Testing: Issues in OO Testing Class Testing

- OO based on hope that objects could be reused without
  - Modification
  - Additional testing
  - Based on notion that objects encapsulate functions and data that belong together

What is a unit in an object orientated system?

- Traditional systems define a unit as the smallest component that can be compiled and executed
  - Units are normally a component which in is only assigned to one programmer
- Two options for selecting units in object orientated systems:
  - Treat each class as a unit
  - Treat each method within a class as a unit

Advantages for Object Orientated Unit Testing

- Once a class is testing thoroughly it can be reused without being unit tested again
- UML class state charts can help with selection of test cases for classes
- Classes easily mirror units in traditional software testing

Disadvantages for Object Orientated Unit Testing

- Classes obvious unit choice, but they can be large in some applications
- Problems dealing with polymorphism and inheritance

Composition Issues

  - Objective of OO is to facilitate easy code reuse in the form of classes
  - To allow this each class has to be rigorously unit tested
  - Due to classes potentially used in unforeseeable ways when composed in new systems
  - Example: A XML parser for a web browser
  - Classes must be created in a way promoting loose coupling and strong cohesion

Encapsulation Issues

- Encapsulation requires that classes are only aware of their own properties, and are able to operate independently
- If unit testing is performed well, the integration testing becomes more important
- If you do not have access to source code then structural testing can be impossible
- If you violate encapsulation for testing purposes, then the validity of test could be questionable

Inheritance and Polymorphism

- The Issues Inheritance is an important part of the object oriented paradigm
- Unit testing a class with a super class can be impossible to do without the super classes methods/variables
- Polymorphism Issues
  - Repeatedly testing same methods
  - Time can then be wasted if not addressed
  - Potentially can be avoided, and actually save time

Levels of Object Orientated Test

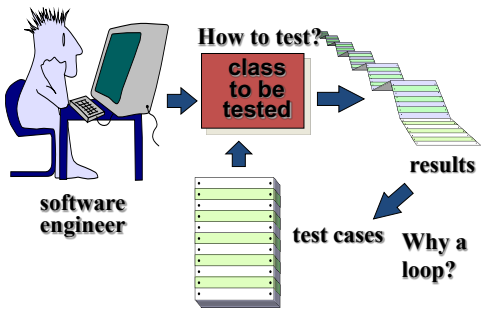
- There are generally 3 or 4 levels of testing for object orientated systems depending on our approach, consisting of:

1. Method Testing (Unit Testing)
2. Class Testing (Unit Testing/Intra class Testing)
3. Interclass Testing (Integration Testing)
4. System Testing

Class (Unit) Testing

- Smallest testable unit is the encapsulated class
- Test each operation as part of a class hierarchy
  because its class hierarchy defines its context of use
- Approach:
  - Test each method (and constructor) within a class
  - Test the state behavior (attributes) of the class between methods
- *How is class testing different from conventional testing?*
  - Conventional testing focuses on input-process-output,
    whereas class testing focuses on each method, then designing sequences of methods to exercise states of a class
  - But white-box testing can still be applied

Class Testing Process



Class Test Case Design

1. Identify each test case uniquely
   - Associate test case explicitly with the class and/or method to be tested
2. State the purpose of the test
3. Each test case should contain:
   a. A list of messages and operations that will be exercised as a consequence of the test
   b. A list of exceptions that may occur as the object is tested
   c. A list of external conditions for setup
   d. Supplementary information that will aid in understanding or implementing the test

– Automated unit testing tools facilitate these requirements

Challenges of Class Testing

- Encapsulation:
  – Difficult to obtain a snapshot of a class without building extra methods which display the classes' state
- Inheritance and polymorphism:
  – Each new context of use (subclass) requires re-testing because a method may be implemented differently (polymorphism).
  – Other unaltered methods within the subclass may use the redefined method and need to be tested
- White box tests:
  – Basis path, condition, data flow and loop tests can all apply to individual methods, but don't test interactions between methods

Random Class Testing

1. Identify methods applicable to a class
2. Define constraints on their use – e.g. the class must always be initialized first
3. Identify a minimum test sequence – an operation sequence that defines the minimum life history of the class
4. Generate a variety of random (but valid) test sequences – this exercises more complex class instance life histories
5. Example:
    1. An account class in a banking application has *open*, *setup*, *deposit*, *withdraw*, *balance*, *summarize* and *close* methods
    2. The account must be opened first and closed on completion
    3. *Open – setup – deposit – withdraw – close*
    4. *Open – setup – deposit –* [deposit | withdraw | balance | summarize] – withdraw – close*. Generate random test sequences using this template

Objectives

- To cover the strategies and tools associated with object oriented testing
    - Analysis and Design Testing
    - Unit/Class Tests
    - Integration Tests
    - System Tests

Object-Oriented Testing

- Analysis and Design:
    - Testing begins by evaluating the OOA and OOD models
    - How do we test OOA models (requirements and use cases)?
    - How do we test OOD models (class and sequence diagrams)?
    - Structured walk-throughs, prototypes
    - Formal reviews of correctness, completeness and consistency
- Programming:
    - How does OO make testing different from procedural programming?
    - Concept of a 'unit' broadens due to class encapsulation
    - Integration focuses on classes and their execution across a 'thread' or in the context of a use case scenario

- Validation may still use conventional black box methods

Completion Criteria

- When are we done testing?

1. One view: testing is never done… the burden simply shifts from the developer to the customer
2. Testing is done when you run out of time or money
3. Use a statistical model:
    - Assume that errors decay logarithmically with testing time
    - Measure the number of errors in a unit period
    - Fit these measurements to a logarithmic curve

Strategic Issues

- Issues to address for a successful software testing strategy:
    - Specify product requirements long before testing commences .For example: portability, maintainability, usability.
    - Understand the users of the software, with use cases
    - Develop a testing plan that emphasizes "rapid cycle testing" . Get quick feedback from a series of small incremental tests
    - Build robust software that is designed to test itself
      Use assertions, exception handling and automated testing tools ,Conduct formal technical reviews/inspections to assess test strategy and test cases .

Testing OOA and OOD Models

- The review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code level.
- If the error is not uncovered during analysis and propagated further more efforts needed during design or coding stages.
- By fixing the number of attributes of a class during the first iteration of OOA, the following problems may be avoided:

- Creation of unnecessary subclasses.

- Incorrect class relationships.

- Improper behavior of the system or its classes.

- Analysis and design models cannot be tested in the conventional sense, because they cannot be executed.

- Formal technical reviews can be used to examine the correctness and consistency of both analysis and design models.
- Correctness:

- Syntax: Each model is reviewed to ensure that proper modeling conventions have been maintained.

- Semantic: Must be judged based on the model's conformance to the real world problem domain by domain experts.

- Consistency:

- May be judged by considering the relationship among entities in the model.

- Each class and its connections to other classes should be examined.

- The Class-responsibility-collaboration model and object-relationship diagram can be used.

Testing Models

- Criteria

Correctness

Completeness

Consistency

- Early informal models are tested informally
- The criteria should be interpretive in the context of iterative incremental approach

Model Testing Approach

- Testing by comparison

- compares each model to its predecessor or to previous forms of the model

- Testing by inspection

- uses checklists to make sure that the model meets certain criteria

- Testing by verification

- follows certain steps to assure completeness and consistency of one part of the model with another

Examples of Analysis and Design Models to be Tested

- CRC cards

- English text descriptions of a single class, its responsibilities, and it collaborators with other classes

- Class specifications

- Complete specification of a class including its data structure, method names, number and type of parameters, return values, pre- and post-conditions.

Examples of Analysis and Design Models to be Tested

- Use cases

  - A representation of the systems usage

- State-Transition Models

  - State transition diagrams for classes, clusters, and subsystems

- Object network

  - Message sequence between methods in classes

- Transaction-Flow Models

Testing the Class Model

1. Revisit the Use Cases, CRC cards and UML class model.
   Check that all collaborations are properly represented. Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition
   - Example: in a point of sale system. A *read credit card* responsibility of a *credit sale* class is accomplished if satisfied by a *credit card* collaborator
2. Invert connections to ensure that each collaborator asked for a service is receiving requests from a reasonable source
   - Example: a credit card being asked for a purchase amount

3. These steps are applied iteratively to each class and through each evolution of the OOA model.

Unit Test

- What is a unit?

  A single, cohesive function?

  A function whose code fits on one page?

  Code that is assigned to one person?

- We can no longer test a single operation in isolation but rather as part of a class.
- In object-oriented programs, a unit is a method within a class.
- Smallest testable unit is the encapsulated class

Generating Test Cases for Unit Testing

- Statement coverage
- Graph based

Branch coverage

Condition coverage

Path coverage

- All unit testing methods are also applicable to testing methods within a class.

Class Test Case Design

Berard proposes the following approach:

1. Identify each test case uniquely
   - Associate test case explicitly with the class and/or method to be tested
2. State the purpose of the test
3. Each test case should contain:

a. list of specified states for the object that is to be tested

b. A list of messages and operations that will be exercised as a consequence of the test

c. A list of exceptions that may occur as the object is tested

d. A list of external conditions for setup (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)

e. Supplementary information that will aid in understanding or implementing the test

- Automated unit testing tools facilitate these requirements

OO Methods: Class Level 1

- Random testing - identify operations applicable to a class. Define constraints on their use.

identify a minimum test sequence

- an operation sequence that defines the minimum life history of the class (object)

generate a variety of random (but valid) test sequences

- exercise other (more complex) class instance life histories

- Example:

Class: Account

Operations: open, setup, deposit, withdraw, balance, summarize, creditlimit, close.

1. *Open – setup – deposit – withdraw – close*
2. *Open – setup – deposit –[deposit | withdraw | balance | summarize] * – withdraw – close*. Generate random test sequences using this template

OO Methods: Class Level 2

- Partition Testing - reduces the number of test cases required to test a class in much the same way as equivalence partitioning for conventional software

state-based partitioning

- categorize and test operations based on their ability to change the state of a class (e.g.: deposit, withdraw)

attribute-based partitioning

- categorize and test operations based on the attributes that they use (e.g.: creditlimit attribute)

category-based partitioning

- categorize and test operations based on the generic function each performs. (e.g.: (Init OP: open , setup) (Comp. OP: deposit, withdraw) (queries: balance, summarize, creditlimit) (Termination OP: close))

What Methods to Test

- New methods: defined in the class under test and not inherited or overloaded by methods in a superclass - Complete testing
- Inherited methods: defined in a superclass of the class under test - Retest only if the methods interacts with new or redefined method.
- Redefined methods: defined in a superclass of but redefined in the class under test - Complete retest reusing tests from the superclass.

Class Testing Techniques

- In addition to testing methods within a class (either glass box or black box), the following three techniques can be used to perform functional testing for the class:

  - State-Transition testing

  - Transaction-Flow testing

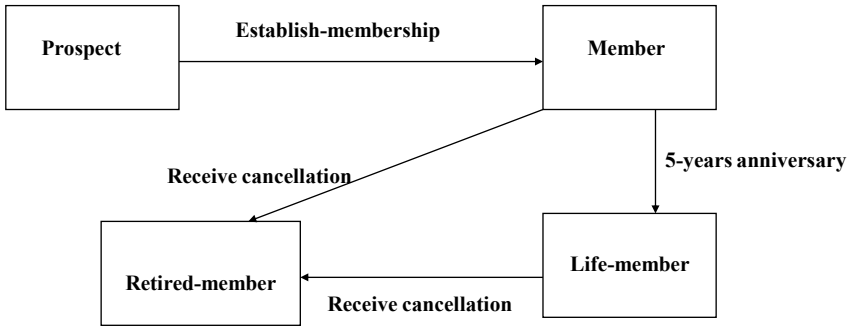
  - Exception testing

State-Transition Testing

- A state-transition model describes the different states and transitions of a class in the context of its position in the inheritance hierarchy.
- The *state* of an object is the combination of all the attribute values and objects that the object contains.
- An object may *transition* from a state to state as a result of an *event*, which may yield an *action.*

Example

Example



State Transition Testing

- Create test cases corresponding to each transition path that represent a full object life cycle
- Make sure each transition is exercised at least once.

OO Integration Testing

What: Integration testing is a phase of software testing in which individual software modules are combined and tested as a group. It follows unit testing and precedes system testing.

Why: The purpose of integration testing is to verify functional, performance and reliability requirements placed on *major design artefacts*.

How: integration testing is a logical extension of unit testing. In its simplest form, two units that have already been tested are combined into a component and the interface between them is tested.

OO) Integration vs RegressionTesting – *another (incremental development) point of view*

1. **Integration tests are performed when new code is added to an existing code base; Integration tests measure whether the new code works -- *integrates* -- with the existing code; these tests look for data input and output, correct handling of variables, etc.**
2. **Regression testing is the counterpart of integration testing: when new code is added to existing code, regression testing verifies that the existing code continues to work correctly, whereas integration testing verifies that the *new* code works as expected.**
3. **Regression testing can describes the process of testing new code to verify that this new code hasn't broken any old code.**

Integration Testing: What about OO?

- **The components to be tested are object classes that are instantiated as objects**
- **Larger grain than individual functions so approaches to white-box testing have to be extended.**
- **Levels of integration are less distinct in object-oriented systems**
- ***Cluster testing* is concerned with integrating and testing clusters of cooperating objects**
- **Identify clusters using knowledge of the operation of objects and the system features that are implemented by these clusters**

Integration Testing: What Clusters?

- **Use-case or scenario testing**

Testing is based on a user interactions with the system has the advantage that it tests system features as experienced by users.

- **Thread testing**

Tests the systems response to events as processing threads through the system.

- **Object interaction testing**

Tests sequences of object interactions that stop when an object operation does not call on services from another object.

OO) Integration Testing

- **Tests complete systems or subsystems composed of integrated components**
- **Integration testing should be black-box testing with tests derived from the specification**
- **Main difficulty is localising errors**
- **Incremental integration testing reduces this problem**

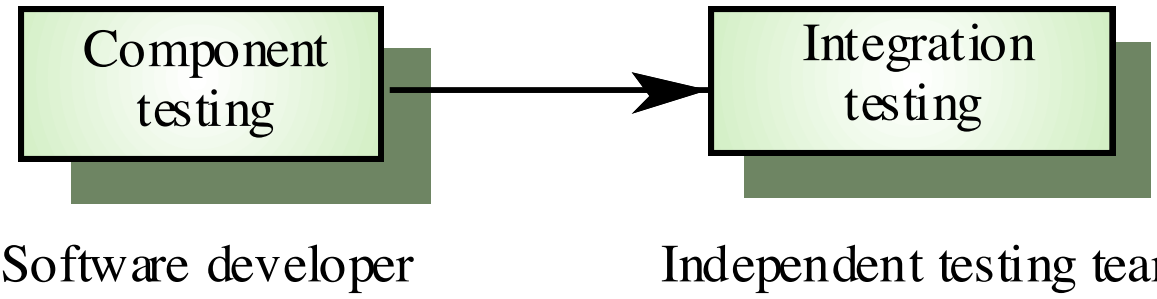
Integration of components

Component testing

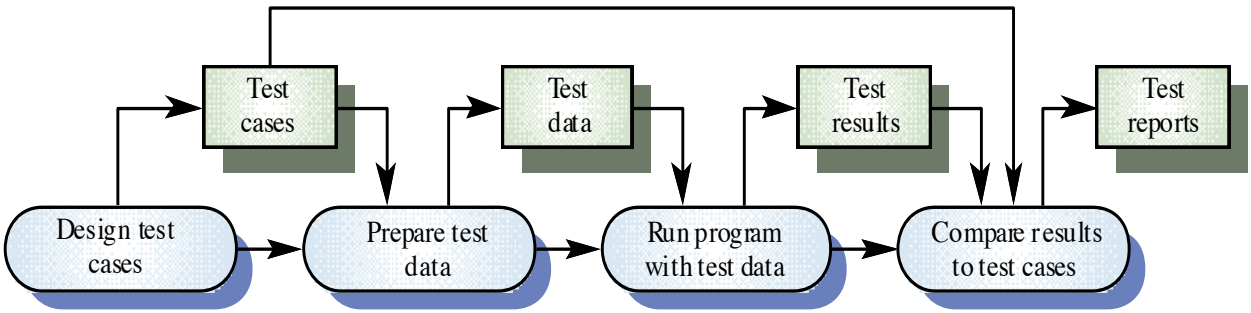
- **Testing of individual program components**
- **Usually the responsibility of the component developer.**
- **Tests are derived from the developer's experience**

Integration testing

- **Testing of groups of components integrated to create a system or sub-system**
- **The responsibility of an independent testing team**
- **Tests are based on a system specification**



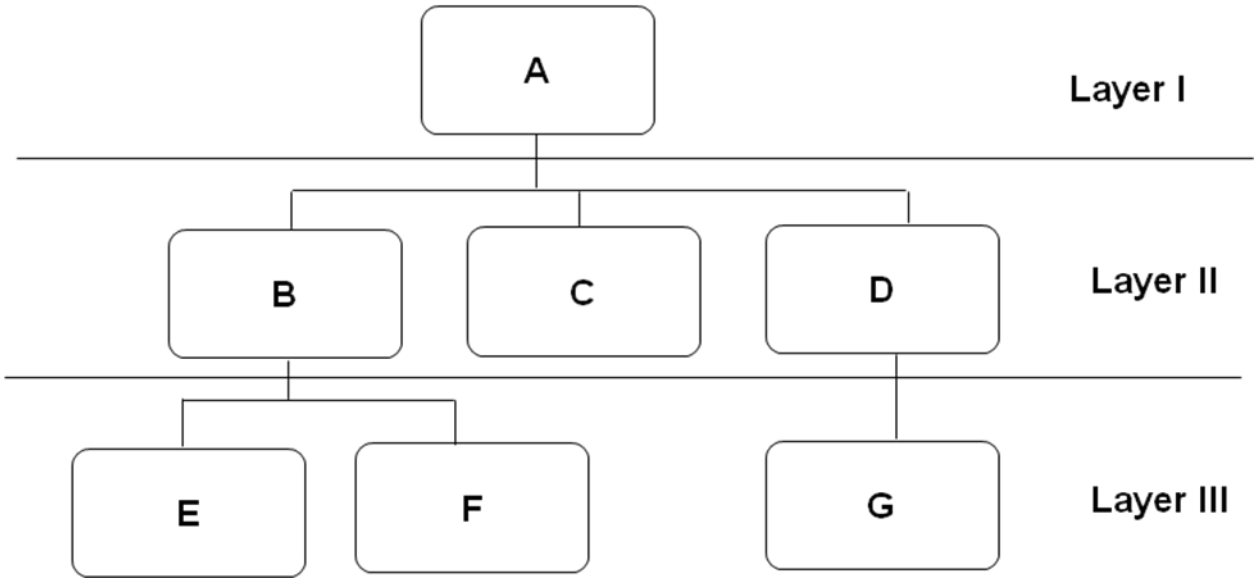
Integration of components: test process is the same



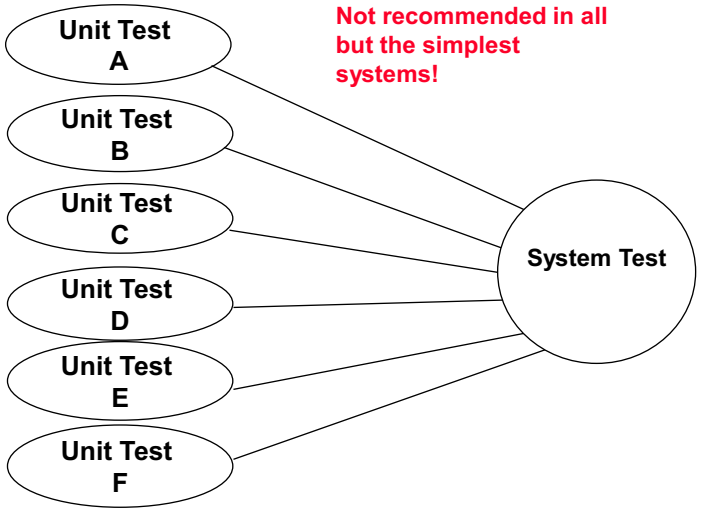
Integration Testing Strategy

- **The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design.**
- **The order in which the subsystems are selected for testing and integration determines the testing strategy**
  - **Big bang integration (Non incremental)**
  - **Bottom up integration**
  - **Top down integration**
  - **Sandwich testing**
  - **Variations of the above**
- **For the selection use the system decomposition from the System Design**

Example Design Structure: Three Layer Call Hierarchy



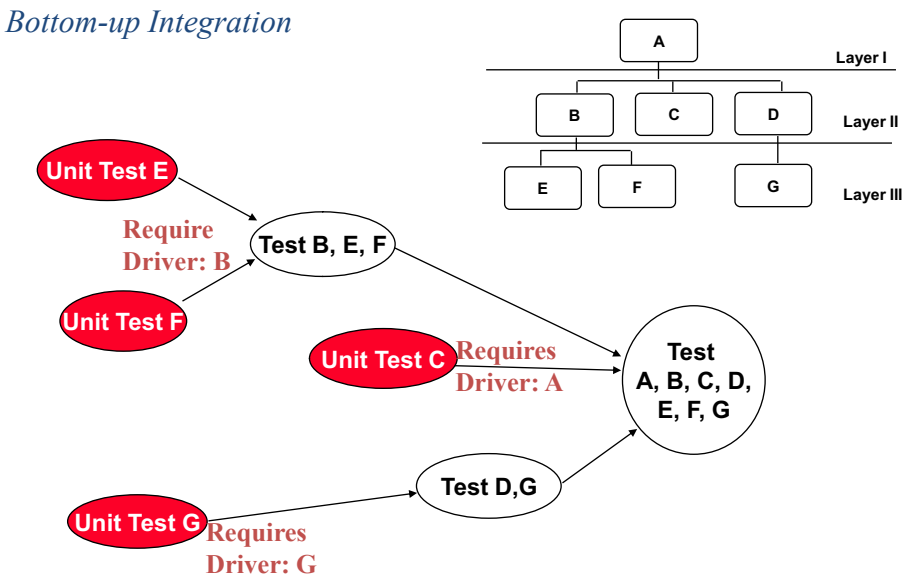
Integration Testing: *Big-Bang* Approach



Bottom-up Testing Strategy

- The subsystem in the lowest layer of the call hierarchy are tested individually
- Then the next subsystems are tested that call the previously tested subsystems
- This is done repeatedly until all subsystems are included in the testing

- Special program needed to do the testing, *Test Driver*:
  - A routine that calls a subsystem and passes a test case to it.



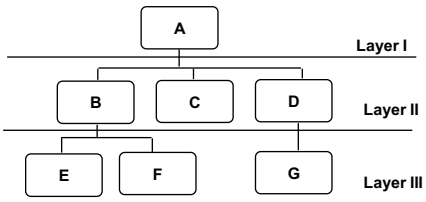
Advantages and Disadvantages of *bottom-up integration* testing

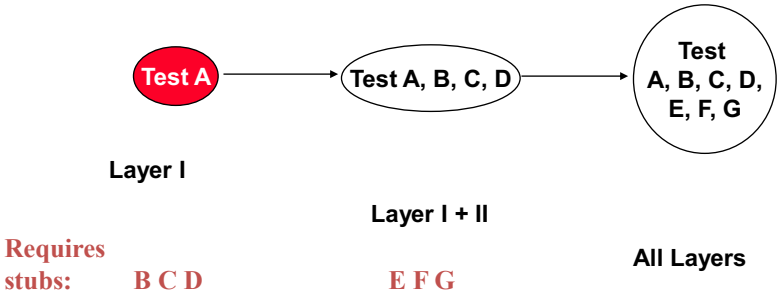
- Bad for functionally decomposed systems:
  - Tests the most important subsystem (UI) last
- Useful for integrating the following systems
  - Object-oriented systems
  - real-time systems
  - systems with strict performance requirements

Top-down Testing Strategy

- Test the top layer or the controlling subsystem first
- Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
- Do this until all subsystems are incorporated into the test
- Special program is needed to do the testing, *Test stub :*
  - A program or a method that simulates the activity of a missing subsystem by answering to the calling sequence of the calling subsystem and returning back fake data.

Top-down Integration Testing





Advantages and Disadvantages of *top-down integration* testing

- Test cases can be defined in terms of the functionality of the system (functional requirements)
- Writing stubs can be difficult: Stubs must allow all possible conditions to be tested.
- Possibly a very large number of stubs may be required, especially if the lowest level of the system contains many methods.
- One solution to avoid too many stubs: *Modified top-down testing strategy*
  - Test each layer of the system decomposition individually before merging the layers
  - Disadvantage of modified top-down testing: Both, stubs and drivers are needed

- Test cases can be defined in terms of the functionality of the system (functional requirements)
- Writing stubs can be difficult: Stubs must allow all possible conditions to be tested.
- Possibly a very large number of stubs may be required, especially if the lowest level of the system contains many methods.
- One solution to avoid too many stubs: *Modified top-down testing strategy*
  - Test each layer of the system decomposition individually before merging the layers
  - Disadvantage of modified top-down testing: Both, stubs and drivers are needed

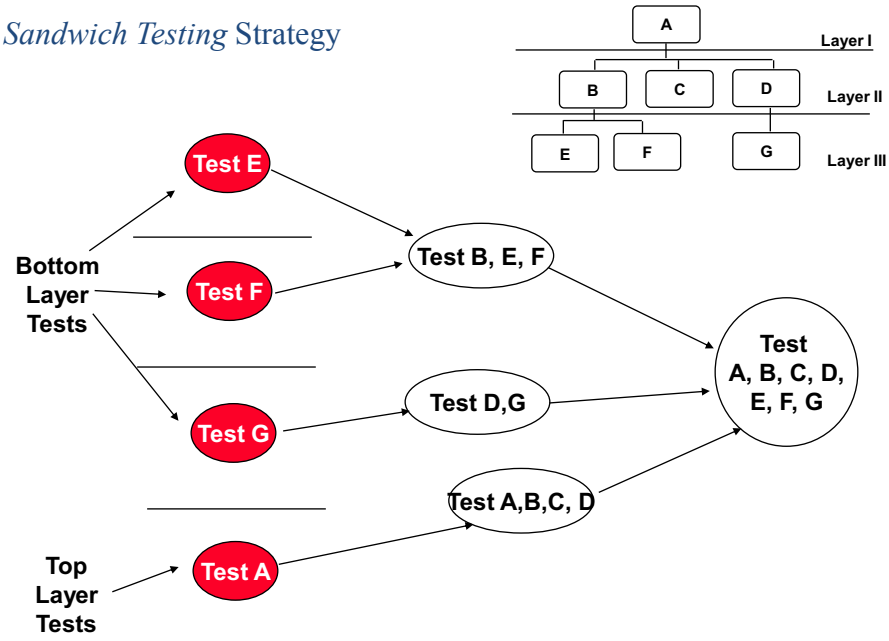
Sandwich Testing Strategy

- Combines top-down strategy with bottom-up strategy
- *The system is view as having three layers*

- A target layer ‘somewhere in the middle’
- A layer above the target
- A layer below the target
- Testing converges at the target layer

- How do you select the target layer if there are more than 3 layers?
  - Heuristic: Try to minimize the number of stubs and drivers

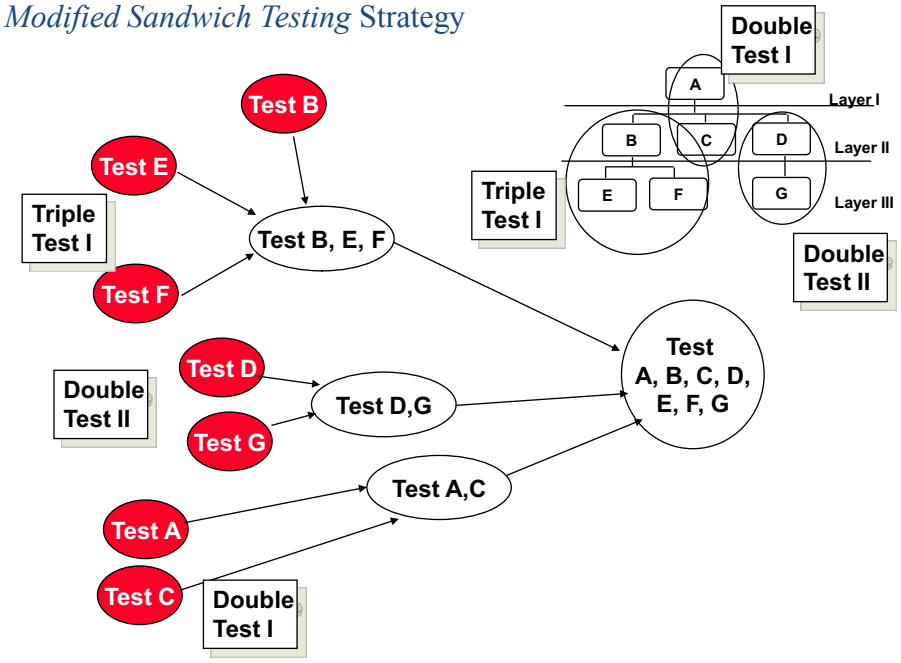
Sandwich Testing Strategy



Advantages and Disadvantages of *Sandwich Testing*

- Top and Bottom Layer Tests can be done in parallel
- Does not test the individual subsystems thoroughly before integration
- Solution: m*odified sandwich testing* strategy:
  - Test in parallel:
    - Middle layer with drivers and stubs
    - Top layer with stubs
    - Bottom layer with drivers
  - Test in parallel:
    - Top layer accessing middle layer (top layer replaces drivers)
    - Bottom accessed by middle layer (bottom layer replaces stubs)

Modified Sandwich Testing Strategy



Steps in Integration-Testing

1. Based on the integration strategy, *select a component* to be tested. Unit test all the classes in the component.
2. Put selected component together; do any *preliminary fix-up* necessary to make the integration test operational (drivers, stubs)
3. Do *functional testing:* Define test cases that exercise all uses cases with the selected component
4. Do *structural testing:* Define test cases that exercise the selected component
5. Execute *performance tests*
6. *Keep records* of the test cases and testing activities.
7. Repeat steps 1 to 7 until the full system is tested.

The primary *goal of integration testing is to identify errors* in the (current) component configuration.

Choosing an Integration Strategy?

- Factors to consider
  - Amount of test harness (stubs &drivers)
  - Location of critical parts in the system
  - Availability of hardware
  - Availability of components
  - Scheduling concerns
- Bottom up approach
  - good for object oriented design methodologies
  - Test driver interfaces must match component interfaces
  - Top-level components are usually important and cannot be neglected up to the end of testing
  - Detection of design errors postponed until end of testing
- Top down approach
  - Test cases can be defined in terms of functions examined
  - Need to maintain correctness of test stubs
  - Writing stubs can be difficult

Remember the Testing Life Cycle

Establish the test objectives

Design the test cases

Write the test cases

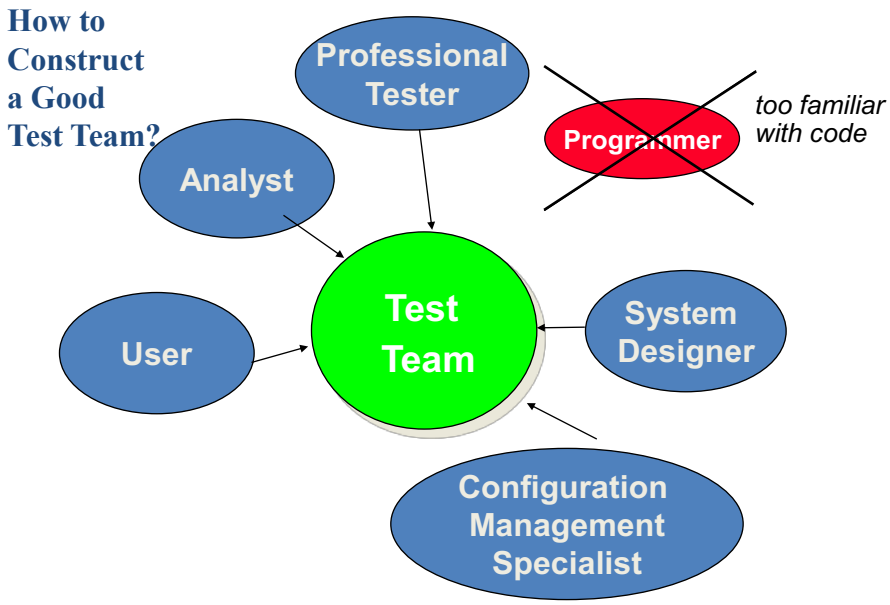
Test the test cases

Execute the tests

Evaluate the test results

Change the system

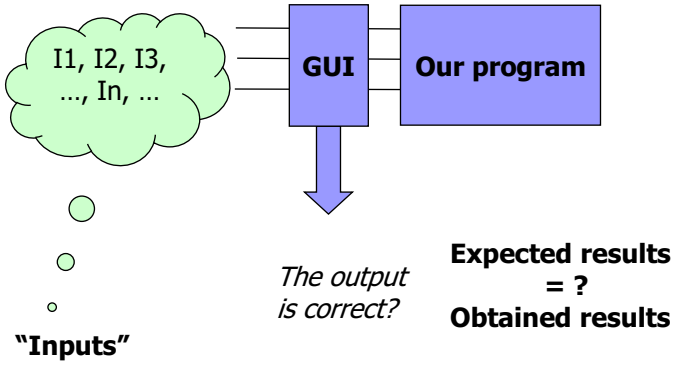
Do regression testing



GUI-based Testing

GUI-based Testing

- **One of the practical methods commonly used** to detect the presence of errors (*failures*) **in a computer program is to exercise it by using its Graphical User Interface.**



GUI-based Testing: again four main questions

- **At which level conducting the testing?**
  - ☐ Unit

- ☐ Integration
- ☐ System
- ☐ Regression

- **How to choose inputs?**
  - ☐ using the specifications/use cases/requirements
  - ☐ using the code
  - ☐ only considering the GUI (functionality, structure)
- **How to identify the expected output?**
  - ☐ test oracles
- **How good test cases are?**
  - ☐ when we can stop the testing activity

GUI …. what?

- GUI as a means to use/interact with the software systems
- GUI are nowadays almost ubiquitous, even in safety critical systems
- Different types of device (web, pc, tablet, palm, mobile)
- GUI interacts with the underlying code by method calls or messages
- GUI can exercise remote code
- GUI responds to user events (e.g., mouse clicks)
  - ☐ GUIs are event-driven systems
- Testing GUI correctness is critical for system usability, robustness and safety
- The whole system can be executed by means of the GUI

GUI …. more formally

A GUI (**Graphical User Interface**) is a hierarchical, graphical front end to a software system

A GUI contains **graphical objects** *w*, called widgets, each with a set of properties *p*, which have discrete values *v* at run-time.

At any time during the execution, the values of the properties of each widget of a GUI define the GUI state: {… (w, p, v), …}

A **graphical event** *e* is a state transducer, which yields the GUI from a state S to the next state S'.

- **Testing GUI software systems is different from testing non-GUI software**
  - ☐ **Non-GUI testing:** suites are composed of test cases that invoke methods of the system and catch the return value/s;

- **GUI-based testing:** suites are composed of test cases that are:
  - able to recognize/identify the components of a GUI;
  - able to exercise GUI events (e.g., mouse clicks);
  - able to provide inputs to the GUI components (e.g., filling text fields);
  - able to test the functionality underlying a GUI set of components;
  - able to check the GUI representations to see if they are consistent with the expected ones;
  - often, strongly dependent on the used technology;

GUI testing difficulties

- GUI test automation is difficult
- Often GUI test automation is technology-dependent
- Observing and trace GUI states is difficult
- UI state explosion problem
  - A lot of possible states of the GUI
- Controlling GUI events is difficult
  - Explosion of the possible combinations of events to do the same thing
- GUI test maintenance is hard and costly
- ……

GUI testing advantages

- Automation is feasible
  - Several frameworks and tools support it
- Easy to conduct for non-expert people
- It is funny to do
- ……

Which type of GUI-based testing?

- **System testing**
  - Test the whole system
- **Acceptance testing**
  - Accept the system
- **Regression testing**

- ☐ Test the system w.r.t. changes

GUI-based Acceptance Testing

Acceptance Tests are specified by the customer and analyst to test that the overall system is functioning as required (*Do developers build the right system?*).

How?

- **Manual Acceptance testing.** User exercises the system manually using his creativity.
- **Acceptance testing with "GUI Test Drivers"** (at the GUI level). These tools help the developer do functional/acceptance testing through a user interface such as a native GUI or web interface.
- **Table-based acceptance testing.** Starting from a user story (or use case or textual requirement), the customer enters in a table the expectations of the program's behavior.
- **Black-Box approaches** can be used to define test specification then executed manually, by means of the GUI or by table-based testing.

Approaches for GUI-based testing

- **Manual based**
    - Based on the domain and application knowledge of the tester
- **Capture and Replay**
    - Based on capture and replay of user sessions
- **Model-based testing**
    - Based on the execution of user sessions selected from a model of the GUI
        - ☐ Which type of model to use?
            - Event-based model
            - State-based model
            - Domain model
        - ☐ How do obtain the model to be used?
            - Specification-based model
            - Model recovered from existing software systems
            - Log-based model

Coverage criteria for GUI-based testing

- Conventional code-based coverage cannot be adequate;
- GUIs are implemented in terms of event-based system, hence, the abstraction level is different w.r.t. the conventional system code. So mapping between GUI events and system code can not be so easy.

- Possible coverage criteria:
  - ☐ **Event-coverage**: all events of the GUI need to be executed at least once
  - ☐ **State-coverage**: “all states” of the GUI need to be exercised at least once
  - ☐ **Functionality-coverage**: .. using a functional point of view

Event-based Model

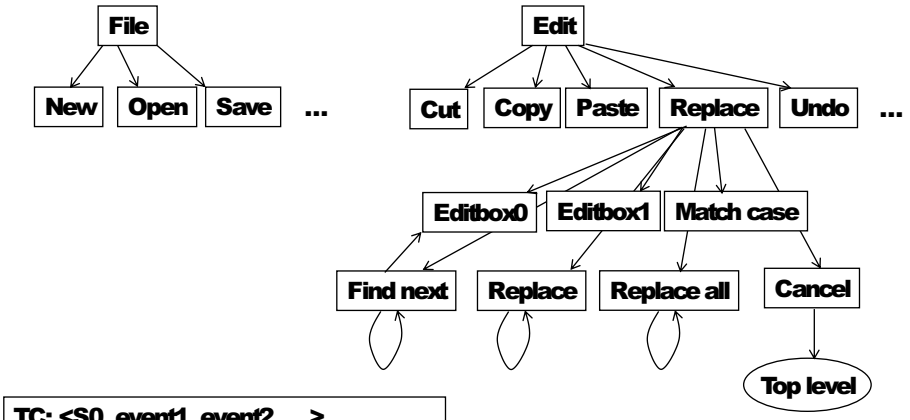
Model the space of **GUI event interactions** as a graph

Given a GUI:

1. create a graph model of all the possible sequences that a user can execute
2. use the model to generate event sequences

Event-based Model

-Event-flow graph“



TC: <S0, event1, event2, ...>
Oracle: <State1, State2, ...> &!CRASH

14

Event-based Model

Model Type:

- ☐ Complete event-model
- ☐ Partial event-model

Event types:

- ☐ Structural events (*Edit, Replace*)
- ☐ Termination events (*Ok, cancel*)
- ☐ System interaction events (*Editbox0, Find next*)

Coverage criteria

- ☐ Event coverage
- ☐ Event coverage according the exercised functionality
- ☐ Coverage of semantically interactive events
- ☐ 2-way, 3-way coverage
- ☐ ….

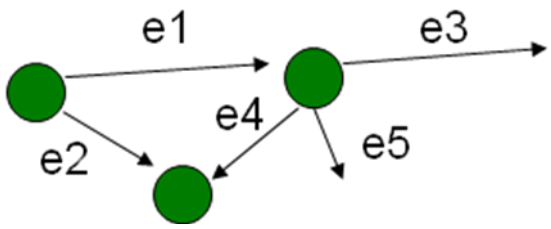
State-based Model

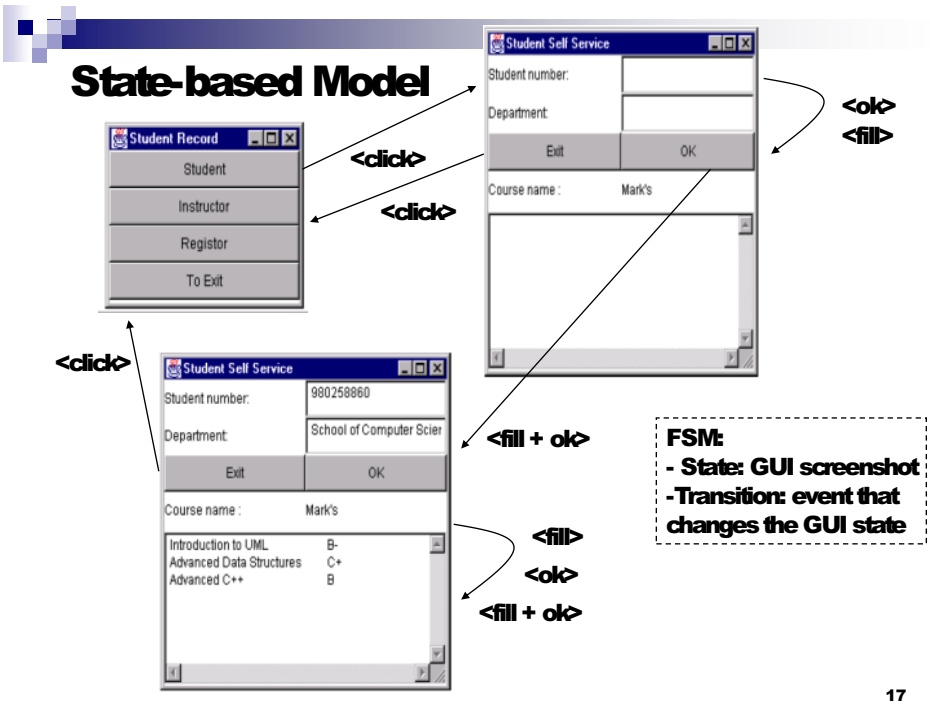
Model the space of **GUI event interactions as a state model**, e.g., by using a finite state machine (FSM):

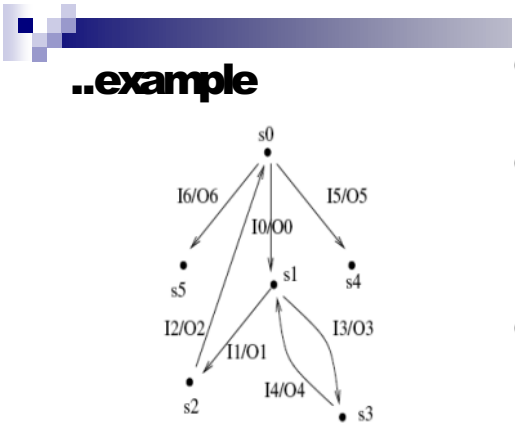
- States are screenshot/representation of the GUI
- Transitions are GUI events that change the GUI state

Given a GUI:

1. create a FSM of the possible sequences that a user can execute, considering the GUI state
2. use the FSM to generate event sequences







Problem: state explosion!

→ **Use of a state abstraction function**
It maps concrete states into abstract states (i.e., sets of concrete states)

I0: click *Student* button

O0: *Student Self Service* window shows up

I1: input student number 980258860 (an existing student number)

O1: *department name* shows *School of Computer Science*, and course-mark shows
Software Engineering: B-
Data Structures: C+
Programming Language with C++: B

I2: click *Exit* button

O2: *Student Self Service* window closed and *Student Record* window shows up

I3: input student number 999999999 (an invalid student number)

O3: information dialog box appears

I4: click *close* button in the information dialog box

O4: information dialog box closed and *Student Self Service* window shows up

I5: click *Instructor* button

O5: *Instructor password* window shows up

I6: click *Registrar* button

O6: *Registrar password* window shows up

Log-based Recovered Model

- **How do obtain the model?**
  - ☐ starting from system specification or requirements
  - ☐ starting from the system (i.e., reverse engineering)

1. trace some system executions (at method calls level)
2. infer a model
3. refine it manually, if needed

Test oracles for GUI-based testing

- It could be difficult to detect faults looking the GUI
  - Crash testing is often used;
- In a GUI test case, an incorrect GUI states can take the user to an **unexpected/wrong interface screen** or it can make the **user unable to do a specific action**;
  - e.g., after the click of a button, we try to click the button again but we fail since the button no longer exists, after the first click.
  - A GUI state can be "represented" by the components expected to be part of the GUI in a give time and their state/value
    - e.g., window position, GUI objects, GUI title, GUI content, GUI screenshots,

GUI errors: examples

- Incorrect functioning
- Missing commands (e.g., GUI events)
- Incorrect GUI screenshots/states
- The absence of mandatory UI components (e.g., text fields and buttons)
- Incorrect default values for fields or UI objects
- Data validation errors
- Incorrect messages to the user, after errors
- Wrong UI construction

GUI-based Testing: process

1. Identify the testing objective by defining a coverage criteria
2. Generate test cases from GUI structure, specification, model
   - Generate sequences of GUI events
   - Complete them with inputs and expected oracles
   - Define executable test cases
3. Run them and check the results

GUI-based Regression Testing

- GUI-based testing means to **execute the GUI of a system exercising its GUI components**;

- A small changes in the GUI layout can make the GUI test cases old and useless;
- Hence, GUI-based test suite need to be maintained and often changed

  → supporting tools are welcomed
- Often, GUIs are realized by means of rapid prototyping or automatic framework. This requires an efficient approach to generate and maintain GUI test suite

  → supporting tools are welcomed

Capture and Replay

A capture and replay testing tool captures user sessions (user inputs and events) and store them in scripts (one per session) suitable to be used to replay the user session.

An ad-hoc infrastructure is needed to intercept GUI events, GUI states, thus storing user sessions and also to be able to replay them.

- they can work at application or VM level

Recorded information

- Inputs, outputs, and other information needed to replay a user session need to be recorded during the capture process.
- Examples:
  - ☐ General information: date/time of recording, etc.
  - ☐ System start-up information
  - ☐ Events from test tool to system
    - ☐ Point of control, event
  - ☐ Events from system to test tool
    - ☐ Checkpoints / expected outputs
  - ☐ Time stamps

Capture and Replay: the process

1. The tester interacts with the system GUI to run the system, thus generating sessions of sequence of mouse clicks, UI and keyboard events;
2. The tool captures and stores the user events and the GUI screenshots;
   - a script is produced per each user session
3. The tester can automatically replay the execution by running the script
   - the script can be also changed by the tester
   - the script can be enriched with expected output, checkpoints

- the script can be replicated to generate many variants (e.g., changing the input values)

4. In case of GUI changes, the script must be updated

Tools for GUI-based testing

- **Marathon**
- **Abbot**
- **Guitar**
- **HtmlUnit, HttpUnit,JWebUnit**
- **HtmlFixture**
- **Selenium**

System Testing

Different Types

- Recovery testing
- Security testing
- Stress testing
- Performance testing
- Recovery testing
    - Tests for recovery from system faults
    - Forces the software to fail in a variety of ways and verifies that recovery is properly performed
    - Tests reinitialization, check pointing mechanisms, data recovery, and restart for correctness
- Security testing
    - Verifies that protection mechanisms built into a system will, in fact, protect it from improper access.
- Stress testing
    - Executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance testing
    - Tests the run-time performance of software within the context of an integrated system

- Often coupled with stress testing and usually requires both hardware and software instrumentation
- Can uncover situations that lead to degradation and possible system failure

System Testing

- Functional Testing
  - Validates functional requirements
- Performance Testing
  - Validates non-functional requirements
- Acceptance Testing
  - Validates clients expectations

Functional Testing

Goal: Test functionality of system

- Test cases are designed from the requirements analysis document (better: user manual) and centered around requirements and key functions (use cases)
- The system is treated as black box
- Unit test cases can be reused, but new test cases have to be developed as well.

Performance Testing

Goal: Try to violate non-functional requirements

- Test how the system behaves when overloaded.
  - Can bottlenecks be identified? (First candidates for redesign in the next iteration)
- Try unusual orders of execution
  - Call a receive() before send()
- Check the system's response to large volumes of data
  - If the system is supposed to handle 1000 items, try it with 1001 items.
- What is the amount of time spent in different use cases?
  - Are typical cases executed in a timely fashion?

Types of Performance Testing

- Stress Testing
  - Stress limits of system
- Volume testing
  - Test what happens if large amounts of data are handled

- Configuration testing
  - Test the various software and hardware configurations
- Compatibility test
  - Test backward compatibility with existing systems
- Timing testing
  - Evaluate response times and time to perform a function
- Security testing
  - Try to violate security requirements
- Environmental test
  - Test tolerances for heat, humidity, motion
- Quality testing
  - Test reliability, maintain- ability & availability
- Recovery testing
  - Test system's response to presence of errors or loss of data
- Human factors testing
  - Test with end users.

Acceptance Testing

- Goal: Demonstrate system is ready for operational use
  - Choice of tests is made by client
  - Many tests can be taken from integration testing
  - Acceptance test is performed by the client, not by the developer.
- Alpha test:
  - Client uses the software at the developer's environment.
  - Software used in a controlled setting, with the developer always ready to fix bugs.
- Beta test:
  - Conducted at client's environment (developer is not present)
  - Software gets a realistic workout in target environment

The 4 Testing Steps

1. **Select what has to be tested**
   - **Analysis: Completeness of requirements**
   - **Design: Cohesion**
   - **Implementation: Source code**
2. **Decide how the testing is done**
   - **Review or code inspection**
   - **Proofs (Design by Contract)**
   - **Black-box, white box,**
   - **Select integration testing strategy (big bang, bottom up, top down, sandwich)**
3. **Develop test cases**
   - **A test case is a set of test data or situations that will be used to exercise the unit (class, subsystem, system) being tested or about the attribute being measured**
4. **Create the test oracle**
   - **An oracle contains the predicted results for a set of test cases**
   - **The test oracle has to be written down before the actual testing takes place.**

Guidance for Test Case Selection

- **Use *analysis knowledge* about functional requirements (black-box testing):**
  - **Use cases**
  - **Expected input data**
  - **Invalid input data**
- **Use *design knowledge* about system structure, algorithms, data structures (white-box testing):**
  - **Control structures**
    - **Test branches, loops, ...**
  - **Data structures**
    - **Test records fields, arrays, ...**

- Use *implementation knowledge* about algorithms and datastructures:
  - Force a division by zero
  - If the upper bound of an array is 10, then use 11 as index.

Summary

- Testing is still a black art, but many rules and heuristics are available
- Testing consists of
  - Unit testing
  - Integration testing
  - System testing
  - Acceptance testing
- Design patterns can be used for integration testing
- Testing has its own lifecycle

System Testing

- Software may be part of a larger system. This often leads to “finger pointing” by other system teams
- Finger pointing defense:
  1. Design error-handling paths that test external information
  2. Conduct a series of tests that simulate bad data
  3. Record the results of tests to use as evidence