17. Assertions
Jelle Slowack, Bart Smets, Glenn Van Loon, Tom Verheyen

Outline
• Introduction (BIT, assertion, executable assertion, why?)
• Implementation-based vs responsibility-based assertions
• Implementation
• Percolation pattern
• Deployment
• Limitations & Caveats
• Some Tools

Built-in tests
• What? code added to an application that checks the application at runtime
• Purpose?
  - Check implementation-specific assumptions (e.g., integer is not zero)
  - Check implementation-independent responsibilities (e.g., if I add money to my bankaccount, it should be on my bankaccount)

Built-in tests
• Scope
  Check relationships that must hold:
  - At method entry/exit, visible to clients of a class
  - Within scope of method, hidden to clients
  - For all methods and states of all class objects
  - Among superclasses and subclasses
The assertion is the workhorse of built-in test for object oriented code.

= boolean expression that defines conditions for correct execution

**General uses**
- Implementation-specific assumption
- Conditions that must be true at entry to a method (precondition)
- Conditions that must be true at exit from a method (postcondition)
- Conditions that must be true at all times for an object (invariant)

**Executable Assertion**

**An executable assertion has 3 parts:**

1. A predicate expression (~ if-then predicate)
2. An action (eg. Writing a message, recovery, ..)
3. Enable/disable mechanism

Assertion predicate = false
=> assertion violation => assertion action

**Why use assertions?**

- Assertions prevent errors!
  Responsibilities, assumptions and their implications are made explicit and executable
- Assertions encourage well-documented code
- Dumb errors are caught early and automatically
- Debugging asserted code is easy!
- ..
  (p811 – p813)
Implementation-based vs. Responsibility-based Assertions

Bart Smets

Outline

• Implementation-based assertions
• Responsibility-based assertions
  - Contracts
  - Strengths of conditions
  - Method Scope
  - Class Scope
  - Liskov Substitution Principle
  - Modal Class: Sequential Constraints
  - Public and Private Contracts

Implementation-based Assertions: Assumption Checking

• Programmer must make assumptions => verify with assertions
  - Necessary conditions for correct execution
  - Expectations for typical conditions
  - Properties that should ALWAYS hold

• Should not be used for
  - Input Checking
  - Exception handling

Assumption Checking Example

• Class responsible for allocating objects

    // Resolve a memory allocation error with an exception
    if (foo == NULL) throw memExhaust;

• Class using previously allocated objects, simply checking the assumption

    // Allocation assumed to be good at this point
    // -- verify the assumption
    assert (foo != NULL);
Dark Alleys, Lurking Faults and Sniffers

- Definitions
  - Lurking Fault: Bug occurring only under very weird or unlikely conditions in a certain code segment
  - Fault Sniffer: Assertion checking the implementation under any circumstances

- How to find lurking faults
  - Mutation Testing
  - Heuristic analysis
- Fault Sniffer best placed within the suspicious code segment

Responsibility-based Assertions

- Express relationships at each scope that are required for the correct execution of the program
- Easier to verify than using external test cases

The Contract Metaphor

- Design-by-contract
  - Explicit statements of rights and obligations between client and server
  - States WHAT should be done, not HOW
  - Contracts expressed by invariants, pre- and postconditions
  - Implemented using assertions

Contracts and exceptions

- Exceptions provide extra stress on the client
- Two possible approaches
  - Defensive server:
    - Check client messages against preconditions and complain if they aren't met
    - Client must be able to handle complaints
  - Cooperative server:
    - Assume client sends correct message so don't check assertions
    - Results are unpredictable with incorrect messages
Assertion Strength

- Assertions can be compared in terms of relative strength by characterizing them as strong or weak with respect to a given set of variables.
- We can only compare assertions working on the same set of variables.
- Variables that make the assertion true satisfy the assertion.

Assertion Strength (2)

- The more restrictive an assertion gets, the stronger it becomes and the less variables satisfy it.
- Some variables that fail a stronger assertion must satisfy a weaker assertion.
- All of the variables that satisfy a stronger assertion must also satisfy the weaker assertion.
- If the satisfying value sets of both assertions are identical, the assertions are considered equivalent.

Assertion Strength: Example

- \(((x > 1) \&\& (x < 42))\) is stronger than \(((x > 0) \&\& (x < 100))\).
- TRUE is the weakest possible condition.
- FALSE is the strongest possible condition.

Method Scope: Preconditions

- When?
  - Evaluated at entry to a method before any code in the method body executes.
- What?
  - Constraints on call argument values and required object state.
  - Boolean expression stating whether the client is satisfying the contract.
- Who?
  - The client sets the message arguments and is therefore responsible for meeting the preconditions.
- Preconditions can be used to catch hidden client bugs.
Method Scope: Loop Invariant

• When?
  - After initialization, after each iteration, after the final
    iteration and even if no iteration occurs
• What?
  - A loop invariant relates variables used in a loop and creates a
    boolean expression
  - The loop invariant should always be true, no matter how
    many times the loop executes
• Who?
  - The server executing the loop

Method Scope: Loop Variant

• When?
  - Evaluated after each iteration
• What?
  - A loop variant relates variables used in a loop and creates an
    integer expression
  - Each successive evaluation of the loop variant should
    produce a number smaller than the previous iteration but
    never a negative value
• Who?
  - The server executing the loop

Method Scope: Loop Example

• Find the minimum and maximum value in an array

```java
min = x[0];
max = x[0];
// Loop invariant before loop
assert (min <= max) && (x[0] <= max) && (x[0] >= min));
for (int i = 0; i < nx; ++i) {
  if (x[i] < min) min = x[i];
  if (x[i] > max) max = x[i];
  // Loop invariant in/after loop
  assert ((min <= max) && (x[0] <= max) && (x[0] >= min));
}
```

Method Scope: Postconditions

• When?
  - Evaluated when a method completes, before the result is
    returned
• What?
  - Postconditions verify if the server’s promises are met
    - Often defined as a boolean expression in terms of message
      arguments and objects returned to the client
• Who?
  - The server executing the method
• Postconditions make it easier to detect implementation
  faults and coding mistakes
Class Scope: Class Invariant

- **When?**
  - Evaluated upon instantiation, upon entry and exit from every method and just before destruction
  - The class invariant should be evaluated in parallel with every pre- and postcondition and should always hold

- **What?**
  - Common conditions across all methods of a class
  - Defines the boundaries of the domain formed by the instance variables

- **Who?**
  - The methods of the serving class

Two exceptions exist:

- Invariant methods calling other class methods which in turn call the invariant method again
- Recovery methods repairing a corrupt state or other exceptional condition should be allowed to work on invalid class states

Liskov Substitution Principle

- Functions that use pointers or references to objects of a Base class type must be able to use objects of the Derived classes as well without knowing it.

When a class hierarchy is compliant to the LSP, the following should hold:

- Preconditions of overriding methods in a Derived class must be equivalent or weaker than those of the same method in the Base class
- Postconditions of overriding methods in a Derived class must be equivalent or stronger than those of the same method in the Base class
- Invariants of subclasses must be equivalent or stronger than those of their superclass
Sequential Constraints

- The result of a method call depends on the current state the server is in.
- State = combination of a set of instance variable values defined by a State Invariant
  (e.g. stack_size > 0 or stack_size == 0)
- Accepting Condition = state in which a method can accept a certain message
  (e.g. stack_size > 0 when calling pop())
- Resulting Condition = state that should result after accepting a message in a valid state
  (e.g. stack_size = old_stack_size – 1 after pop())

State Invariant

- Properties:
  - Every class state corresponds to a different state invariant
  - State invariants are at least as strong as their class' invariant
  - A class is in exactly one state at all times
  - No two state invariants can define the same state
- State invariants greatly simplify state-based testing
- Implemented the same way as class invariants

Accepting and Resulting conditions

- Accepting Conditions are asserted at the same time as preconditions
- Resulting Conditions are asserted at the same time as postconditions

Example:

```java
class A {
    void foo() {
        assert(is_stateA() || is_stateB());
        // Save the old state
        // Do stuff...
        assert ((is_stateB() && was_stateA()) ||
                (is_stateA() && was_stateB()));
    }
}
```
Public and Private Contracts

- Public Contract = Every contract the client sees
- Never use private methods/objects in the contracts of public methods (e.g., Pop() requires nonempty stack -> add empty() method so the client can check the precondition)
- Both client and server must comply with the server's public contract
- Private methods may allow transient states that are invalid with the public contract

Conclusion

- Implementation-based assertions check assumptions about the implementation
- Responsibility-based assertions are used to express contracts between entities
- The strength of a contract can be expressed by the amount of values that satisfy it
- Contracts can be used to verify if the LSP holds for a given class hierarchy

Outline

- Implementation
  - Assertion actions
  - Executable vs. Nonexecutable assertions
  - Percolation pattern
- Considerations
  - Verification
  - Using assertions to design tests
  - Pre- and post-release considerations
  - Limitations and caveats
Implementation

Assertion actions

What action should be produced when an assertion violation occurs?

- Notification
  - Generate an error message, log to file, ...
- Continuation
  - Open debugger, terminate, ...

Executable vs nonexecutable assertions

- Executable assertions
  - Assert statement in code
  - void Account::debit(Money tx_amt)
    
    ```
    assert(balance >= tx_amt); // pre-condition
    // implementation
    assert(balance >= 0); // post-condition
    ```
Executable vs nonexecutable assertions

- Nonexecutable assertions
  - Assertion predicates in comments and documentation
  - void Account::debit(Money tx_amt)

  - Comments
    - // PURPOSE subtract tx_amt from balance
    - // REQUIRE balance must be greater than tx_amt
    - // PROMISE balance will be no less than $0

  - Verifast
    - //@ requires balance |-> ?oldb & *& tx_amt <= oldb
    - //@ ensures balance |-> oldb - tx_amt

Percolation pattern

- Intent
  - Automatic checking of superclass assertions to support DBC and Liskov Substitution Principle

- Motivation
  - Assertions cannot be inherited in most languages
  - Reveal inheritance and dynamic binding bugs

- Applicability
  - Class hierarchy with polymorphic functions that does not follow LSP is most likely buggy

- Participants
  - Protected functions that implement pre-conditions, post-conditions and invariants

- Collaboration
  - Checking begins at lowest class and up to base
  - Base class assertions should be visible to derived
  - Flatten assertions by concatenating assertions in hierarchy using and/or
Percolation pattern

- Flattened invariants
  - `<flat.inv> ::= <derived.inv>`
    // There is no base class
  - `<flat.inv> ::= <derived.inv> && <base.inv>`
    // Nontrivial invariants for both derived and base
  - `<flat.inv> ::= <derived.inv> && TRUE`
    // All base class invariants are trivial
  - `<flat.inv> ::= TRUE && <base.inv>`
    // Derived class invariant is trivial

Percolation pattern

- Flattened pre-conditions and post-conditions
  - Depends on inheritance usage
    - Define / specialize a method
    - Override a method
    - Extend a method
  - C++ example
    - Invariant, pre- and post-condition methods are protected, inline and const

Percolation pattern

- C++ example
  - Derived adds new method `bar()` (specialization)
  - Derived overrides `foo()`

Percolation pattern

- C++ example
  - Derived adds new method `bar()` (specialization)
  - Derived overrides `foo()`
Percolation pattern

• Define / specialize a method
  - Pre-conditions
    \[ <\text{flat.pre}> ::= <\text{derived.pre}> \&\& <\text{flat.inv}> \]
    // No base class pre-condition to inherit
  - Post-conditions
    \[ <\text{flat.post}> ::= <\text{derived.post}> \&\& <\text{flat.inv}> \]
    // No base class post-condition to inherit

C++ example:

```cpp
class Derived {
public:
  // ...
  void bar() {
    invariant(); barPre();
    //...
    invariant(); barPost();
  }

protected:
  // ...
  /*inline*/ bool barPre() {
    return assert(/*bar pre-condition*/);
  }
  /*inline*/ bool barPost() {
    return assert(/*bar post-condition*/);
  }
};
```

Percolation pattern

• Define / specialize a method
  - C++ example (continued):
    protected:
    // ...
    /*inline*/ bool barPre() {
      return assert(/*bar pre-condition*/);
    }
    /*inline*/ bool barPost() {
      return assert(/*bar post-condition*/);
    }
                

Percolation pattern

• Override a method
  - Pre-conditions
    \[<\text{flat.pre}> ::= (<\text{derived.pre}> || <\text{base.pre}>^) \&\& <\text{flat.inv}>\]
    // Base and derived both have pre-conditions
    <\text{derived.pre}> and <\text{base.pre}> can be trivial
  - Post-conditions
    \[<\text{flat.post}> ::= (<\text{derived.post}> \&\& <\text{base.post}>^) \&\& <\text{flat.inv}>\]
    // Base and derived both have post-conditions
    <\text{derived.post}> and <\text{base.post}> can be trivial

Universiteit Antwerpen
Percolation pattern

- Override a method
  - C++ example:
    ```cpp
    class Derived {
    public:
        // ...
        /*virtual*/ void foo() {
            invariant(); fooPre();
            // ...
            invariant(); fooPost();
        }
    }
    ```

Percolation pattern

- Override a method
  - C++ example (continued):
    ```cpp
    protected:
        // ...
        /*inline*/ bool fooPre() {
            return assert(/*foo pre-condition*/ || Base::fooPre());
        }
        /*inline*/ bool fooPost() {
            return assert(/*foo post-condition*/ && base::fooPost());
        }
    }
    ```

Percolation pattern

- Override a method
  - C++ example (continued):
    ```cpp
    /*inline*/ bool fooPost() {
        return assert(/*foo post-condition*/ && base::fooPost());
    }
    ```

Percolation pattern

- Extend a method
  - Pre-conditions
    ```
    <flat.pre> ::= <base.pre> && <flat.inv>
    // Inherit base class pre-condition
    ```
  - Post-conditions
    ```
    <flat.post> ::= <base.post> && <flat.inv>
    // Inherit base class post-condition
    ```
Percolation pattern

- Advantages
  (+) Improved readability
  (only 4 extra calls per method)
  (+) Liskov Substitution Principle

- Disadvantages
  (-) Effort to design, implement, maintain
  (-) Performance penalty

Considerations

Verification

Beware of mistakes!
- Don't rule out the possibility that a failing assertion itself is wrong
- The risk of mistakes can be reduced by letting different people identify and code assertions
- Use negative assertion tests to verify the assertion and its action

Using assertions to design tests

- Testing is still required when using contracts
- Pre-conditions provide boundary values
- Post-conditions identify domains
- Cover all assertion branches
- Simulate failures in server objects to cause post-condition exceptions
- Rerun regression tests after disabling assertions
Pre-release considerations

Built-in test supports design-time bug prevention, effective testing and efficient debugging

- Begin as early as possible
- Postpone disabling them as long as possible
- Design to be robust, assertions as safety net

Beware of side effects!
- Assertions may not be enabled in the field
- Develop and test with pre-conditions enabled, enable post-conditions and invariants if possible
- BIT should be disabled by a compiler parameter
  - Manual revision is time-consuming and error-prone
- Rerun regression test after disabling BIT
  - Otherwise shipped system will not have been tested

Post-release considerations

Keeping assertions vs. Disabling assertions

Why disable?
- Better performance (most important reason)
- You’re already confident your system is reliable

Keeping assertions vs. Disabling assertions

Why keep?
- No need to maintain debug and release version
- No need for extra regression test run
- Easier problem diagnostics in the field
- Undefined behaviour when disabled
- Might have necessary side effects
Limitations and caveats

- Not feasible for all systems
- Testing is still required
- Assertions cannot detect all types of bugs
- Coverage analysis may report lower coverage
- Performance penalty
- Human factors

Questions?