Alternatives to Testing: Reviews

- Complementary approach: a group of people systematically looking for problems.
  - Can find bugs that are hard to detect through testing alone.

Alternatives to Testing: Reviews

- Complementary approach: a group of people systematically looking for problems.
  - Can find bugs that are hard to detect through testing alone.
  - Can discover when requirements have been misunderstood.

Alternatives to Testing: Reviews

- Complementary approach: a group of people systematically looking for problems.
  - Can find bugs that are hard to detect through testing alone.
  - Can discover when requirements have been misunderstood.
  - Can spot unmaintainable or overly complex code.

Alternatives to Testing: Reviews

- Complementary approach: a group of people systematically looking for problems.
  - Can find bugs that are hard to detect through testing alone.
  - Can discover when requirements have been misunderstood.
  - Can spot unmaintainable or overly complex code.
  - Works on non-executable artefacts: requirements specifications, UML models, test plans.

Alternatives to Testing: Reviews

- Complementary approach: a group of people systematically looking for problems.
  - Can find bugs that are hard to detect through testing alone.
  - Can discover when requirements have been misunderstood.
  - Can spot unmaintainable or overly complex code.
  - Works on non-executable artefacts: requirements specifications, UML models, test plans.
- Outside views complement the author's own perspective.

Alternatives to Testing: Reviews

- Complementary approach: a group of people systematically looking for problems.
  - Can find bugs that are hard to detect through testing alone.
  - Can discover when requirements have been misunderstood.
  - Can spot unmaintainable or overly complex code.
  - Works on non-executable artefacts: requirements specifications, UML models, test plans.
- Outside views complement the author's own perspective.
- Reviews, walkthroughs, and inspections are treated as equivalent in this course ('Review').

Alternatives to Testing: Static Analysis

- Automatically inspecting code to determine properties without running it — very active research area.
  - Type-checking during compilation is a basic form of static analysis.
  - Tools address runtime exception issues: null pointer exceptions, array index out of bounds.
  - Tools address correctness of pre/post-condition specifications.
  - Tools address concurrency bugs: race conditions, deadlocks.

Alternatives to Testing: Static Analysis

- Automatically inspecting code to determine properties without running it — very active research area.
  - Type-checking during compilation is a basic form of static analysis.
  - Tools address runtime exception issues: null pointer exceptions, array index out of bounds.
  - Tools address correctness of pre/post-condition specifications.
  - Tools address concurrency bugs: race conditions, deadlocks.
- More complex properties checked ⟹ smaller programs, less automation, more annotations required.

Alternatives to Testing: Static Analysis

- Automatically inspecting code to determine properties without running it — very active research area.
  - Type-checking during compilation is a basic form of static analysis.
  - Tools address runtime exception issues: null pointer exceptions, array index out of bounds.
  - Tools address correctness of pre/post-condition specifications.
  - Tools address concurrency bugs: race conditions, deadlocks.
- More complex properties checked ⟹ smaller programs, less automation, more annotations required.
- More automated / large-scale tools: may flag false positives and miss some bugs.

Alternatives to Testing: Static Analysis

- Automatically inspecting code to determine properties without running it — very active research area.
  - Type-checking during compilation is a basic form of static analysis.
  - Tools address runtime exception issues: null pointer exceptions, array index out of bounds.
  - Tools address correctness of pre/post-condition specifications.
  - Tools address concurrency bugs: race conditions, deadlocks.
- More complex properties checked ⟹ smaller programs, less automation, more annotations required.
- More automated / large-scale tools: may flag false positives and miss some bugs.
- Think of them as bug-hunting aids, not correctness guarantors.

Static Analysis Tools for Java

- SpotBugs (formerly FindBugs) — looks for bug patterns and incorrect code idioms; widely used.
- ThreadSafe (Contemplate, Informatics spin-out) — focuses on concurrency bugs.
- Infer (Meta/Facebook) — lightweight analysis scaling to 10^6+ LOC; finds null pointer and concurrency issues.
- Give SpotBugs or Infer a try!

Reading & Resources

- JUnit 5: http://www.junit.org
- JUnit Tutorial: vogella.com/articles/JUnit/article.html
- JUnit 5 Assertions API: petrikainulainen.net (JUnit 5 tutorial series)
- Writing tests in IntelliJ IDEA: jetbrains.com/help/idea/tests-in-ide.html
- Inline assertions in Java (Preconditions, Postconditions, Invariants): docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html
- Statement & branch coverage: guru99.com/code-coverage.html
- Unit testing & coverage in IntelliJ IDEA (video): youtube.com/watch?v=QDFI19lj4OM
- JIRA bug tracking: atlassian.com/software/jira/bug-tracking
- SpotBugs: spotbugs.github.io | Infer: fbinfer.com