Testing Object Oriented Software

Dr. Paul West

Department of Computer Science
College of Charleston

March 20, 2014

Combinatorial Explosion

```
 abstract class Credit {
...
   abstract boolean validateCredit( Account a, int amt, CreditCard c);
...
}
```

- Credit could have many implementations:EduCredit, BizCredit, IndividualCredit
- Account could have many implementations: USAccount, UKAccount, EUAccount, JPAccount, OtherAccount
- CreditCard could have many implementations: VISACard, AmExpCard, StoreCard
- That is $3x5x3 = 45$ possible combinations!
- Are all required for effective testing?

The Combinatorial approach

- Identify a set of combinations that cover all pairwise combinations of dynamic bindings.
- EX: USAccount & UKAccount *may* be treated the same.

Combined calls: undesired effects

```
public abstract class Account { ...
    public int getYTDPurchased() {
if (ytdPurchasedValid) { return ytdPurchased; }
int totalPurchased = 0;
for (Enumeration e = subsidiaries.elements() ; e.hasMoreElements(); )
    {   Account subsidiary = (Account) e.nextElement();
totalPurchased += subsidiary.getYTDPurchased();
    }
for (Enumeration e = customers.elements(); e.hasMoreElements(); )
    {   Customer aCust = (Customer) e.nextElement();
totalPurchased += aCust.getYearlyPurchase();
    }
ytdPurchased = totalPurchased;
ytdPurchasedValid = true;
return totalPurchased;
}   }
```

Problem: different implementations of methods getYDTPurchased refer to different currencies.

A data flow approach

- identify polymorphic calls, binding sets, defs and uses
- Where is:
    - totalPurchased used
    - defined
    - totalPurchased used and defined

Def-Use (dataflow) testing of polymorphic calls

- Derive a test case for each possible polymorphic <def,use> pair
  - Each binding must be considered individually
  - Pairwise Combinatorial selection may help in reducing the set of test cases
- Example: Dynamic binding of currency
  - We need test cases that bind the different calls to different methods in the same run
  - We can reveal faults due to the use of different currencies in different methods

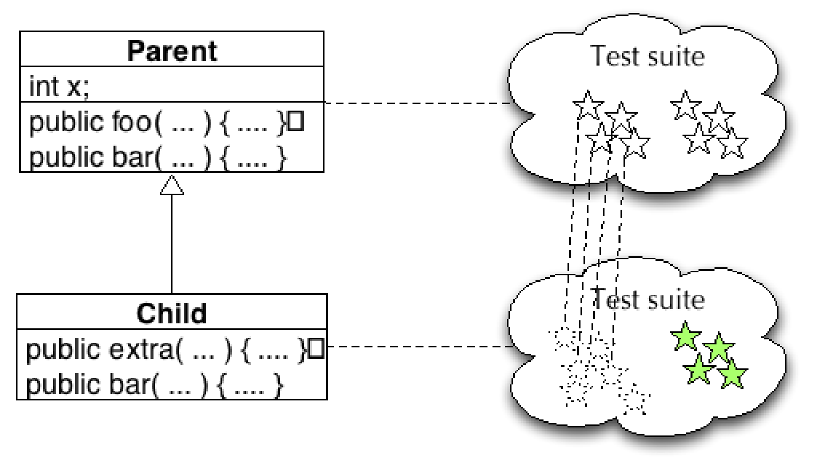
Inheritance

- When testing a subclass ...
    - We would like to re-test only what has not been thoroughly tested in the parent class
        - for example, no need to test hashCode and getClass methods inherited from class Object in Java
    - But we should test any method whose behavior may have changed
        - even accidentally!

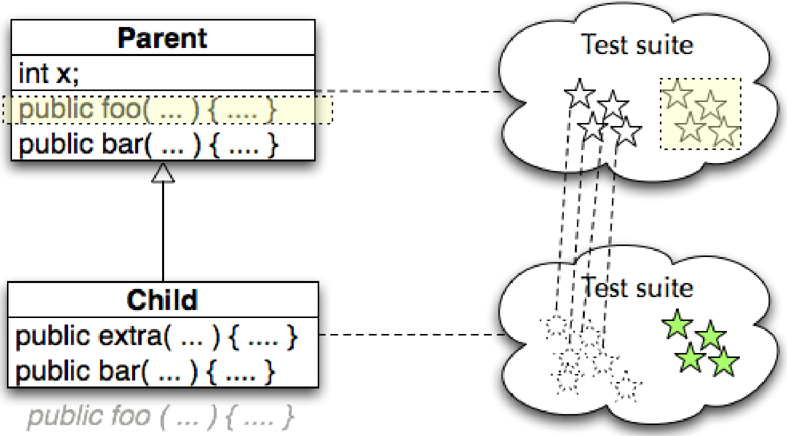
Reusing Tests with the Testing History Approach

- Track test suites and test executions
  - determine which new tests are needed
  - determine which old tests must be re-executed
- New and changed behavior ...
  - new methods must be tested
  - redefined methods must be tested, but we can partially reuse test suites defined for the ancestor
  - other inherited methods do not have to be retested

Testing History

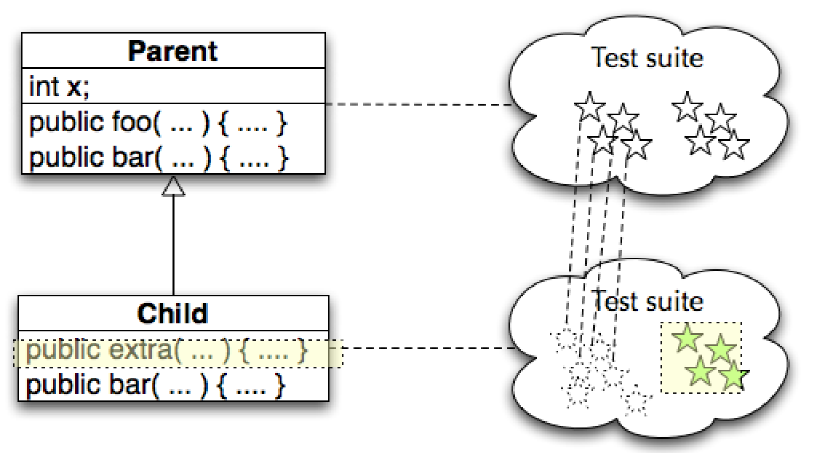

Parent
int x;
public foo(...) { .... }
public bar(...) { .... }
Child
public extra(...) { .... }
public bar(...) { .... }
Test suite
Test suite

Inherited, unchanged


Parent
int x;
public foo(...) { .... }
public bar(...) { .... }
Child
public extra(...) { .... }
public bar(...) { .... }
public foo (...) { .... }
Test suite
Test suite

Inherited, unchanged ("recursive"):
No need to re-test

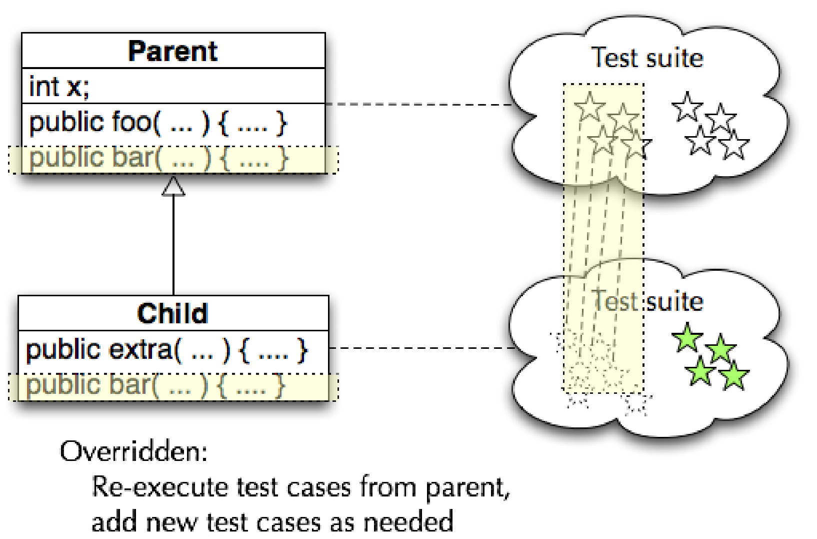
Newly introduced methods


Parent
int x;
public foo(...) { .... }
public bar(...) { .... }
Child
public extra(...) { .... }
public bar(...) { .... }
Test suite
Test suite

New:

Design and execute new test cases

Overridden methods


Parent
int x;
public foo(...) { .... }
public bar(...) { .... }
Child
public extra(...) { .... }
public bar(...) { .... }
Test suite
Test suite

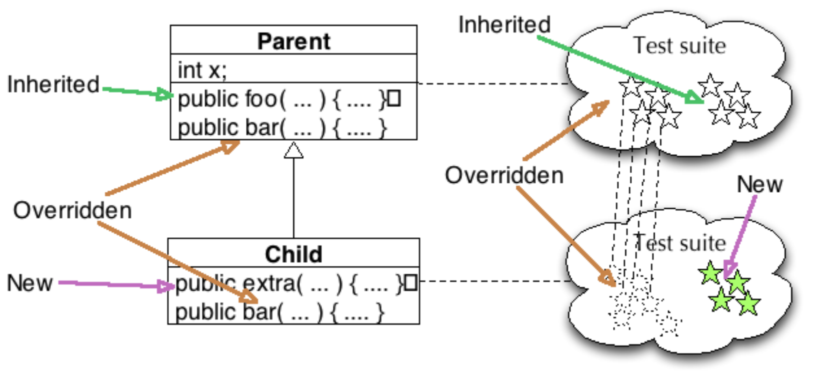
Overridden:

Re-execute test cases from parent, add new test cases as needed

Testing History - some details

- Abstract methods (and classes)
  - Design test cases when abstract method is introduced (even if it can’t be executed yet)
- Behavior changes
  - Should we consider a method “redefined” if another new or redefined method changes its behavior?
    - The standard “testing history” approach does not do this
    - It might be reasonable combination of data flow (structural) OO testing with the (functional) testing history approach

Testing History - Summary


Parent
int x;
public foo(...) { .... }
public bar(...) { .... }
Child
public extra(...) { .... }
public bar(...) { .... }
Inherited
Overridden
New
Inherited
Test suite
Overridden
New
Test suite

Does testing history help?

- Executing test cases should (usually) be cheap
  - It may be simpler to re-execute the full test suite of the parent class
  - ... but still add to it for the same reasons
- But sometimes execution is not cheap ...
  - Example: Control of physical devices
  - Or very large test suites
    - Ex: Some Microsoft product test suites require more than one night (so daily build cannot be fully tested)
  - Then some use of testing history is profitable

Testing generic classes

a generic class
class PriorityQueue<Elem Implements Comparable> {...}
is designed to be instantiated with many different parameter types
PriorityQueue<Customers>
PriorityQueue<Tasks>

A generic class is typically designed to behave consistently some set of permitted parameter

Testing can be broken into two parts
Showing that some instantiation is correct
showing that all permitted instantiations behave consistently

Show that some instantiation is correct

- Design tests as if the parameter were copied textually into the body of the generic class.
  - We need source code for both the generic class and the parameter class

Identify (possible) interactions

- Identify potential interactions between generic and its parameters
  - Identify potential interactions by inspection or analysis, not testing
  - Look for: method calls on parameter object, access to parameter fields, possible indirect dependence
  - Easy case is no interactions at all (e.g., a simple container class)
- Where interactions are possible, they will need to be tested

Example interaction

class PriorityQueue <Elem implements Comparable> ...

- Priority queue uses the “Comparable” interface of Elem to make method calls on the generic parameter
- We need to establish that it does so consistently
    - So that if priority queue works for one kind of Comparable element, we can have some confidence it does so for others

Testing variation in instantiation

- We can’t test every possible instantiation
  - Just as we can’t test every possible program input
- ... but there is a contract (a specification) between the generic class and its parameters
  - Example: “implements Comparable” is a specification of possible instantiations
  - Other contracts may be written only as comments
- Functional (specification-based) testing techniques are appropriate
  - Identify and then systematically test properties implied by the specification

Example: Testing instantiation variation

```
Most but not all classes that implement Comparable also satisfy the rule
(x.compareTo(y) == 0) == (x.equals(y))
(from java.lang.Comparable)
```

So test cases for PriorityQueue should include

- instantiations with classes that do obey this rule:
  class String
- instantiations that violate the rule:
  class BigDecimal with values 4.0 and 4.00

Exception handling

```
void addCustomer(Customer theCust) {
customers.add(theCust);
    }
    public static Account
newAccount(...)
throws InvalidRegionException
    {
Account thisAccount = null;
String regionAbbrev = Regions.regionOfCountry(
      mailAddress.getCountry());
if (regionAbbrev == Regions.US) {
    thisAccount = new USAccount();
} else if (regionAbbrev == Regions.UK) {
    ....
} else if (regionAbbrev == Regions.Invalid) {
    throw new InvalidRegionException(mailAddress.getCountry());
}
...
    }
```

- exceptions create implicit control flows and may be handled by different handlers

Testing exception handling

- Impractical to treat exceptions like normal flow
  - too many flows: every array subscript reference, every memory allocation, every cast, ...
  - multiplied by matching them to every handler that could appear immediately above them on the call stack.
  - many actually impossible
- So we separate testing exceptions
  - and ignore program error exceptions (test to prevent them, not to handle them)
- What we do test: Each exception handler, and each explicit throw or re-throw of an exception

Testing program exception handlers

- Local exception handlers
  - test the exception handler (consider a subset of points bound to the handler)
- Non-local exception handlers
  - Difficult to determine all pairings of <points, handlers>
  - So enforce (and test for) a design rule: if a method propagates an exception, the method call should have no other effect

Summary

- Several features of object-oriented languages and programs impact testing
  - from encapsulation and state-dependent structure to generics and exceptions
  - but only at unit and subsystem levels
  - and fundamental principles are still applicable
- Basic approach is orthogonal
  - Techniques for each major issue (e.g., exception handling, generics, inheritance, ...) can be applied incrementally and independently

- `books.cat-v.org/computer-science/mythical-man-month/tmmm.pdf`
- Skim this book, we will do an overview.
- Also, think of a chapter you wish to present–because you will.

- Pick 1 problem from Chapter 15 (I recommend 15.1) and turn into the dropbox.