Code Coverage

GuessGame

Element	Missed Instructions	Cov.	Missed Branches	Cov.
makeGuess(int)		93%		83%
GuessGame(int)		100%		100%
isFinished()		100%		100%
toString()		100%		n/a
isValidGuess(int)		100%		100%
static {...}		100%		n/a
isGameBeginning()		100%		100%
hasMoreGuesses()		100%		100%
GuessGame()		100%		n/a
guessesLeft()		100%		n/a
Total	3 of 148	97%	2 of 28	92%

Model Tier

SWEN-261
Introduction to Software Engineering

Department of Software Engineering
Rochester Institute of Technology

Software Engineering
Rochester Institute of Technology

Code coverage analysis is measuring how well your unit tests exercise the production code.

- Code coverage works like this:
  1. ***Compile the project into bytecode***
  2. ***Instrument the bytecode with "touch points"***
  3. ***Run the unit tests, which gathers coverage data***
  4. ***Generate a coverage report from the gathered data***
- There are a few Java coverage tools.
  - ***Your project will use JaCoCo***
  - ***It integrates well with Maven***
- Having this information is a double-edge sword.
  - ***It's mostly a positive thing; telling the team where to spend additional testing effort.***
  - ***But don't be a slave to the metrics; we'll talk more about this later.***

JaCoCo's coverage report is a simple HTML web site that lets you drill down for more information.

- The report is stored in `/target/site/jacoco`.

guessing-game

Element	Missed Instructions	Cov.	Missed Branches	Cov.
com.example		0%		n/a
com.example.ui		83%		85%
com.example.model		90%		100%
com.example.appl		94%		100%
Total	159 of 780	79%	1 of 43	97%

com.example.model

Element	Missed Instructions	Cov.	Missed Branches	Cov.
GuessGame		97%		92%
GuessGame.GuessResult		100%		n/a
Total	3 of 197	98%	2 of 28	92%

Model Tier

It's at the class-level where you can start a meaningful analysis.

GuessGame

Element	Missed Instructions	Cov.	Missed Branches	Cov.
makeGuess(int)		93%		83%
GuessGame(int)		100%		100%
isFinished()		100%		100%
toString()		100%		n/a
isValidGuess(int)		100%		100%
static {...}		100%		n/a
isGameBeginning()		100%		100%
hasMoreGuesses()		100%		100%
GuessGame()		100%		n/a
guessesLeft()		100%		n/a
Total	3 of 148	97%	2 of 28	92%

Model Tier

```
113.  public synchronized GuessResult makeGuess(final int myGuess) {
114.    final GuessResult thisResult;
115.    // validate arguments
116.    if (myGuess < 0 || myGuess >= UPPER_BOUND) {
117.      thisResult = GuessResult.INVALID;
118.    } else {
119.      // assert that the game isn't over
120.      if (howManyGuessesLeft == 0) {
121.        throw new IllegalStateException("No more guesses allowed.");
122.      }
```

Color Legend

Green → covered
Yellow → partially covered
Red → not covered

Software Engineering
Rochester Institute of Technology

The `GuessGame` code had 97% coverage. So what do you do?

- On the one hand:
  - ***That's REALLY good already.***
  - ***The only missing test is a defensive check so maybe say "that's good enough."***
- On the other hand:
  - ***This is a core Model tier class.***
  - ***We want these to be "friendly" test dependencies.***
  - ***So maybe the team agrees to make this 100% covered.***
- What tests need to be added?

There needs to be a test to check making an invalid guess.

- Here's a test.

```
@Test
public void make_an_invalid_guess() {
  final GuessGame CuT = new GuessGame();
  assertEquals(CuT.makeGuess(TOO_SMALL), GuessResult.INVALID);
  assertFalse(CuT.isFinished(), "Game is not finished");
}
```

- Here's the updated analysis:

```
113.  public synchronized GuessResult makeGuess(final int myGuess) {
114.    final GuessResult thisResult;
115.    // validate arguments
116.    if (myGuess < 0 || myGuess >= UPPER_BOUND) {
117.      thisResult = GuessResult.INVALID;
118.    } else
```

This line is tested but only through this part of the branch.

We need to test the second part of the branch.

If we test that second branch, we should be there.

- Here's a test of a guess that is too big.

```
@Test
public void make_an_invalid_guess_too_big() {
  final GuessGame CuT = new GuessGame();
  assertEquals(CuT.makeGuess(TOO_BIG), GuessResult.INVALID);
  assertFalse(CuT.isFinished(), "Game is not finished");
}
```

- Here's the updated analysis:

```
113.  public synchronized GuessResult makeGuess(final int myGuess) {
114.    final GuessResult thisResult;
115.    // validate arguments
116.    if (myGuess < 0 || myGuess >= UPPER_BOUND) {
117.      thisResult = GuessResult.INVALID;
118.    } else {
```

- Now the Model tier is fully tested!

com.example.model

Element	Missed Instructions	Cov.	Missed Branches	Cov.
GuessGame		100%		100%
GuessGame.GuessResult		100%		n/a
Total	0 of 197	100%	0 of 28	100%

Model Tier

Deciding what level of coverage depends upon several factors…

- Some components (Model tier) are used across multiple other architectural tiers.
  - ***We recommend 95% or better for Model tier.***
- Others, like the UI Controllers, are only used by the web server.
  - ***We recommend 80% or better in all other tiers.***
- Other factors:
  - ***Team and company culture***
  - ***Application domain***
    - Regulatory requirements may specify testing requirements.
    - Those defensive checks may be safety checks. You can not know if the system is safe if you do not test the checks.

The coverage data is cumulative across all tests which may make results look better than they are.

- You want to gather coverage data from unit tests of a class not use of the class by tests of other classes.
  - ***The ultimate is to test one class at a time which is not reasonable.***
  - ***A reasonable compromise is measure code coverage for testing one tier at a time.***
- The JUnit framework and build tools allow that.
  - `@Tag("name")` ***each test file to place it into a tier category. Use*** `Model-tier`, `Application-tier`, `UI-tier`
  - ***Reset the coverage data after each tier is tested and generate the report in a separate location.***

Your project's `pom.xml` file has several test execution ids defined.

- Clean the target directory, and run all three tier-based tests
  - `mvn exec:exec@tests-and-coverage`
  - ***The reports are in*** `/target/site/jacoco/`**`tier`**`/index.html` ***where*** **`tier`** ***is*** `model`, `ui`, ***or*** `appl`.
- To run tests on a single tier
  - `mvn clean test-compile surefire:test@`**`tier`** `jacoco:report@`**`tier`** ***where*** **`tier`** ***is*** `model`, `ui`, ***or*** `appl`.
  - ***The report is in the directory listed above.***