Inf2C: Software Engineering
Lecture 19: Verification, validation and testing: Test automation, test coverage, bug reporting, alternatives to testing

Cristina Adriana Alexandru

School of Informatics
University of Edinburgh

Last lecture

Verification, validation and testing ("VV&T")

- Motivation
- Definitions
- Essence of testing
- Terminology of what can go wrong
- Kinds of tests
- How to test:
  - Test-first development
  - Test-driven development
  - Behaviour-driven development
- Evolving tests
- Limitations of testing

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
  - The IntelIJ IDEA coverage tool
- Bug reporting and tools for it
  - Bug reporting with Trac and 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.

JUnit is a framework for automated testing of Java programs. You will use JUnit 5 in Coursework 3.

Similar frameworks now available for most languages

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 message if the test fails.

The test method can also have additional annotations:

- `@DisplayName("<Name>")` for a more readable description of the test name (spaces allowed) for when it is run.
- `@Disabled("reason")` to make the test inactive for a reason.
- `@Tag("<TagName>")` to tag the test; JUnit 5 allows running tests with a certain tag.

`@RepeatedTest(<Number>)` can be used instead of `@Test` to repeat a test a number of times.

Main components of a JUnit 5 test class

There can also be other methods in a test class, annotated with:

- `@BeforeEach`: executed before each test, to prepare the test environment
- `@AfterEach`: executed after each test to clean up the environment
- `@BeforeAll`: executed before all tests in the test class to e.g. connect to a database
- `@AfterAll`: executed after all tests to e.g. disconnect from a database

Assertions

JUnit provides a library of *assert methods* to use in test code for checking output of the program being tested.

Typical use:

```
assertEquals(expectedResult, obj.yourMethod());
```

Assertions

Some alternatives to assertEquals:

assertTrue(boolean condition)	Asserts that the supplied condition is true.
assertFalse(boolean condition)	Asserts that the supplied condition is not true.
assertNull(Object actual)	Asserts that actual is null.
assertNotNull(Object actual)	Asserts that actual is not null.
assertNotEquals(Object unexpected, Object actual)	Asserts that unexpected and actual are not equal.
assertSame(Object expected, Object actual)	Asserts that expected and actual refer to the same object.
assertNotSame(Object unexpected, Object actual)	Asserts that unexpected and actual do not refer to the same object.
assertArrayEquals(type[] expected, type[] actual)	Asserts that expected and actual arrays of type type are equal.

Each of these can also get as an extra parameter a custom error message (`String`) to be displayed when the assertion fails.

Inline assertions

Checks can also be spread throughout the program code itself.

Goals: *record assumptions throughout the code; do 'sanity checks' during execution; turn faults into failures (AssertionError exception errors raised on failure)*

Examples:

- ```
  public static double sqrt (double x){
       assert x>=0;
       /*...*/
  }
  ```
- ```
  int numWheels = 2 * numBikes;
  /* ... */
  numWheels+ = 2;
  /* ... */
  assert numWheels % 2 == 0;
  ```

Such assertion checking can be switched on/off.

Alternatives to inline assertions

Don't do this: *overly defensive programming*

```
public static double sqrt (double x){
     if (x<0){
          return NAN;
     }
    /* ... */
}
```

Usually a bad idea, as it carries faults across. Better to find and avoid faults, than to hide them.

Alternatives to inline assertions

Why not always handle errors properly, by using *defensive programming with error handling*?

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

Pros: program will fail gracefully (as opposed to inline assertions).

Cons: checking inputs everywhere this way leads to duplicated code between classes. Often better to only do this at the level of the UI. In this case, inline assertions still useful to catch other faults.

Test coverage

Test coverage is a measure of the degree to which the source code of a program is executed by its tests.

Checking coverage and ensuring that tests achieve a high enough level of coverage is often done during white-box testing.

Some types of test coverage:

- Statement coverage: what percentage of the lines of the code were executed by at least one test?
- Branch coverage: what percentage of the branches of the code were executed by at least one test?
- Basic condition coverage, modified condition/decision coverage, path coverage, ...

We will only look at statement and branch coverage in this course.

Statement and branch coverage

Example:

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

One test where a = 5 and b = 4 (or any number of tests where value of a > value of b):

- We achieve 100% statement coverage because the test would exercise all lines of code.
- But do we have 100% branch coverage?
  - No, because we never execute the branch of the code in which `a > b` is false.

Statement and branch coverage

We can add tests where the value of a <= value of b.

We then find that these tests do not print the expected "b is greater or equal to a" (i.e. they fail), and we can fix the code:

```
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");
}
```

Coverage has helped unveil an unconsidered case in the code.

Things more complicated when we have combinations of (complex) nested conditions.

Statement and branch coverage

100% branch coverage guarantees 100% statement coverage.

But 100% statement doesn't guarantee 100% branch coverage!

They help realise whether we have missed any code or its branches in out tests, and may help reveal bugs, but:

- Achieving high (even 100%) coverage does not mean code is bug-free! E.g. maybe not all classes of inputs considered
- It may be easy to achieve high coverage without proper testing
- It may be impossible to achieve 100% coverage, e.g. dead code, branches that can never be reached
- Other types of coverage, e.g. path, complicate things a lot!

In general, *such testing should be combined with black-box testing*.

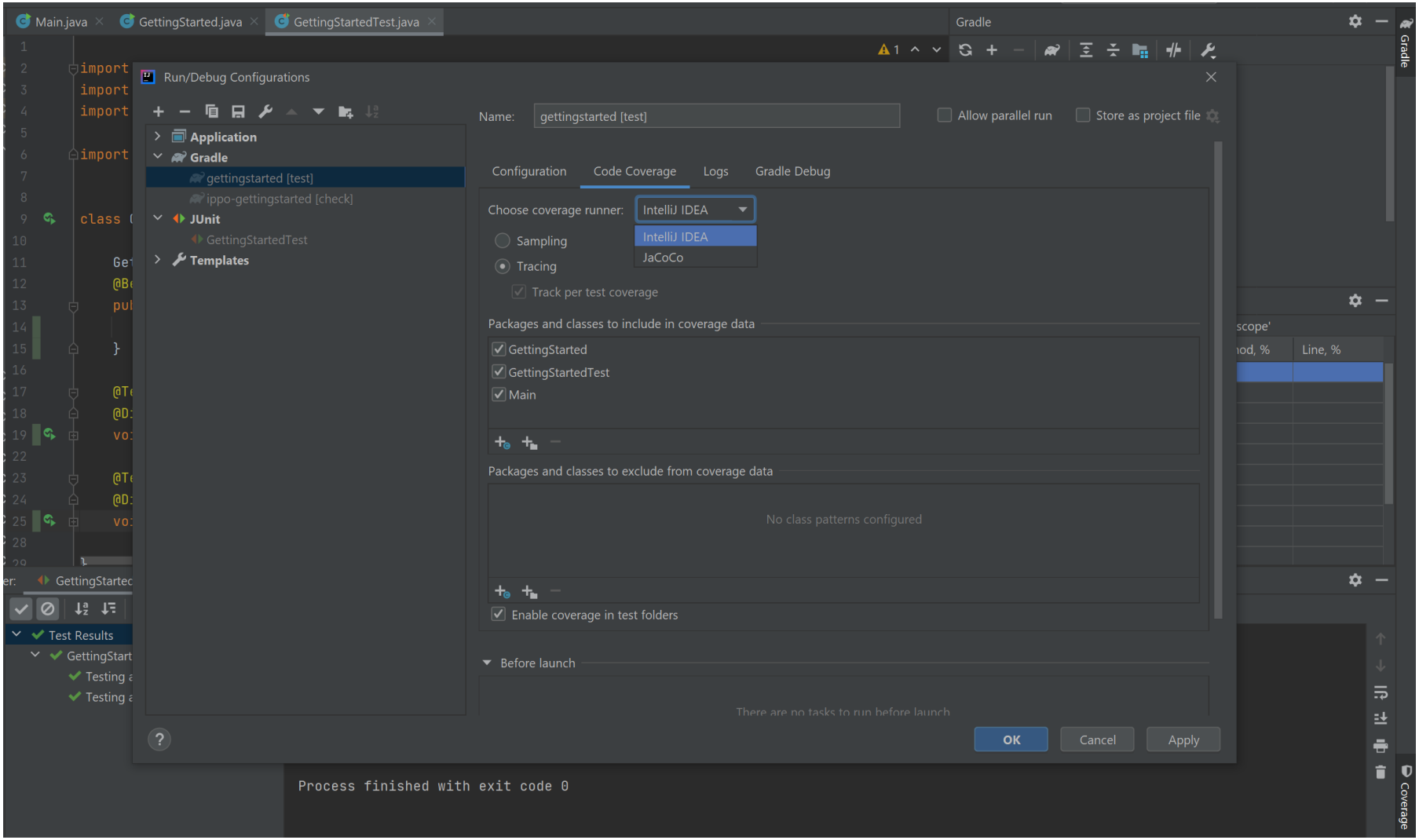
Test coverage tools in general and in IntelliJ IDEA

There are numerous:

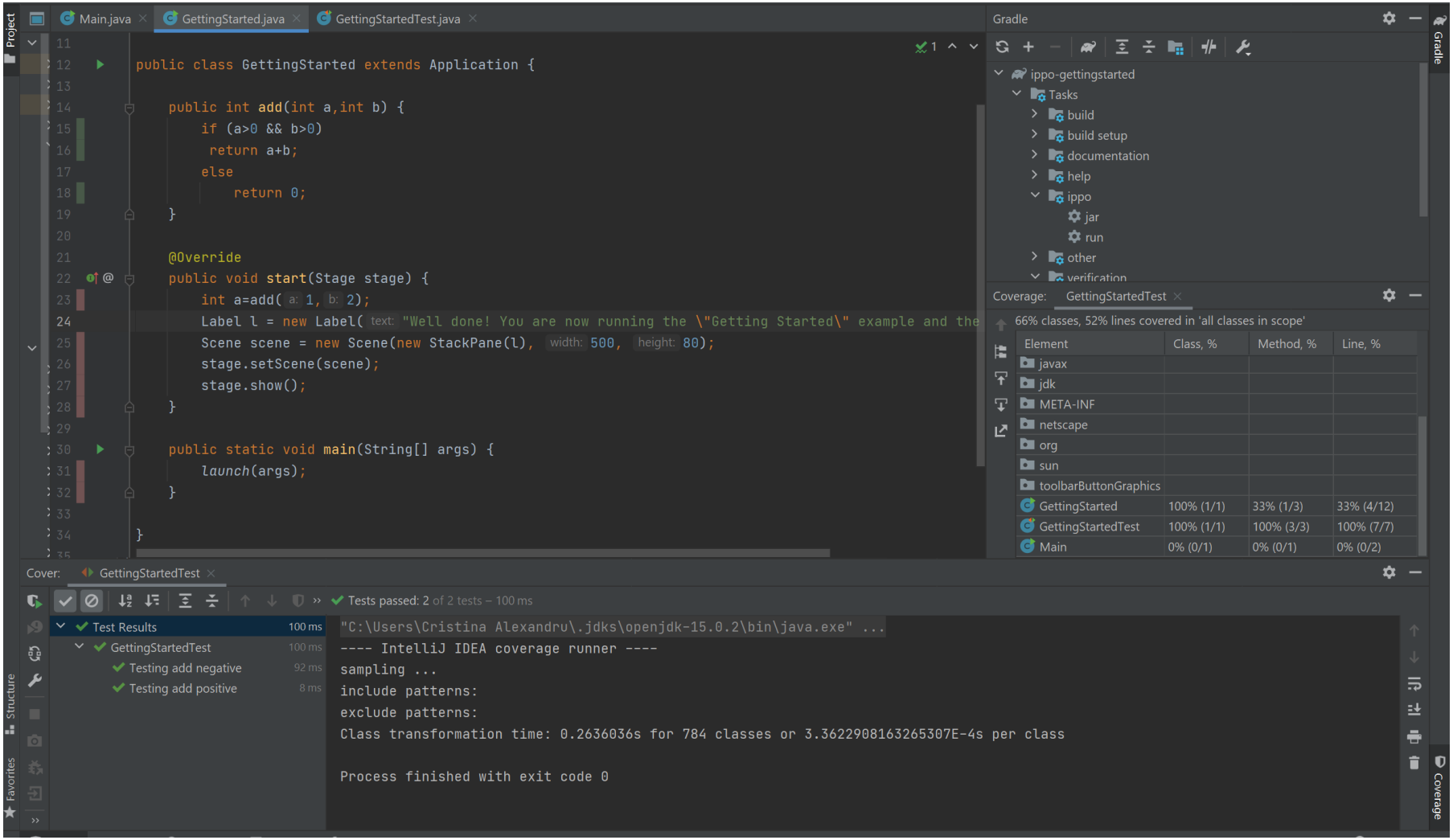
- Open-source code coverage tools for Java with JUnit: CodeCover, EMMA, Gretel, JaCoCo, Quilt.
- Commercial test coverage tools: Atlassian Clover, Testwell.

IntelliJ IDEA has its own test coverage tool, as well as the possibility to integrate with JaCoCo and EMMA.

Test coverage in IntelliJ IDEA



Test coverage in IntelliJ IDEA



Bug tracking

How does one keep track of bugs, the status of addressing them and splitting this work within a team?

Many projects use a *bug tracking system* for both bug reports and new feature requests.

Open source tools include: Bugzilla, Gnats, Trac , RT (used by our support), MantisBT, Redmine and others.

Commercial tools include: JIRA, Axosoft, HP ALM/ Quality Center, BugHost, IBM Rational ClearQuest

These provide extensive support for receiving, tracking, notifying, monitoring, etc.

We will look at Trac and JIRA (very popular, free for teams smaller than 10).

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

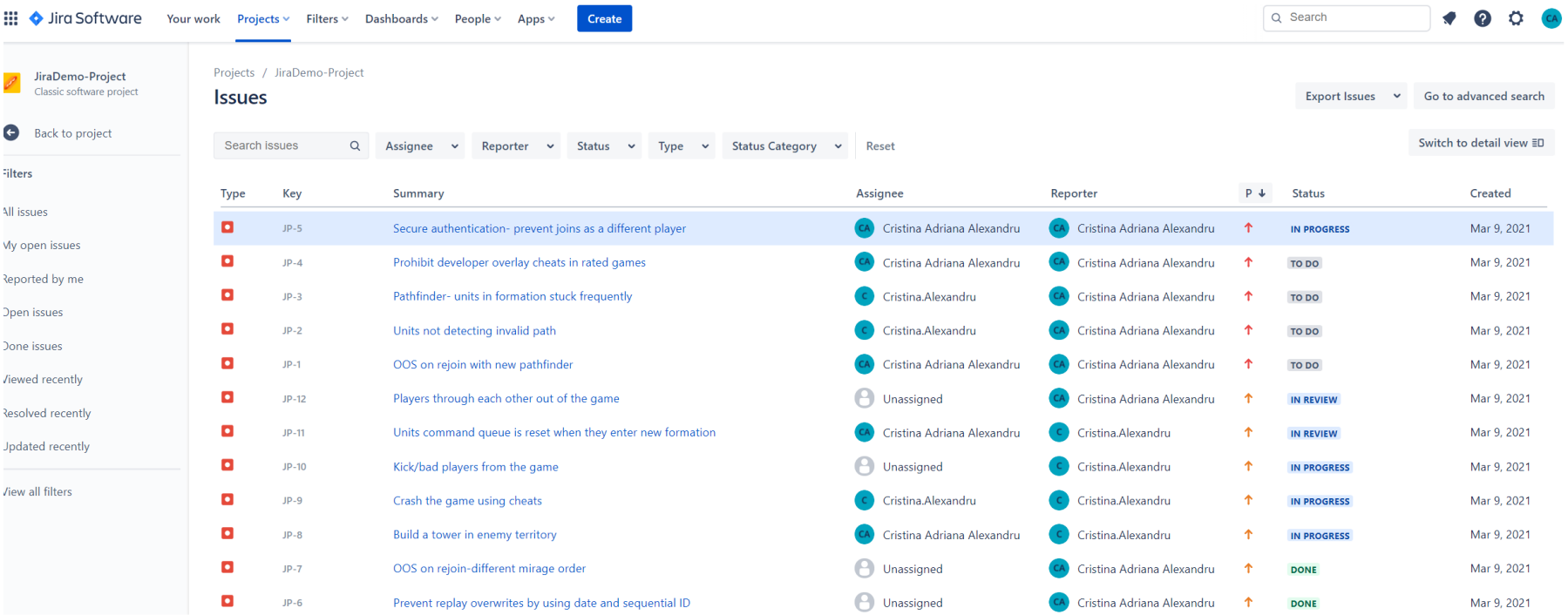
We played a match today with elexis and ffm, and we noticed that units would often try to go from a point 'a' to a point 'b' without realising they wouldn't be able to reach it. So they just walk into the void.

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



Bug tracking: Jira

Jira Software | Your work | Projects | Filters | Dashboards | People | Apps | Create

JiraDemo-Project
Classic software project

Board
Reports
Issues
Components
Code
Releases
Project pages
Add item
Project settings

Projects / JiraDemo-Project / JP-1

OOS on rejoin with new pathfinder

Attach | Create subtask | Link issue

Description • Unsaved changes

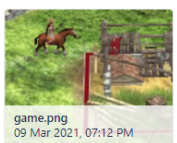
I played a game today and noticed that units would often try to go from a point 'a' to a point 'b' without reslising they wouldn't be able to reach it. So they just walk into the void.

The only way we can work around it is to cancel orders and remove formations.

Environment

This did not work on Windows 10 OS.

Attachments (1)



Activity

Show: Comments | History | Work log

CA Add a comment...

Pro tip: press M to comment

1

To Do

Assignee	CA Cristina Adriana Alexandru
Reporter	CA Cristina Adriana Alexandru
Labels	None
Original estimate	3d
Time tracking	No time logged — 2d remaining
Due date	2021/03/11
Priority	High

Show 4 more fields
Epic Link, Components, Fix versions and Affects versions

Created 6 hours ago
Updated 6 hours ago

Configure

Alternatives to testing: 1. Reviews/walkthroughs/inspections

One complementary approach is to get a group of people to look for problems.

This can:

- find bugs that are hard to find by testing,
- discover when requirements have been misunderstood,
- spot unmaintainable code,
- work on non-executable things:
  - e.g. requirements specification, UML model, test plan.

Of course the author(s) of each artefact should be looking for such problems – but it can help to have outside views too.

For our purposes reviews/walkthroughs/inspections are all the same; "Review" for short.

Alternatives to testing: 2. Static Analysis

Static analysis involves automatically (more or less) inspecting code to determine properties of it *without* running it. This is a very active area for research.

E.g. Type-checking during compilation is a basic kind of static analysis.

Tools vary in what problems they address, e.g.

- runtime exception issues (e.g. null pointer exceptions, array index out of bounds)
- correctness of pre/post-condition specification of methods
- concurrency bugs e.g. race conditions

Alternatives to testing: 2. Static Analysis- Trade-offs

When more complicated properties checked, tools generally

- can analyse only smaller programs,
- are less automated (e.g. annotations required)

As tools more automated and designed to work on larger programs, they often cannot

- guarantee every problem flagged is a real bug,
- find every bug.

Can think of such tools more as bug-hunting tools rather than tools ensuring correctness.

Static analysis tools for Java

SpotBugs (FindBugs) is relatively widely used: looks for *bug patterns*, code idioms that are often incorrect

ThreadSafe from the Informatics spinout *Contemplate* focusses on finding concurrency bugs
`http://www.contemplateltd.com/threadsafe`

Infer from Facebook applies lightweight static analysis techniques that scale to 10^6+ LOC. Finds e.g. concurrency and null pointer exception issues.

Give **SpotBugs** or **Infer** a try!

Reading

Essential: Prepare to be able to write tests in JUnit 5:

- `http://www.junit.org`
- *JUnit Tutorial* `http://www.vogella.com/articles/JUnit/article.html`
- *JUnit 5 Tutorial: Writing Assertions With JUnit 5 Assertion API* `https://www.petrikainulainen.net/programming/testing/junit-5-tutorial-writing-assertions-with-junit-5-ap`
- *Writing tests in JUnit 5*: `https://blog.jetbrains.com/idea/2020/09/writing-tests-with-junit-5/`
- *Testing in IntelliJ IDEA* and sublinks: `https://www.jetbrains.com/help/idea/tests-in-ide.html`

Reading

Essential On inline assertions in Java (focus on the Preconditions, Postconditions, and Class Invariants): `https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html`

Essential: On statement and branch coverage: `https://www.guru99.com/code-coverage.html`

Recommended: On unit testing and coverage in IntelliJ IDEA: `https://www.youtube.com/watch?v=QDFI19lj4OM`

Recommended: JIRA bug tracking: `https://www.atlassian.com/software/jira/bug-tracking`

Recommended: SpotBugs: `https://spotbugs.github.io/`

Recommended: Infer: `https://fbinfer.com/`