ST Week 10 Notes

L1: Basic OO Integration Concepts

- Several object-oriented concepts and language features are relevant for integration testing of object-oriented languages.
- Some of them apply at the inter-method level (between methods in the same class) and some at the inter-class level (between methods in different classes).
- Features that we will re-cap (with testing in mind):
  - Encapsulation
  - Class inheritance
  - Method overriding
  - Variable hiding
  - Class constructors
  - Polymorphism
  - Overloading

- Encapsulation is an abstraction mechanism to enforce information hiding. It frees clients of an abstraction from unnecessary dependence on design decisions in the implementation of the abstraction.
- Objects can restrict access to their member variables and methods by other objects using encapsulation.
- Java supports 4 such access levels: private, protected, public and default (also called package).
- Default access is package.

Specifier	Same class	Different class same package	Different package subclass	Different package non-subclass
private	Y	n	n	n
package	Y	Y	n	n
protected	Y	Y	Y	n
public	Y	Y	Y	Y

- A subclass inherits variables and methods from its parent and all of its ancestors. Subclass can then
  - use them as defined, or
  - override the methods, or
  - hide the variables.
- Java does not support multiple class inheritance, so every class has only one immediate parent.
- Sub-classes can also explicitly use their parent's variables and methods using the keyword `super` (`super.methodname();`).
- Java's inheritance allows method overriding, variable hiding and class constructors.
- Method overriding allows a method in a subclass to have the same name, arguments and result type as a method in its parent.
- Overriding allows sub-classes to re-define inherited methods.
  - The child class method has the same signature, but a different implementation.
- Variable hiding is achieved by defining a variable in a child class that has the same name and type of an inherited variable.
- This has the effect of hiding the inherited variable from the child class.
- A constructor in a class is a special type of subroutine called to create an object.
- It prepares the new object for use, often accepting arguments that the constructor uses to set required member variables.
- They are not inherited in the same way methods are.
- To use a constructor, we must explicitly call it using the `super` keyword. The call must be the first statement in the derived class constructor and the parameter list must match the parameters in the argument list of the parent constructor.
- Java supports two versions of polymorphism: attributes and methods.
- They both use dynamic binding.
- Each object has a *declared type* and an *actual type*.
  - The actual type can be the declared type or any type that is descended from the declared type.
- A polymorphic attribute is an object reference that can take on various types.
- At any location in the program, the type of the object reference can be different in different executions.
- A polymorphic method can accept parameters of different types by having a parameter that is declared of type `Object`.
- Polymorphic methods are used to implement type abstraction.
- Overloading is the use of the same name for different constructors or methods in the same class.
- They must have different signatures, or lists of arguments.
- Note: Overloading is different from overriding.
  - The former occurs with two methods in the same class, whereas overriding occurs between a class and one of its descendants.

- Instance and class variables are associated with a class.
- Class variables are also called static, in the sense that there is only one copy of the variable that is shared with all instances of that class.
- Instance variables belong to an object (an instance of a class). Every instance of that class has its own copy of the variable. Changes made to the instance variable don't reflect in other instances of that class.
- Levels:
  - Instance variables are declared at class level.
  - Class variables are declared with `static`.
  - Local variables are declared within methods.

L2: Mutation Operators OO Integration

- Mutation operators are available for integration testing of several OO features:
  - Information hiding language features
  - Inheritance
  - Polymorphism and dynamic binding
  - Method overloading, and
  - classes.
- Access Modifier Change (AMC): The access level for each instance variable and method is changed to other access levels.
  - The AMC operator helps to generate tests that ensure that accessibility is correct.
  - AMC created mutants can be killed only if the new access level denies access to another class or allows access that causes a name conflict.
- Hiding Variable Deletion (HVD): Each declaration of an overriding, or hiding variable is deleted.
  - This causes references to that variable to access the variable defined in the parent (or ancestor), which is a common programming mistake.
- Hiding Variable Insertion (HVD): A declaration is added to hide the declaration of each variable declared in an ancestor.
  - Such mutants can be killed only by test cases that can show that the reference to the overriding variable is incorrect.
- Overriding Method Deletion (OMD): Each entire declaration of an overriding method is deleted.
  - References to the method will then use the parent's version.
  - This ensures that the method invocation is to the intended method.
- Overriding Method Moving (OMM): Each call to an overridden method is moved to the first and last statements of the method and up and down one statement.
  - Overriding methods in child classes often call the original method in the parent class, for e.g., to modify a variable that is private to the parent.
  - A common mistake is to make a call to parent's version at the wrong time. This operator will catch this error.
- Overridden Method Rename (OMR): Renames the parent's versions of methods that are overridden in a sub-class so that the overriding does not affect the parent's method.
- This operator is designed to check if an overridden method causes problems with other methods. For e.g.
  - Two classes: `List` and child class `Stack`.
  - Two methods: `m()` and `f()`, both in class `List`, `m()` calls `f()`.
  - `m()` is inherited without change in `Stack`, but `f()` is overridden in `Stack`.
  - When `m()` is called on an object of type `Stack`, it calls the latter's version `f()` instead of `List`'s version. This may have unintended consequences.
- Super Keyword Deletion (SKD): Delete each occurrence of the `super` keyword.
  - After the change, the reference will be to the local version instead of the ancestor's version.
  - The SKD operator is designed to ensure that hiding/hidden variables and overriding/overridden are used appropriately.
- Parent Constructor Deletion (PCD): Each call to a `super` constructor is deleted.
  - The parent's or ancestor's default constructor will be used.
  - To kill these mutants, we need to find a test case for which the parent's default constructor creates an initial state that is incorrect.
- Actual Type Change (ATC): The actual type of a new object is changed in the `new()` statement.
  - This causes the object reference to refer to an object of a type that is different from the original actual type.
  - The new actual type must be in the same "type family" of the original actual type.

- Declared/Parameter Type Change (DTC/PTC): The declared type of each new object/each parameter object is changed in the declaration.
  - The new declared type must be an ancestor of the original type.
  - The instantiation will still be valid.
  - To kill these mutants, a test case must cause the behavior of the object to be incorrect with the new declared type.
- Reference Type Change (RTC): This right side objects of assignment statements are changed to refer to objects of a compatible type.
  - For e.g., if an `Integer` is assigned to a reference of type `Object`, the assignment may be changed to that of a `String`.
  - Since both are descended from `Object`, both can be assigned interchangeably.
- Overloading Method Change (OMC): For each pair of methods that have the same name, the bodies are interchanged.
  - This ensures that overloaded methods are invoked appropriately.
- Overloading Method Deletion (OMD): Each overloaded method declaration is deleted, one at a time.
  - This operator ensures coverage of overloaded methods— all of them must be invoked at least once.
  - If the mutant still works correctly without the deleted method, there may be an error in invoking one of the overloading methods— the incorrect method may be invoked or an incorrect parameter type conversion has occurred.
- Argument Order Change (AOC): The order of the arguments in method invocations is changed to be the same as that of another overloading method, if one exists.
  - This causes a different method to be called, this checking for a common fault in the use of overloading.
- Argument Number Change (ANC): The number of the arguments in method invocations is changed to be the same as that of another overloading method, if one exists.
  - This helps ensure that the programmer did not invoke the wrong method.
  - When new values need to be added, they are the constant default values of primitive types or the result of the default of constructors for objects.
- `this` Keyword Deletion (TKD): Each occurrence of the keyword `this` is deleted.
  - Within a method body, uses of the keyword `this` refers to the current object if the member variable is hidden by a local variable or method parameter that has the same name.
  - This operator checks if the member variables are used correctly by replacing occurrences of `this.X` with `X`.
- Static Modifier Change (SMC): Each instance of the `static` modifier is removed, and the `static` modifier is added to instance variables.
  - This operator validates use of instance and class variables.
- Variable Initialization Deletion (VID): Remove initialization of each member variable.
  - Instance variables can be initialized in the variable declaration and in constructors for the class.
  - This operators removes the initializations so that member variables are initialized to the default values.
- Default Constructor Deletion (DCD): Delete each declaration of default constructor (with no parameters).
  - This ensures that user-defined default constructors are implemented properly.

L3: OO Integration Testing

OO languages use

- Classes to represent data abstraction.
- In addition inheritance, polymorphism and dynamic binding support abstraction.
- New type created by inheritance are descendents of the existing type.
- Extension and refinement:
  - A class extends its parent class if it introduces a new method name and does not override any methods in an ancestor class.
  - A class refines the parent class if it provides new behaviour not present in the overridden method, does not call the overridden method and its behaviour is semantically consistent with that of the overridden method.

Two types of inheritance are used:

- Sub-type inheritance: Famously called the substitution principle. If class `B` uses sub-type inheritance from class `A`, it is possible to freely substitute any instance of `B` for `A` and still satisfy an arbitrary client of `A`. `B` has an *is-a* relationship with `A`.
- Sub-class inheritance: This allows descendant classes to reuse methods and variables from ancestor classes without necessarily ensuring that instances of the descendants meet the specifications of the anscestor type.

If class `B` inherits from class `A` and both `A` and `B` define a method `m()` then, `m()` is called a polymorphic method.

If an object `x` is *declared* to be of type `A` then during execution `x` can have either the *actual* type `A` or `B`.

- The version that is executed depends on the *current* actual type of the object.

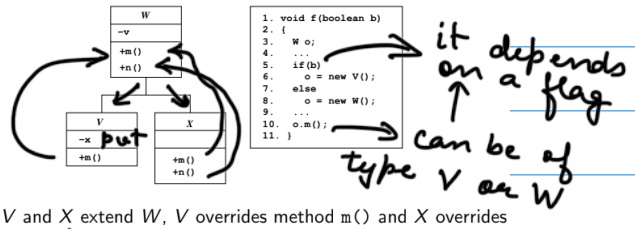
The collection of methods that can be executed is called the polymorphic call set. For this e.g., it is `{A:m(), B:m()}`.

With respect to testing that is unique to OO software, a class is usually regarded as the basic unit of testing.
There are four levels of testing classes.

- Intra-method testing: Tests are constructed for individual methods (traditional unit testing).
- Inter-method testing: Multiple methods within a class are tested in concert (traditional module testing).
- Intra-class testing: Tests are constructed for a single class, usually as sequences of calls to methods within the class.
- Inter-class testing: More than one class is tested at the same time, usually to see how they interact (kind of integration testing).

similar

- We assume that a class encapsulates state information in a collection of *state variables*. The behaviors of a class are implemented by methods that use the state variables.
- The interactions between the various classes and methods within/outside a class that occur in the presence of inheritance, polymorphisn and dynamic binding are complex and difficult to *visualize*.
- We will go through some examples to illustrate the issues.



- V and X extend W, V overrides method `m()` and X overrides methods `m()` and `n()`.
- $-$: attributes are private, $+$: attributes are non-private.
- The declared type of o is W, but at line 10, the actual type can be either V or W.
- Since V overrides `m()`, which version of `m()` is executed depends on the input flag to the method.
- The root class A has four state variables and six methods, two descendents B and C. Methods of A are called in sequence.
- The state variables of A are protected, i.e., available to B and C.
- B declares one state variable and three methods, C declares three methods.
- `B::h()` overrides `A::h()`, `B::i()` overrides `A::i()`, `C::i()` overrides `B::i()`, `C::j()` overrides `A::j()` and `C::l()` overrides `A::l()`.
- Data flow anamoly: Suppose an instance of B is bound to an object o and a call to `d()` is made. B's version of `h` and `i` are called, `A::u` and `A::w` are not given values, and thus the call to `A::j` can result in a data flow anamoly.

Understanding which version of a method *will* be executed and which versions *can* be executed is very difficult.

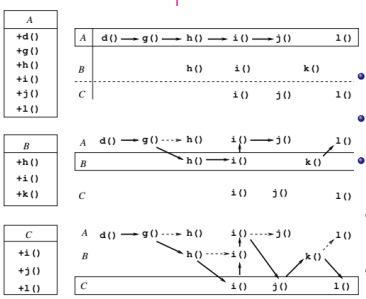
Execution can *bounce* up and down among levels of inheritance.

The yo-yo graph is defined on an inheritance hierarchy.

- It has a root and descendants.
- Nodes are methods: new, inherited and overridden methods for each descendent.
- Edges are method calls as given in the source: directed edge is from caller to callee.

In addition,

- Each class is given a *level* in the yo-yo graph that shows the actual calls made if an object has the actual type of that level. These are depicted by *bold arrows*.
- *Dashed arrows* are calls that cannot be made due to overriding.



- A's implementation: `d()` calls `g()`, `g()` calls `h()`, `h()` calls `i()` and `i()` calls `j()`.
- B's implementation: `h()` calls `i()`, `i()` calls its parent's (A's) version of `i()` and `k()` calls `l()`.
- C's implementation: `i()` calls its parent's (B's) version of `i()` and `j()` calls `k()`.
- Top level: A call is made to method `d()` through an object of actual type A.
  - This sequence of calls is simple and straightforward.
- Second level: Object is of actual type B. When `g()` calls `h()`, the version of `h()` defined in B is executed. The control then continues to `i()` in B, `i()` in A and then to `j()` in A.
- Third level: Object is of actual type C. Control proceeds from `g()` in A, to `h()` in B to `i()` in C, then, to `i()` in A and B etc— exhibiting a *yo-yo* effect.

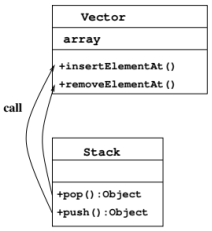
L4: OO Faults

- Complexity is relocated to the connections among components.
- Less *static determinism*; many faults can now only be detected at runtime.
- Inheritance and Polymorphism yield *vertical* and *dynamic* integration.
- Aggregation and use relationships are more complex.
- Designers do not carefully consider visibility of data and methods.
- Faults related to inheritance, polymorphism, constructors and visibility are listed.
- Assumption: Anamoly/fault is manifested through polymorphism in a context that uses an instance of the ancestor. i.e., instances of descendant classes can be substituted for instances of the ancestor.
- Faults described are *language independent*.

Acronym	Fault/Anomaly
ITU	Inconsistent type use
SDA	State definition anomaly
SDIH	State definition inconsistency
SDI	State defined incorrectly
IISD	Indirect inconsistent state definition
ACB1	Anomalous construction behaviour (1)
ACB2	Anomalous construction behaviour (2)
IC	Incomplete construction
SVA	State visibility anomaly

ITU

- A descendent class does not override any inherited method— hence, no polymorphic behaviour.
- Class C extends class T, and C adds new methods (extension).
- An object is used "as a C", then as a T, then as a C.
- Methods in T can put object in state that is inconsistent for C.
- Class `Vector` is a sequential data structure that supports direct access to its elements.
- Class `Stack` uses methods inherited from `Vector` to implement the stack.

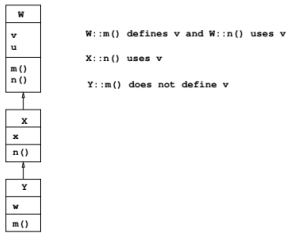



```
s.push("string1");
s.push("string2");
s.push("string3");
dumb(s);
pop();
pop();
pop(); //Stack is empty

void dumb(Vector v)
{
  v.removeElementAt(v.size()-1);
}
```

SDA

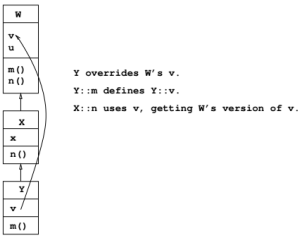
- The state interactions of a descendant are not consistent with those of its ancestor.
- The refining methods fail to define some variables and hence required definitions might not be available.
- For e.g., let us say that a class `X` extends class `W` and `X` overrides some methods of `W`.
- The overriding methods in `X` fail to define some variables that the overridden methods in `W` defined.



SDIH

For an object of actual type `Y`, a data flow anomaly exists and results in a fault if `m()` is called, then `n()`.

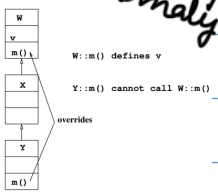
- A local variable is introduced to a class definition and the name of the variable is the same as an inherited variable `v`.
  - The inherited variable is hidden from the scope of the descendent (unless explicitly qualified, as in `super.v`).
- A reference to `v` will refer to the descendent's `v`.
- Anomaly exists if a method that normally defines the inherited `v` is overridden in a descendant when an inherited state variable is hidden by a local definition.



SVA

For an object of actual type `Y`, a data flow anomaly exists and results in a fault if `m()` is called, then `n()`.

- Consider a class `W`, which is an ancestor `X` and `Y`: `X` extends `W` and `Y` extends `X`.
- `W` declares a private variable `v`, `v` is defined by `W::m`.
- `Y` overrides `m()`, and calls `W::m()` to define `v`.
- This causes a data flow anomaly as `v` is private for `W`.
- State defined incorrectly: An overriding method defines the same state variable that the overridden method defines. In the absence of identical computations by the two methods, this could result in a behavior anomaly.
- Indirect inconsistent state definition fault: A descendent adds an extension method that defines an inherited state variable.



L5: Coupling Criteria

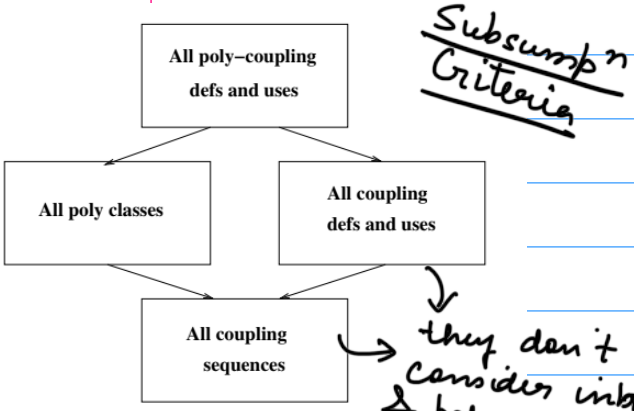
- Coupling variables: Variables defined in one unit and used in another unit.
- Last-def: The set of nodes that define a variable x and has a def-clear path from the node through a call site to a use in the other module.
  - Can be from caller to callee (parameter or shared variable) or from callee to caller (return value).
- First-use: The set of nodes that have uses of a variable y and for which there is a def-clear and use-clear path from the call site to the nodes.
- Coupling du-path: A du-path from a last definition to a first use.
- Pairs of method calls within body of method under test.
  - Made through a common *instance context*.
  - With respect to a set of state variables that are commonly referenced by both methods.
  - Consists of at least one coupling path between the two method calls with respect to a particular state variable.
- Represent potential *state space interactions* between the called methods with respect to calling method.
- Used to identify points of integration and testing requirements.

In the definitions below, `o` is an identifier whose type is a *reference to an instance* of an object pointing to a memory location that contains an instance (value) of some type.

- A reference `o` can refer to instances whose actual instantiated types are either the base type of `o` or a descendent of `o`'s type.
  - For e.g., in Java, `A o = new B();`, `o`'s base or declared type is `A` and its instantiated or actual type is `B`.
- `o` is considered to be defined when one of the state variables `v` of `o` is defined.

- Definitions and uses in OO-applications for coupling variables can be indirect.
  - Indirect definitions and uses occur when a method defines/references a particular value `v`.
- In the presence of indirect definitions and uses, we have to consider all the methods that can potentially execute.
- Polymorphic call set or Satisfying set: Set of methods that can potentially execute as result of a method call through a particular instance context.
- Coupling sequence: A pair of nodes that call two methods in sequence. The first method defines a variable, the second method uses it.
- The calling method is the coupling method `f()`, it calls `m()`, the antecedent method, to define a variable, and `n()`, the consequent method, to use the variable.
- Transmission set: Variables for which a path is def-clear.
- In the figure, the path from `h` to `i` to `k` and finally to `l` forms a transmission path with respect to `o.v`. The object `o` is the context variable.
- $S_{j,k}$: Coupling sequence.
- $\Theta^{t}_{S_{j,k}}$: Coupling set of $S_{j,k}$ is the intersection of the variables defined by `m()`, used by `n()`, through the instance context provided by a context variable `o` that is bound to an instance of `t`.
- As usual, the versions of `m()` and `n()` that run are determined by the actual type `t` of the instance bound to `o`.

- All-Coupling-Sequences (ACS): For every coupling sequence S_j in $f()$, there is at least one test case t such that there is a coupling path induced by $S_{j,k}$ that is a sub-path of the execution trace of $f(t)$.
  - At least one coupling path must be executed.
  - This does not consider inheritance and polymorphism.
- All-Poly-Classes (APC): For every coupling sequence $S_{j,k}$ in method $f()$, and for every class in the family of types defined by the context of $S_{j,k}$, there is at least one test case t such that when $f()$ is executed using t, there is a path p in the set of coupling paths of $S_{j,k}$ that is a sub-path of the execution trace of $f(t)$.
  - Includes instance contexts of calls.
  - At least one test for every type the object can bind to.
  - Test with every possible type substitution.
- All-Coupling-Defs-Uses (ACDU): For every coupling variable v in each coupling $S_{j,k}$ of t, there is a coupling path induced by $S_{j,k}$ such that p is a sub-path of the execution trace of $f(t)$ for at least one test case t.
  - Every last definition of a coupling variable reaches every first use.
  - Does not consider inheritance and polymorphism.
- All-Poly-Coupling-Defs-and-Uses (APDU): For every coupling sequence $S_{j,k}$ in $f()$, for every class in the family of types defined by the context of $S_{j,k}$, for every coupling variable v of $S_{j,k}$, for every node m that has a last definition of v and every node n that has a first-use of v, there is at least one test case t such that when $f()$ is executed using t, there is a path p in the coupling paths of $S_{j,k}$ that is a sub-path of the trace of $f()$.
  - Every last definition of a coupling variable reaches every first use for every type binding.
  - Combines previous criteria.
  - Handles inheritance and polymorphism.
  - Takes definitions and uses of variables into account.



Subsumption Criteria

they don't consider inheritance & polymorphism