Testing Object-Oriented Software (an Overview)

Software Testing and Verification

Lecture 12

Prepared by
Stephen M. Thebaut, Ph.D.
University of Florida
Reference:


(Other sources are readily identifiable via your favorite Internet search engine.)
Similarities and Differences

• The same basic pattern of testing **procedural software**...
  - using functional (black-box) tests supplemented by structural (white-box) tests,
  - working from the unit level toward the system level with one or more integration steps,

...also applies to testing **object-oriented software**.

(cont’d)
But some features of O-O languages require special strategies and/or considerations:
- the inherently state dependent behavior of methods,
- encapsulation of methods and state,
- inheritance, and
- polymorphism and dynamic binding.

(cont’d)
Similarities and Differences (cont’d)

• We begin by summarizing the issue(s) and testing-related considerations associated with each of these.

• Next, we present simple models for unit (object class) and integration (initial inter-object class) testing.

• Higher-level testing of object-oriented systems is basically similar to that of other systems, and is not considered further here.
State Dependent Behavior of Methods

• Method behavior may depend on both parameter values ("inputs") and object state.

• For example, consider a linear "list" object class with Boolean inspection method "is_empty?".

• The result – "true" or "false" – clearly depends on the state of "list".

(cont’d)
State Dependent Behavior of Methods (cont’d)

• Thus, test case design strategies that do **not** explicitly take state dependent program behavior into account must be modified to do so.

• Consider, for example, **Input Space Partitioning** and **Cause-Effect Analysis**. How might these approaches be modified to account for state-dependent program behavior?
State Dependent Behavior of Methods (cont’d)

- Ad-hoc modifications of these approaches are usually straightforward – especially when augmented with state machine models of program behavior.
- We’ll come back to this when considering object class testing...
Encapsulation of Methods and State

- Objects are comprised of both public and private attributes and methods.
- This is problematic when effects of operations are private and therefore “hidden” from the tester.
- The effect of method “append” on object class “list”, for example, may only be visible through a limited set of public attributes or “inspector operations.”

(cont’d)
Encapsulation of Methods and State (cont’d)

- **Special instrumentation** may be required when methods result in hidden state changes, or are themselves private.
Inheritance

• Inheritance is an abstraction mechanism that allows classes to be specialized or extended from one or more other classes.

• A “child” class inherits attributes and methods from its ancestor classes, changing some, and adding others.

(cont’d)
Inheritance (cont’d)

• A testing issue which arises involves deciding which methods in a derived class need to be (re-)tested.

• *New* methods obviously must be, as must *changed* methods unless it can be shown that their behavior will be unaffected.

• *Unchanged* methods which *interact* with changed or new methods may also need to be tested in the context of those interactions.

(cont’d)
• In general, testers must determine when new tests must be designed, when old tests should be re-run, and when re-testing can be avoided altogether.
Polymorphism and Dynamic Binding

• Many object-oriented languages allow variable types and method bindings to change dynamically.

• For example, when the generic elements of object class “list” are dynamically bound to particular data types, list operations such as “append(value)” will be bound to particular methods.

(cont’d)
Testers must be aware of the bindings that may occur, and systematically determine which methods (and which combinations of methods) to test.
Unit (Object Class) Testing
What are the “units” of O-O programs?

- “Unit testing” of O-O software often focuses on object classes as opposed to individual methods, as testing methods in isolation is not always practical since...
  - Method behavior can be obscured (due to encapsulation) when considered in isolation.
  - Methods often interact in altering object state.
State Machine Models

- State machine models are often useful for describing object class states and transitions.
- "States" can be inferred from descriptions of methods that act differently or return different results depending on the state of the object. (Consider Algebraic Specification axioms that incorporate conditionals.)
- Partitioning and combinatorial test case design techniques may be applied on a state-by-state basis.
A Simple Example

Consider the operations for a generic Stack object:

**New** – Bring a stack into existence
**Push** – Add an element to the top of a stack
**Top** – Evaluate the top element of a stack (without removing it)

(cont’d)
Retract – Remove the top element from a stack and return the modified stack

Is_Empty – True if and only if there are no elements on a stack

(cont’d)
A Simple Example (cont’d)

With method signatures:

New -> Stack
Push (Stack, Elem) -> Stack
Top (Stack) -> Elem
Retract (Stack) -> Stack
Is_Empty (Stack) -> Boolean

(cont’d)
A Simple Example (cont’d)

• Suppose the stack specification requires a storage capacity of $N$ elements. How many different states would be required to represent every possible count of the elements stored?

• How might the states be partitioned into equivalence classes if the goal is to produce a simple model reflecting how state affects method behavior?

(Cont’d)
A Simple Example (cont’d)

• Suppose the stack specification requires a storage capacity of \(N\) elements. How many different states would be required to represent every possible count of the elements stored? \(N+1\) (including “0”)

• How might the states be partitioned into equivalence classes if the goal is to produce a simple model reflecting how state affects method behavior?

(cont’d)
A Simple Example (cont’d)

- Suppose the stack specification requires a storage capacity of \( N \) elements. How many different states would be required to represent every possible count of the elements stored? \( N+1 \) (including “0”)

- How might the states be partitioned into equivalence classes if the goal is to produce a simple model reflecting how state affects method behavior?

(cont’d)
A Simple Example (cont’d)

- The Boolean inspection method “Is_Empty” suggests two important abstract states for modeling the state-dependent behavior of methods: empty and non-empty
- “Top” is usually not defined in state empty.
- “Retract” also acts differently depending on the state of the stack…
A Simple Example (cont’d)

State Machine Model

**empty**
- New
- Push
- Retract

**non-empty**
- Push
- Retract
- Retract

(cont’d)
A Simple Example (cont’d)

• Test cases (sequences of method calls) should be designed which cover all transitions in the state machine model.

• State machine models are sometimes included in object-oriented program specifications.

• In the UML family of notations, they take the form of “state diagrams.”
Other object class testing strategies

- Functional test cases based on the state machine model should be augmented with *structural tests* derived from class source code.
- Special attention should be given to *exception handling*.
- Design tests for *intra-class polymorphic calls*. (Polymorphic attributes give rise to different “class families”. All interactions between the methods of a class and each “class family” should be verified.)
Integration (Initial Inter-Object Class) Testing
Object class integration

• Higher level O-O testing begins with the integration of (already unit-tested) object classes to form *inter-object class* functional entities.

• The primary objective of testing during this process is to discover interface and blatant higher-level design errors among the object classes being integrated.

• As with imperative software, integration should proceed *incrementally*.

(cont’d)
Use/include relations

• Just as a calling hierarchy allows design of an integration strategy for imperative software, use/include relations serve this purpose for object-oriented software.

• Object classes A and B are related by a use/include relation if objects of class A make method calls on objects of class B, or if class A includes class B.

• Use/include relations may be derived from a conventional UML class diagram.
Incremental strategy

- Since there is generally no single “root” class, testing usually proceeds cluster by cluster in a “bottom-up” fashion, starting with “leaf” classes that depend on no others.
- The “brute force” approach of testing all combinations of calls and states while climbing use/include relations is impractical for most non-trivial systems.

(cont’d)
Practical test coverage strategies

- Coverage strategies based on the (other, less conservative) combinatorial approaches considered previously are therefore more appropriate.

- Nominal interaction combinations may be identified from UML sequence or collaboration diagrams.

- Such diagrams can be thought of as test scenarios created during system design.

(cont’d)
Practical test coverage strategies (cont’d)

- Unexpected or illegal interaction sequences should also be exercised to check error handling.
Scaffolding for O-O testing

- Both Unit (intra-class) and integration (inter-class) testing require drivers and stubs to exercise classes under test, and oracles to interpret/inspect test results.
- Constructing drivers and stubs is similar to that for procedural software.
- **Encapsulation** of object state, however, can make the construction of oracles more difficult.
The result of executing a sequence of methods consists of both outputs produced and changes in object state.

For example, if

```plaintext
Top(Push(stack, Top(stack)))
```

does not add a copy of the top-most element to the stack, the result is erroneous regardless of the output produced.
Oracles for Object Classes (cont’d)

Expected result of $\text{Top}($Push$(\text{stack}, \text{Top}(\text{stack})))$

(cont’d)
Oracles for Object Classes (cont’d)

Expected result of $\text{Top}(\text{Push}(\text{stack}, \text{Top}(\text{stack})))$

Output: $X$

(cont’d)
Oracles for Object Classes (cont’d)

Expected result of $\text{Top(\text{Push(stack,Top(stack))})}$

**Output:** X

**State change:**

![Diagram showing state change from X Y Z to X X Y Z](image-url)
Oracles for Object Classes (cont’d)

• Therefore, Oracles need to check the correctness of both output and state, but the state of objects may not be directly accessible.

• One solution: build oracles to subvert the encapsulation by modifying source code to allow inspection of private variables.

• If implemented, should such modifications be removed after testing, or left in the delivered code?

(cont’d)
Oracles for Object Classes (cont’d)

- If modifications are *removed*, we risk differences in behavior between what is tested and what is used...e.g.,
  - masked or introduced faults and/or
  - performance differences
- Modifications *left in* the code (or design rules requiring programmers to provide *observability interfaces*) avoid such discrepancies, but incur some overhead.

(cont’d)
Oracles for Object Classes (cont’d)

• Ideally, the interface created for inspection of object state should be separated from the main class, as allowed, for example, by C++ “friend” classes†.

• An interface that produces a readable representation of object values is also useful in debugging as well as testing.

†A friend class in C++ can access the “private” and “protected” members of the class in which it is declared as a friend.
Want to learn more?

- For more on testing O-O software, see Pezze and Young, or one of the many other recent texts/manuscripts covering this topic.
- Note that much of the available “hands-on O-O testing” material is both source code and tool specific.
Testing Object-Oriented Software (an Overview)

Software Testing and Verification

Lecture 12

Prepared by
Stephen M. Thebaut, Ph.D.

University of Florida