Testing

CSC207 Fall 2016

Computer Science
UNIVERSITY OF TORONTO

Unit Testing

The goal is to fully test each *unit*.

In Java, a unit is (often) a method.

For example, your tests should call each method at least once. Further, if the behaviour of the method varies with different circumstances, then testing each circumstance is necessary.

Assertion

Single-Outcome Assertions:

```
fail;
```

Stated Outcome Assertions:

```
assertNotNull(object); OR assertNotNull(msg, object);
assertTrue(booleanEx); OR assertTrue(msg, booleanEx);
```

Equality Assertions

```
assertEqual(exp, act); OR assertEqual(msg, exp, act);
```

Floating-Point Equality Assertions

```
assertEqual(msg, expected, actual, tolerance);
```

Possible results

- **pass**: test produced the expected outcome
- **fail**: test ran but produced an incorrect outcome
- **error**: test ran but produced an incorrect behaviour (i.e., it threw an exception that you hadn’t unexpected)

Unit Testing

Unit testing follows a pattern

- Lots of small, independent tests
- Reports passes, failures, and errors
- Some optional setup and teardown shared across tests
- Aggregation (combine tests into test suites)

We could accomplish all of this “by hand”, but these design principles inspired the development of JUnit:

- When you see a pattern, build a framework
- Write shared code once
- Make it easy for people to do things the right way

Setup and Teardown

There are three steps in running a test:
setup, **run**, and **teardown**.

The **setup** phase is in a single method annotated with `@Before`

The **teardown** phase is in a single method annotated with `@After`

These run before and after every test method.

The methods annotated with `@BeforeClass` run once before all test methods in that test class are executed, and those methods annotated with `@AfterClass` run once after.

The setup and teardown methods are used to avoid repetition. For example, to create/destroy data structures required for more than one test method.

Using JUnit in Eclipse

Define the method signatures for the class to be tested.

Select the class.

Have Eclipse create JUnit tests.

Replace the dummy method bodies with real ones.

Add more test cases.

(Now, write your code.)

Selecting Test Cases

Test for success

- General cases, well-formatted input, boundary cases
- Classics:0, 1, more; odd, even; beginning, “middle”, end
- Check for data structure consistency

Test for atypical behaviour

- Does it handle invalid input (if required)?
- Does it throw the exceptions it is supposed to?

Testing Guidelines

Have one test class per class being tested.

Have at least one test method per method begin tested.

> More only if there multiple test cases.

Name your test methods `testMethodName`*`Description`*

Use annotations (e.g., `@Test, @Before, @After, ...`).

Document your test cases.

Avoid duplicate test cases.

Design for Testability

When you are writing code, think about what you need to test and how you can test it.

- Write methods that do a single task.
- Separate input, computation, and output when possible.
- Modularity, modularity, modularity.

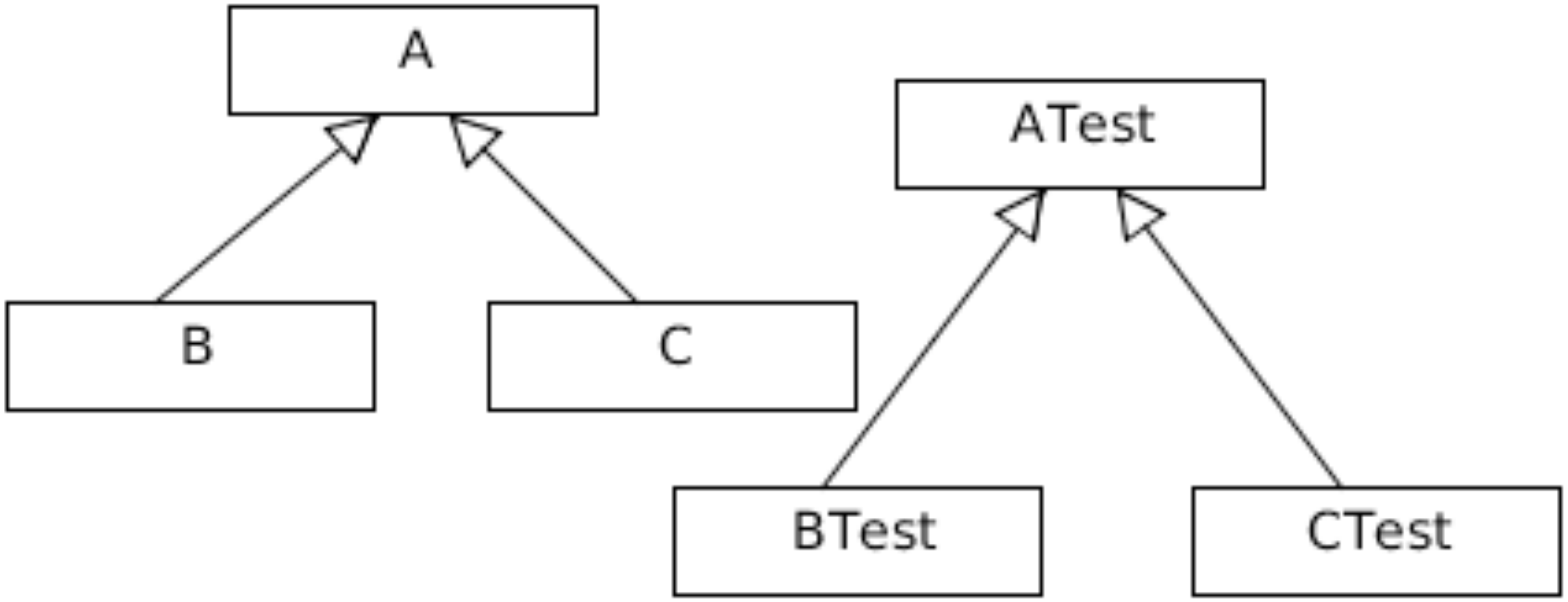
Don’t delay writing tests! Write tests *before* you write code as part of the requirements stage and update those tests *as or after* you write code.

Testing Code with Exceptions

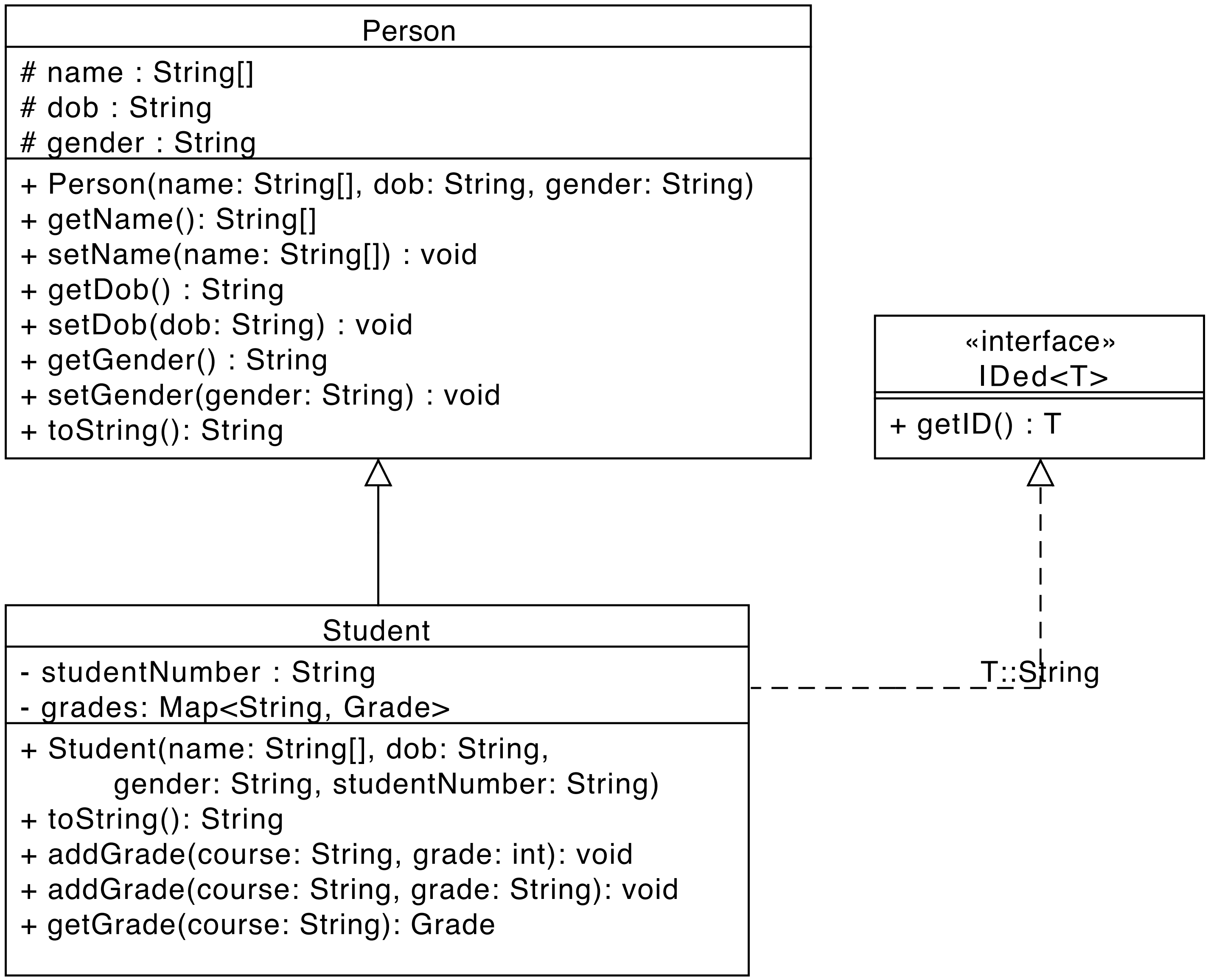
```
@Test(expected=IndexOutOfBoundsException.class)
public void testIndexOutOfBoundsException() {
    ArrayList emptyList = new ArrayList();
    Object o = emptyList.get(0);
}
```

```
@Test
public void testIndexOutOfBoundsException() {
    ArrayList emptyList = new ArrayList();
    try {
        Object o = emptyList.get(0);
        AssertFail(“IndexOutOfBoundsException not thrown: 0.”);
     } catch (IndexOutOfBoundsException e) {
     }
}
```

Testing Code with Inheritance

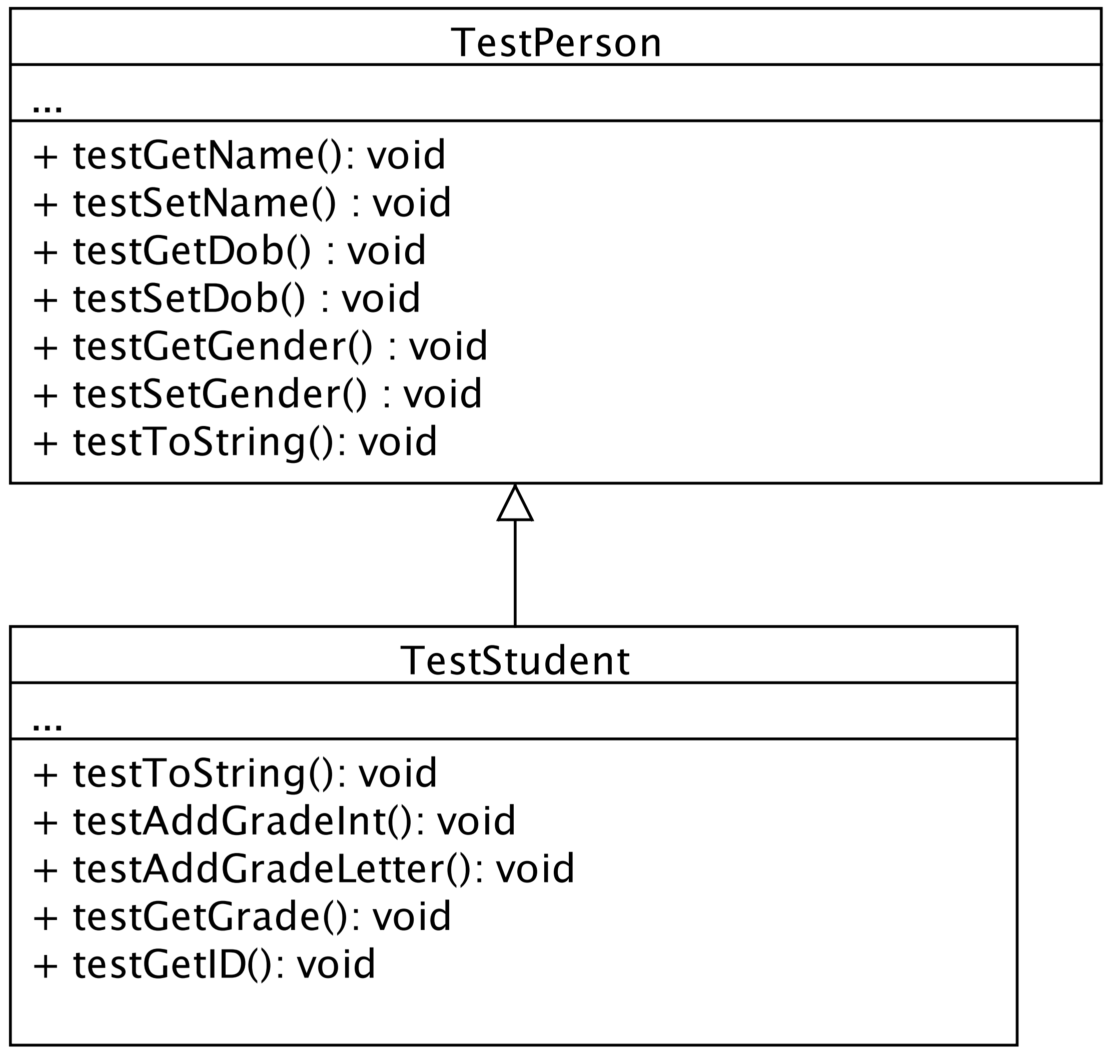


Example: inheritance



Person
# name : String[]
# dob : String
# gender : String
+ Person(name: String[], dob: String, gender: String)
+ getName(): String[]
+ setName(name: String[]) : void
+ getDob() : String
+ setDob(dob: String) : void
+ getGender() : String
+ setGender(gender: String) : void
+ toString(): String
«interface»
IDed<T>
+ getID() : T
T::String
Student
- studentNumber : String
- grades: Map<String, Grade>
+ Student(name: String[], dob: String,
gender: String, studentNumber: String)
+ toString(): String
+ addGrade(course: String, grade: int): void
+ addGrade(course: String, grade: String): void
+ getGrade(course: String): Grade

Example: testing



Test-driven Development

Write your tests first!

- In the “real world”, this is the ideal.
- The tests are based on requirements rather than code.
- The tests determine the code you need to write.
- A common programmer’s mantra: T*here is no code unless there is a test that requires it in order to pass.*

This approach aids in the definition of requirements.

It provides tangible evidence of progress.