

# SE 2AA4, CS 2ME3 (Introduction to Software Development)

Winter 2018

## 35 Analysis (Ch. 6) DRAFT

Dr. Spencer Smith

Faculty of Engineering, McMaster University

December 15, 2017



## 35 Analysis (Ch. 6) DRAFT

- Administrative details
- Module testing
- Integration testing
- Testing OO programs
- Testing concurrent and real-time systems
- Mutation testing
- Analysis
  - ▶ Code walk throughs and inspections
  - ▶ Correctness proofs
  - ▶ Symbolic execution
  - ▶ Model checking
- Debugging

# Administrative Details

TBD

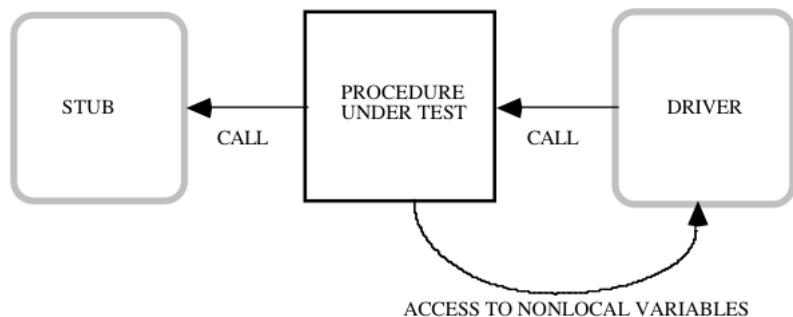
# The Oracle Problem Continued

- Devising an oracle for concurrent and real time systems
  - ▶ The challenge is not coming up with the oracle
  - ▶ The challenge is having adequate test coverage (or analysis) of all of the cases (more on this later)
- Other examples where an oracle is challenging to devise
  - ▶ Artificial general intelligence
  - ▶ Machine learning
  - ▶ Simulation with random numbers

# Module Testing

- Scaffolding needed to create the environment in which the module should be tested
- Stubs - a module used by the module under test
- Driver - module activating the module under test

# Testing a Functional Module



# Integration Testing

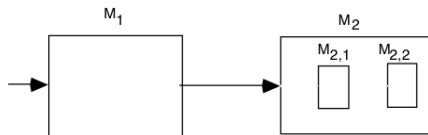
- Big-bang approach
  - ▶ First test individual modules in isolation
  - ▶ Then test integrated system
- Incremental approach
  - ▶ Modules are progressively integrated and tested
  - ▶ Can proceed both top-down and bottom-up according to the USES relation

# Integration Testing and USES relation

- If integration and test proceed bottom-up only need drivers
- Otherwise if we proceed top-down only stubs are needed

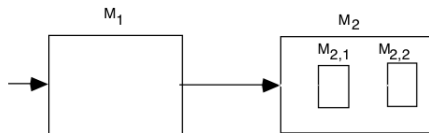


# Example



- $M_1$  USES  $M_2$  and  $M_2$  IS\_COMPOSED\_OF  $\{M_{2,1}, M_{2,2}\}$
- In what order would you test these modules?

# Example

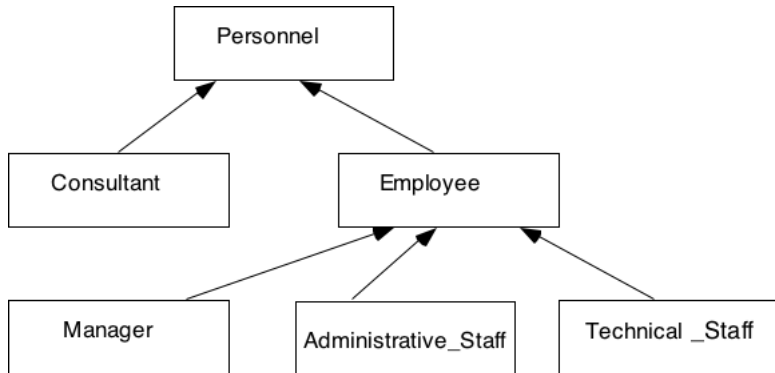


- $M_1$  USES  $M_2$  and  $M_2$  IS\_COMPOSED\_OF  $\{M_{2,1}, M_{2,2}\}$
- Case 1
  - ▶ Test  $M_1$  providing a stub for  $M_2$  and a driver for  $M_1$
  - ▶ Then provide an implementation for  $M_{2,1}$  and a stub for  $M_{2,2}$
- Case 2
  - ▶ Implement  $M_{2,2}$  and test it by using a driver
  - ▶ Implement  $M_{2,1}$  and test the combination of  $M_{2,1}$  and  $M_{2,2}$  (i.e.  $M_2$ ) by using a driver
  - ▶ Finally implement  $M_1$  and test it with  $M_2$  using a driver for  $M_1$

# Testing OO and Generic Programs

- New issues
  - ▶ Inheritance
  - ▶ Genericity
  - ▶ Polymorphism
  - ▶ Dynamic binding
- Open problems still exist

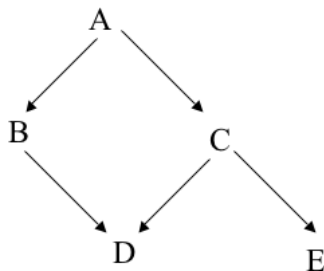
# Inheritance



# How to Test Classes of the Hierarchy

How would you approach testing for a class hierarchy?

# How to Test Classes of the Hierarchy



- “Flattening” the whole hierarchy and considering every class as totally independent component
- This does not exploit incrementality
- Finding an ad-hoc way to take advantage of the hierarchy
- Think about testing `PointT.java` and `PointMassT.java`

# A Sample Strategy

- A test that does not have to be repeated for any heir
- A test that must be performed for heir class X and all of its further heirs
- A test that must be redone by applying the same input data, but verifying that the output is not (or is) changed
- A test that must be modified by adding other input parameters and verifying that the the output changes accordingly

# Testing Concurrent and Real-time Systems

What are the challenges for testing concurrent and real-time systems?



# Testing Concurrent and Real-time Systems

- Nondeterminism inherent in concurrency affects repeatability
- For real-time systems, a test case consists not only of input data, but also of the times when such data are supplied
- Many potential time traces for the different inputs
- System changes depends on the control actions
- Considerable care and detail when testing real-time systems

# Testing your Tests

- How did we estimate the number of errors in our code?
- Can any of the ideas from estimating the number of errors in our code be used to test our tests?

# Testing your Tests: Mutation Testing

- Generate changes to the source code, called mutants, which become code faults
- Mutants include changing an operation, modifying constants, changing the order of execution, etc.
- The adequacy of a set of tests is established by running the tests on all generated mutants

# Analysis Versus Testing

- Testing characterizes a **single** execution
- Analysis characterizes a **class** of executions; it is based on a **model**
- They have complementary advantages and disadvantages

# Informal Analysis Techniques and Code Walkthroughs

- Recommended prescriptions
  - ▶ Small number of people (three to five)
  - ▶ Participants receive written documentation from the designer a few days before the meeting
  - ▶ Predefined duration of the meeting (a few hours)
  - ▶ Focus on the **discovery** of errors, not on fixing them
  - ▶ Participants: designer, moderator, and a secretary
  - ▶ Foster cooperation; no evaluation of people
  - ▶ Experience shows that most errors are discovered by the designer during the presentation, while trying to explain the design to other people
- Forces looking at the code from a different viewpoint
- Can be used for documentation too

# Informal Analysis Techniques Code Inspection

- A reading technique aiming at error discovery
- Based on checklists
  - ▶ Use of uninitialized variables
  - ▶ Jumps into loops
  - ▶ Nonterminating loops
  - ▶ Array indexes out of bounds

# Correctness Proofs

- Formal program analysis is a verification aid that may enhance program reliability
- Mathematically prove that the program's semantics implies its specification
- Can use pre and post conditions
- We can prove correctness of operations (like those on an abstract data type)
- Use the proof of operations to prove fragments that operate on the objects of an ADT
- Tabular expressions can be proven to match between specification of requirements and a specification of the design
- In many cases verification can be automated, at least partially

# Assessment of Correctness Proofs

- Not often used in practice
- However
  - ▶ May be used for very critical portions
  - ▶ Assertions may be the basis for a systematic way of inserting runtime checks
  - ▶ Proofs may become more practical as more powerful support tools are developed
  - ▶ Knowledge of correctness theory helps programmers being rigorous



# Symbolic Execution

- Can be viewed as a middle way between testing and analysis
- Executes the program on symbolic values
- One symbolic execution corresponds to many actual executions

# Model Checking

- Correctness verification, in general, is an undecidable problem
- Model checking is a recent verification technique based on the fact that most interesting system properties become decidable (algorithmically verifiable) when the system is modelled as a finite state machine

# Model Checking Continued

- Describe a given system - software or otherwise - as an FSM
- Express a given property of interest as a suitable formula
  - ▶ Does a computation exist that allows a process to enter a critical region?
  - ▶ Is there a guarantee that a process can access shared resources?
- Verify whether the system's behaviour does indeed satisfy the desired property
  - ▶ This step can be performed automatically
  - ▶ The model checker either provides a **proof** that the property holds or gives a **counter example** in the form of a test case that exposes the system's failure to behave according to the property

# Why so Many Approaches to Testing and Analysis?

- Testing versus (correctness) analysis
- Formal versus informal techniques
- White-box versus black-box techniques
- Techniques in the small/large
- Fully automatic versus semi-automatic techniques (for undecidable problems)
- ...

View all of these as complementary

# Debugging

What approaches do you use for debugging?

# Debugging

- The activity of locating and correcting errors
- It can start once a failure has been detected
- The goal is closing the gap between a fault and a failure
  - ▶ Memory dumps, watch points
  - ▶ Intermediate assertions can help
  - ▶ Tools like gdb, valgrind, etc.
- Incremental integration tests helps
- Incrementally add complexity to test cases
- Like investigating an experiment - one controlled variable at a time

# Verifying Performance

How might you measure/assess performance?