Integration Testing

Arnav Gupta

April 5, 2024

Contents

1 Integration Testing

1.1 Strategy

How individual components are assembled to form larger program entities:

- **problem 1**: test component interaction
- **problem 2**: determine optimal order of integration

Strategy impacts:

- form of writing unit test cases
- type of test tools to use
- order of coding/testing components
- cost of generating test cases
- cost of locating and correcting detected defects

1.2 Testing

Objective: ensure components work in isolation and when assembled, requires a component dependency structure

If all components work in isolation, faults will most be interface related.

1.2.1 Stubs

Stubs replace called modules and can replace whole components. They must be declared/invoked as the real module (same name, param list, return type, modifiers)

Common functions:

- display/log trace message or passed params
- return value according to test objective

1.2.2 Drivers

Driver modules are used to call tested modules (param passing, handling return values), typically simpler than stubs.

1.3 Strategies

Comparison criteria for strategies: fault localization, effort needed, degree of testing of modules achieved, possiblity for parallel development

1.3.1 Big Bang Integration

Non-incremental strategy to integrate all components as a whole. Assumes components are initially tested in isolation.

Advantages

- convenient for small/stable systems

Disadvantages

- does not allow parallel development
- fault localization difficult
- easy to miss interface faults

1.3.2 Top-Down Integration

Incremental strategy to test high level components, then called components until lowest-level components. Possible to alter order to test as early as possible (critical or IO components).

Advantages:

- fault localization easier
- few or no drivers needed
- possiblity to obtain an early prototype
- testing can be in parallel with implementation
- different order of testing/implementation possible
- major design flaws found first

Disadvantages:

- needs lots of stubs
- potentially reusable components can be inadequately tested

1.3.3 Bottom-Up Integration

Incremental strategy to test low level components, then components calling them until highest-level components.

Advantages:

- fault localization easier
- no stubs needed
- reusable components tested thoroughly
- testing can be in parallel with implementation
- different order of testing/implementation possible

Disadvantages:

- needs lots of drivers
- high-level components tested last
- no concept of early skeletal system

1.3.4 Sandwich Integration

Combines top-down and bottom-up by distinguishing 3 layers: logic (top), middle, operational (bottom)

1.3.5 Other Integration Testing Strategies

- **risk driven integration**: integrate based on criticality
- **function/thread-based integration**: integrate components according to threads/functions they belong to

2 Testing Object-Oriented Systems

2.1 Motivation

Object-oriented code needs more testing. Where unit testing applies the same, OO concepts must be accounted for such as encapsulation, inheritance, polymorphism, abstract classes, and exceptions.

Unit and integration testing especially affected since they are white-box.

2.2 Class vs Procedure Testing

Procedural programming

- basic component is function
- testing method is based on IO relation

OO programming

- basic component is class
- objects are tested
- correctness cannot simply be defined as an IO relation, but also object state

Methods can be tested independently, though complexity lies in method interactions. Method behaviour is meaningless unless analyzed in relation to other operations and joint effect on shared state.

Testing classes requires recognizing:

- the identification of behaviour to be observed
- manipulation of object state without violating encapsulation
- polymorphism and dynamic binding to test all possible invocations of interfaces
- each exception to be tested

New fault models are required to target OO specific faults.

2.3 OO Integration Levels

Classes introduce a new abstraction level leading to more unit testing methods:

- **basic unit testing**: the testing of a single operation (method) of a class (intra-method testing)
- **unit testing**: the testing of methods within a class (intra-class testing), the integration of methods
- **intra-class testing**: black box and white box approaches (data flow), with:
  - each exception raised at least once

    - each interrupt forced to occur at least once
    - each attribute set and got at least once
    - testing based on states
    - done with big bang approach
    - full cycle of objects
    - complex methods tested with stubs or mocks

For integration testing, there are different scopes for interactions among classes:

- **cluster integration**
    - integration of 2+ classes through inheritance
        - incremental test of inheritance hierarchies
    - integration of 2+ clasxes through containment
    - integration of 2+ associated classes/clusters to form a component
    - big bang, bottom-up, top-down, scenario based
- **system/subsystem integration**
    - integration of components into a single application
    - similar techniques as cluster integration
    - client/server integration

2.4 Mock Objects

Testing of OO systems requires drivers and stubs, but it is difficult to flexibly stub dependent code without changing code under test or maintaining a library of stub objects (which is also difficult).

Mock object:

- a form of stubs based on interfaces
- easier to setup and control
- isolates code from details that may be filled in later
- can be refined incrementally by replacing with actual code

- based on dependency inversion principle
- test control mock behaviour so mock transparently replaces actual code

Creation can be done manually (more control) or using frameworks like Mockito.

2.5 Inheritance

Makes understanding code more difficult. Test suite for a method almost never adequate for overriding methods.

Inherited methods must be tested in the context of inheriting classes, with accidental reuse, abstract classes, and multiple inheritance.

Interactions possible to miss:

- inherited methods should be retested in the context of a subclass
  - modifying a superclass requires retesting subclasses
  - adding or modifying subclasses requires retesting methods inherited from each ancestor superclass
- overriding of methods
  - overriding subclass method has to be tested but different test sets are needed
  - reason 1: if test cases are derived from program structure (data and control flow), the structure of the overriding method may be different
  - reason 2: the overriding method behaviour could also be different
- integration and polymorphism
  - can make test sets grow exponentially

Coverage must be extended for inheritance to cover the level of coverage in the context of each class as separate measurements. 100% inheritance means all code fully exercised in each appropriate context.

2.6 Abstract Classes

Cannot be instantiated but they define an interface and behaviour that implementing classes must adhere to. Must test abstract classes for functional compliance.

Functional compliance: a module's compliance with some documented or published functional specification.

2.6.1 Abstract Test Pattern

Provides the following:

- a reusable way to build a test suite across descendants
- a test suite that can be reused for future descendants

Tests that define the *functionality* of the interface belong in the abstract test class. Test specific to an *implementation* belong in a concrete test class.

Rule 1

- write an abstract test class for every interface and abstract class
- an abstract test should have test cases that cannot be overridden
- should also have an abstract factory method for creating instances of the class to be tested

Rule 2

- write a concrete test class for every implementation of the interface (or abstract class)
- the concrete test class should extend the abstract test class and implement the factory method

2.7 Cluster Integration

Needs a class dependency tree since the dependency structure is not always that clear cut.

2.7.1 Big Bang

Recommmended only when the cluster is stable, small, and the components are tightly coupled.

2.7.2 Bottom-Up

Most widely used technique: integrate classes starting from leaves and moving up the depdendency tree

2.7.3 Top-Down

Widely used technique: integrate classes starting from the top and moving down the dependency tree to leaves.

2.7.4 Scenario-Based

Describes interaction (collaboration) of classes, represented as interaction diagrams.

Approach:

1. Map collaborations onto dependency tree
2. Choose a sequence to apply collaborations
3. Develop and run test exercising collaboration

2.8 Integration Order Problem

Issues related to integration order:

- not advised to perform a big-bang integration (must be done stepwise, leading to stubs)
- not always feasible to construct a stub that is simpler than the code it simulates
- stub generation cannot be automated since it requires an understanding of the semantics of the simulated functions
- fault potential for some stubs may be the same or higher than real function
- minimizing the number of stubs to be developed should result in drastic savings
- class dependency graphs usually do not form simple hierarchies but complex networks
- most OO systems have strong connectivity and cyclic interdependencies

2.8.1 Kung et al Strategy

Aims at producing a partial ordering of testing levels based on class diagrams.

Classes under test at given level should only depend on classes previously tested, no stub required since previously tested classes can be included in subsequent testing levels.

Class dependency info can either come from reverse engineering or design documentation.

Dynamic relationships between classes are not taken into account by Kung et al.

Some testing levels become infeasible because of abstract classes.

2.8.2 Object Relation Diagram

Uses static info only. The ORD for a program P is an edge-labelled graph where the nodes represent the classes in P and the edges represent the inheritance, composition, and association relations.

For any two classes:

- edge labelled I means subclass to superclass
- edge labelled C means from composed of to
- edge labelled A means from associates with to

Identiy the effect of a class change at the class level using a CFW. CFW(X) for a class X: set of classes that could be affected by a change to class X (should be retested)

Assuming the ORD is not modfied by the change, for adequate testing CFW(X) has to include subclasses of X, classes composed with X, and classes associated with X.

Can be defined from UML (more straightforward) or source code.

CFW(X) contains all the classes such that there is a directed path from them to X in the ORD.

Test order provides the tester with a road map for conducting integration testing. Desirable order is the one that minimizes the number of stubs or stubbing effort.

- test independent classes first, then dependent classes based on their relationships
- similarly a subclass would be tested after its superclasses

Order of Acyclic ORDs: can generate a test order that ensures that X is tested before all classes of CFW(X), so stubs are not required (simply a topological sort)

Order of Cyclic ORDs: topological sort cannot be applied, so this requires cycle breaking

Cluster: maximal set of nodes that are mutually reachable through the relation (strongly connected component)

Cycle breaking: identify and remove an edge from a cluster until the graph becomes acyclic

- each deletion results in at least one stub
  - cannot be a parent class
  - should not be a composition
  - association is good for deletion, every directed cycle always contains at least one association edge