Verification

Any activity that is undertaken to determine if the system meets its objectives or not.

- Every product should be verified (e.g., code, design documentation, user documentation).
- Every quality should be verified (e.g., behaviour, modifiability, robustness, usability).
- Some qualities or products will not yield yes/no verification results
  - Impossible/difficult to measure (e.g., correctness)
  - Subjective (e.g., modifiability)
- Implicit qualities should be verified.

Approaches to verification

1. Testing
2. Static Analysis
    - Peer review
    - Insepction/Walk-through/Structured review
    - Formal verification
3. Symbolic execution — algebraic analysis of program
4. Model checking — analysis of finite state model of system

Testing

Execute the system and observe the behaviour to determine if it is acceptable.

> *"Testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence." (E. W. Dijkstra)*

The goal of testing is to find bugs.

1. What *test cases* (input values) will be used?
2. How many test will be run?
3. How will we do the testing (testing structure)?
4. How do we know if the behaviour is correct?

Test case selection 1: Black-Box Testing

- Based on externally observable behaviour of a component.
- No reference to implementation.
- Normally divide *input domain* (possible inputs) into *equivalence classes* — sets of inputs for which the future behaviour is the same.
- Choose some test cases from each (or as many as possible) of the equivalence classes.
- Try to choose some where errors are likely (e.g., boundaries of the equivalence classes).
- Assumes that the implementation chooses the same classes.
- Number of classes may be very large.

Test case selection 2: Clear-Box Testing

- Based on examination of code.
- Choose test cases so that all parts of the code are tested.
  - Lines
  - Conditions
  - Paths
- Danger of the tester missing the same cases as the implementer.
- Line coverage is very hard.
- Path coverage is practically impossible.

Test case selection 3: Random Testing

- Randomly choose test cases according to some probability density function (usage profile).
- Typically requires more test cases to find faults.
- May find cases that were overlooked.
- Can be used to estimate reliability (likelihood of fault occurring in practice).
- Validity of reliability is very dependent on the validity of the usage profile.

How many Tests?

- *Exhaustive testing* — Try every possible input.
- Until you're confident that all bugs have been found.
- Until you stop finding bugs.
  - Track rate of fault detection (faults / hour of testing).
  - Set a threshold for acceptable rate.
- As many as you have time for.

How good is Random Testing?

Consider this simple (wrong) program to compare equal length strings:

```
bool stringcmp(string s1, string s2)
{
  bool eq = true;
  unsigned i = 0;
  while (i < s1.length()) {
    eq = (s1[i] == s2[i]);
    i++;
  }
  return(eq);
}
```

What's the probability of finding this error by testing?

Probability of finding bug

= Pr(two unequal test strings have the same last character)

= 1 - Pr(strings differ in their last character)^{n}

where n is the number of test cases.

Assume random strings from an alphabet of 100 characters.

$= 1 - \frac{99}{100}^{n}$

n	Pr(detecting error)
1	0.010
5	0.049
10	0.096

So how many test cases to be 99% sure of detecting the error?

$0.99 \geq 1 - \frac{99}{100}^{n}$

$0.99^{n} \leq 0.01 \Rightarrow n \geq 459$

Testing Structure 1: Unit Testing

- Test each ‘unit’ (class/module/package) independently.
- If the parts all work then the whole should work.

Bottom-up Test the units at the bottom of the *uses* hierarchy first.

- Requires *driver functions* to call the units.
- Tested units can be used when testing higher level units.

Top-down Test the units at the top of the uses hierarchy first.

- Requires *stub functions*.

Testing Structure 2: Integration Testing

- Test the interaction between components.
- May require driver or stubs on either side.
- Will help find places were developers didn't have the same understanding of the design. (Fix the documents, they're probably ambiguous.)

Testing Structure 3: System Testing

- Test overall system behaviour.
- Very hard to isolate bugs.
- Can only be done late in the process, so cost of fixes is high.
- Typically used for acceptance testing (customer, regulatory body).

Checking Correct Behaviour: Oracles

An *oracle* is a means of determining if the observed behaviour is correct or not.

- Most common form: human observation.
  - Time consuming
  - Expensive
  - Error prone
- Automated oracles — use a program to check.
  - Fast, cheap, accurate.
  - Must be coded somehow (can be generated from spec. if spec. is written formally).
  - Could itself have errors.
- Partial oracles — don't check all required properties.
  - Check those that are easiest to check.
  - Check those that are likely to be source of faults.

Test Driven Development

Three principles:

1. Don’t write any production code until you have written a failing test case.
2. Don’t write more of a unit test than is sufficient to fail or fail to compile.
3. Don’t write more production code than is sufficient to pass the failing test.

- Leads to very short cycles between writing tests and production code.
- Every method has tests that verify its operation.
- Forces us to view code under development from point of view of caller—helps us to get the interface right.
- Tests are valuable documentation.

Decoupling Design

- In order to test parts we need to have parts that it uses.
- Mock objects or stub methods are needed.
- Implementation is cleaner if (Java) interfaces are defined so that mock and real objects can be interchanged.
- TDD results in better decoupling in design.

Mock Objects

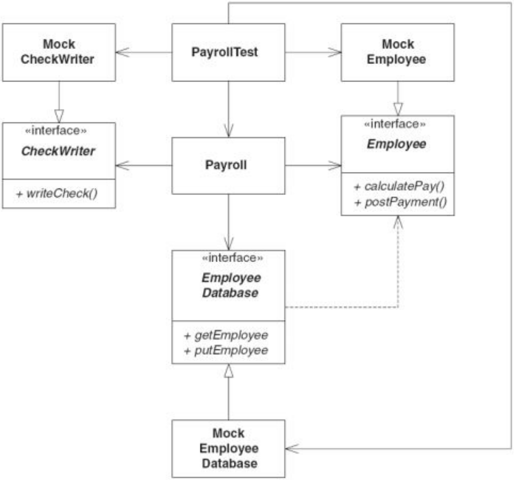


Figure 4-2
Decoupled `Payroll` using MOCK OBJECTS for testing

Acceptance Tests

- Intended to verify that the system works as a whole.
- Should be written by client in a notation that he/she can understand (i.e., not code).
- Become true documentation of feature requirements — the requirements specification for each feature.
- Should be automated.
- Developing acceptance tests early will influence architecture — decoupling to facilitate the testing.

Testing with JUnit

Running a test case:

1. Get the component to a known state (set up).
2. Cause some event (the test case).
3. Check the behaviour.
    - Record pass/fail
    - Track statistics

- Typically we want to do a lot of test cases so it makes sense to automate.
- Test cases are mostly similar in structure, so we can generalize them.

JUnit

- JUnit is a framework for writing repeatable tests.
- classes for structuring test cases.
- runners to run test cases and collect statistics.
  - `junit.textui.TestRunner` – Text based
  - `junit.swingui.TestRunner` – Swing based
  - `junit.awtui.TestRunner` – AWT based
- Normally used by extending `TestCase` with specifics for testing a particular class/system/component.

Test Fixture

- Usually some set of test cases operate on a similar set of objects — the *test fixture*.
- Fixture is implemented by member variables of extension (subclass) of `junit.framework.TestCase`.
- Override two methods:
  - `protected void setUp()` — initialize fixture prior to each test case.
  - `protected void tearDown()` — clean up fixture after each test case.

Test Cases

- By default, methods named `test`*Something* are test cases.
- Write one method `test`... for each test case.
- Use `assert`*XXX* from `junit.framework.Assert` (a parent of `TestCase`) to evaluate results.
  - `assertEquals`
  - `assertTrue`
  - `assertFalse`
  - `assertSame`
  - `assertNotSame`
  - `fail` — for when you know a test has failed.

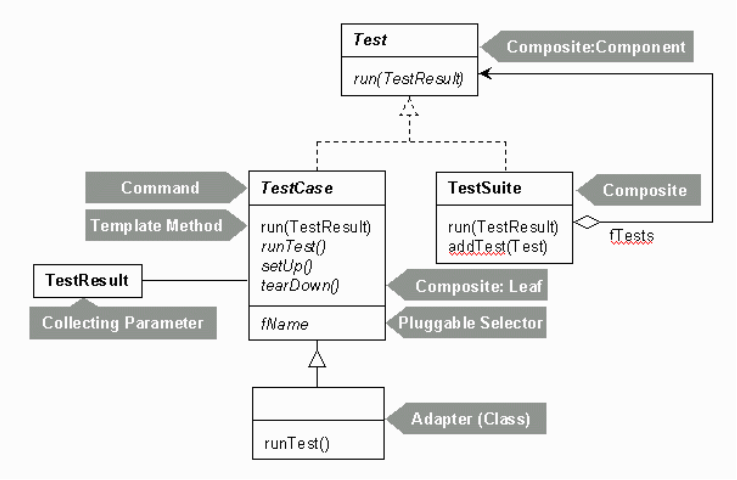
Test Suite

- To run a group of tests together, create a test suite.
- Simplest to simply implement:

```
public static Test suite() {
  return new TestSuite(YourTestClass.class);
}
```

- Will form test suite containing all methods that start with “test”.
- Can also use no-argument constructor and explicitly add tests with `addTest`.

Design of JUnit



References

There's lots of info at `http://www.junit.org`, including:

Kent Beck and Erich Gamma.
JUnit A Cook's Tour, 2004.

Kent Beck and Erich Gamma.
JUnit Cookbook, 2004.