Chapter 5.2 - Testing Tools

Walter F. Tichy

Testing Tools of the Software Amateur

- Testing with **print statements** in the program
  - print variables at specific places in the program
  - print statements will be deleted or commented out after test
  - slightly better: switch on and off via translator directives

    - In C:

```
#define TEST
.....
#ifdef TEST print(x);
.....
#ifdef TEST print(y);
.....
```

Testing Tools of the Software Amateur

- Testing with an **interactive debugger**
  - Set breakpoints—execution of a program stops at breakpoints
  - Inspect variables interactively at the breakpoints
    - Some debuggers can display objects and linked data structures (lists, trees) graphically
  - if inspection is finished, resume execution (stops at the next breakpoint)
- Disadvantage of debugger:
  - Information about breakpoints, examination is lost after debugger run
  - only the developer of the code can really work with the debugger.
  - anyone else does not know what the values of the variables should be.

Testing Tools of the Software Amateur

- Testing with **test scripts**
    - Run automatically
    - combined with `print statements`
- Common to all these methods:
    - Manual check of the outputs—not repeatable automatically
    - over time, information is lost about what the output should be, even for the developer.
    - Repetition of tests after changes not possible.

Disadvantages of the amateurish methods

- Test cases must be rerun when program changes are made
  - Manual check is tedious and impractical and often impossible
    - print statements often no longer present
    - if still present, they need to be removed for production (and inserted later again)
    - but the expected output is also unknown
    - when testing with debugger there is no information what variables should be inspected.

Other disadvantages of these methods

- Results must be checked manually
    - only the programmer knows what the outputs mean and when they are correct
    - the programmer forgets this knowledge over time.
- It is impractical to combine test cases for many components and execute them collectively
    - technically difficult (test setups vary)
    - too much output data to check.

Alternatives

1. Use *assertions*
2. Write automatically running test cases that check themselves.
3. Use analysis programs that examine software for vulnerabilities.

1. Assertions

- assertions
  - Boolean functions
    - Pre- and postconditions (e.g., that a passed parameter must satisfy, or a reference must not be Null).
    - Invariants of a data structures
  - Are executed at runtime

- In the event of an error, they respond with an exception or error message.

Assertions

- In Java (1.4 and later) and C#, assertions are built into the language.
- With C and C++ you can define macros that work like assertions
- Important: Assertions can be switched on and off.

Assertions in Java

```
AssertStatement:
  assert Expression1;
  assert Expression1 : Expression2;
```

- Semantics of `assert Expression1;`
  - If `Expression1` returns "false", exception `AssertionError is` raised.
- Semantics of `assert Expression1 : Expression2;`
  - **`Expression1: boolean`**, **`Expression2:`** not `void`
  - `Expression2` is evaluated if `Expression1` returns "false”
  - Result of `Expression2` is passed to constructor of `AssertionError`
  - Program ends with `AssertionError` containing `Expression2` and *stack trace*.

Use of assertions

- Use of assertions
  - At the beginning of a method, preconditions are checked regarding parameter values; at the end results and invariants are checked

- Examples
  - At the end of a sorting method, for example, an assertion could verify that the data to be sorted is in ascending order.
    ```
    assert isSorted();
    ```

  - To check if a tree is still balanced before and after a manipulation, one could write the function `isBalanced()` and use it in assertions:
    ```
    assert isBalanced();
    ```

Use of assertions

- Unreachable code segments can be secured with an assertion:

```
switch(color) {
  case GREEN:
       ...
       break;
  case RED:
       ...
       break;
  default:
       assert false; // unreachable
       break; // for completeness
}
```

Use of assertions

- Convention for **public** methods:
  - Check input parameters not with assertions but raise `IllegalArgumentException` if incorrect
  - **Consequence**: Client programs can respond

- Convention for **private** methods:
  - Check input parameters, postconditions and invariants of all private methods with assertions
  - Reason: these would be unexpected defects
  - (Incorrect parameters in *public* methods are not unexpected. A reaction by client should be possible in such cases)

Use of assertions

- With assertions, misunderstandings and misinterpretations by programmers can be quickly uncovered.
- In production runs, assertions are turned off for performance reasons (In Java, this can be done selectively for individual classes).
- When a failure occurs or when testing program changes, the assertions are switched back on.

Use of assertions

- Assertions are not test cases
  - Assertions are checked for *all inputs*.
  - In a test case, only *that input* is checked, whether it produces the correct answer.

- A test case provides concrete inputs and expects concrete outputs.
- Assertions check pre- and post-conditions for all inputs in general, and don´t check concrete outputs.

2. Automatically running test cases

- See chapter 0.1 on JUnit

3. Analysis programs

- Warnings and defects
  - Are displayed by the development environment. These include, for example:
    - Potentially uninitialized variables
    - Unreachable instructions
    - Unnecessary instructions
- Check programming style

Checkstyle

  - Indentation
  - JavaDoc comments
  - Method parameters not declared as `final`
- Find defects based on error patterns

SpotBugs

  - With the help of a static analysis, defects can be found based on certain patterns.

Checking programs in Eclipse - SpotBugs (1)

- SpotBugs searches for possible defects in the code with the help of so-called bug patterns.
- A *bug pattern* describes recurring relationships between found failures and the underlying defects.

- https://spotbugs.github.io/

Checking programs in Eclipse - SpotBugs (2)

- Find bad practices
  - **Class defines clone() but does not implement Cloneable.**
    This can be ok (e.g., if you want to control how subclasses clone), but check if this is intentional.
- Correctness
  - **Confusing method names**
    Method names differ only by upper/lower case. Is this a case of missed override and is the correct method called?
- Internationalization
  - **Use a locality parameter in method**
  - A string is converted to upper/lower case using the platform's default conversion. This can go wrong with international fonts.
- Code vulnerability (malicious code vulnerability)
  - **A static attribute should be visible only in the package,**
    otherwise, it could be changed externally or accidentally.

Checking programs in Eclipse - SpotBugs (3)

- correctness for multithreaded applications
  - **notify() should be replaced by notifyAll()**
    Java monitors are mostly used for multiple conditions. `notify()` activates any thread, but that could be the wrong one waiting for another condition.
- Performance
  - **toString() is executed on a string.**
    This is redundant.
- Security
  - **Empty password**
    Software creates a database with an empty password. This could mean that the database is not password protected.
- Dodgy code (questionable code)
  - Checking a variable for zero that is known to be zero.