School of
Science

AUSTRALIA ECU EDITH COWAN UNIVERSITY

Module 7: OO - Testing recap

CSI3105:
Software Testing

Unit? Integration? System?

Unit? Integration? System?

My software!!!

Test wheel and axle together unit

Unit? Integration? System?

My software!!!

Test it all

Characteristics of OO Software

Typical OO software characteristics that impact testing

- State dependent behavior
- Encapsulation
- Inheritance
- Polymorphism and dynamic binding
- Abstract and generic classes
- Exception handling

State dependent behavior

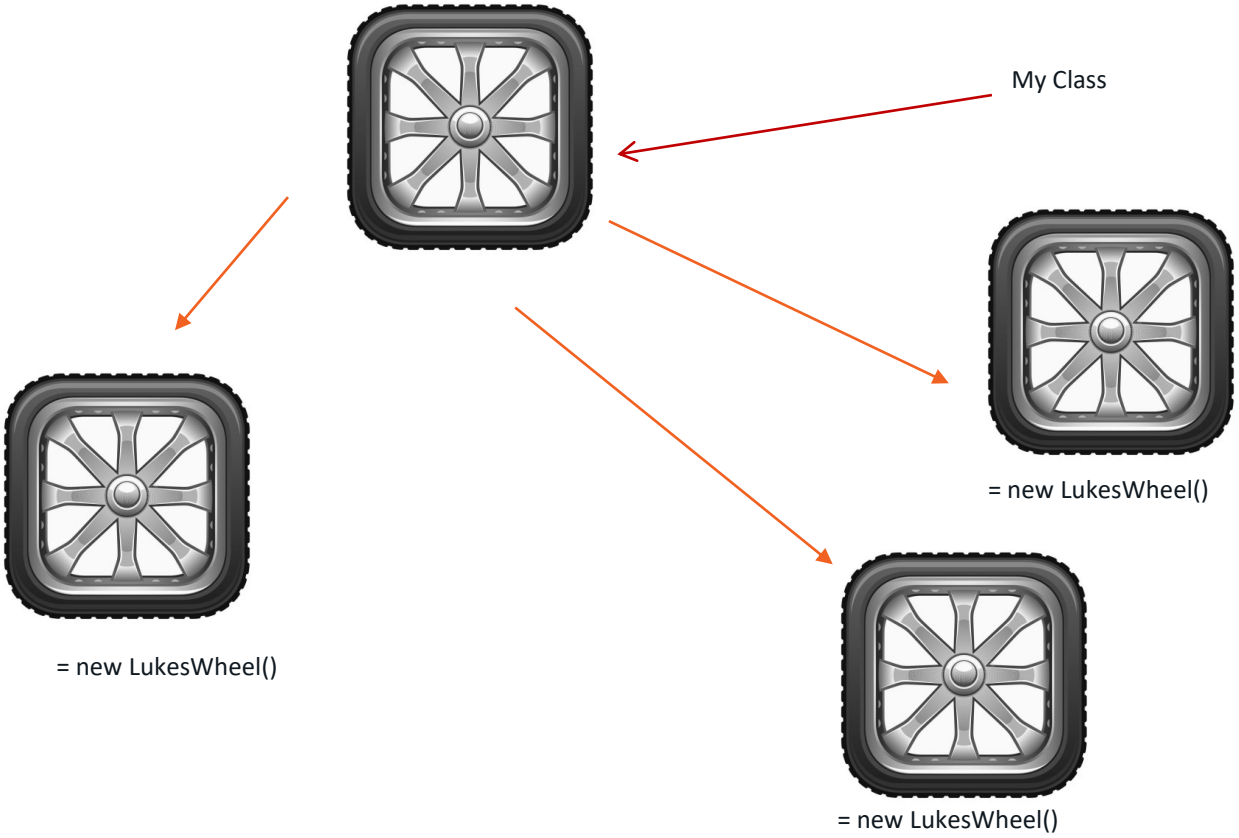
- State-dependent behaviour in object-oriented programming can significantly impact software testing
- In object-oriented programming, an object's behaviour depends not only on its internal implementation but also on its current state.
    - Variables associated with a class change when methods are called
- Different inputs or events can cause the object to transition to a different state, which may result in different behaviour

Objects

My Class

- Class… it’s a thing
  - Object
  - Entity
  - Thing
- Store's data
  - Properties/ Fields
- Allows us to manipulate that data
  - Operations/ Methods

Objects as Template



Objects as Template

My Class

= new LukesWheel()

W1.inflated = true
W1.spokes = 8

W1= new LukesWheel()

= new LukesWheel()

Objects State

My Class

W1.inflated = true
W1.spokes = 8

State of the object

W1= new LukesWheel()

Objects State

My Class

New state of the object

W1.inflated = true
W1.spokes = 8

W1.inflated = false
W1.spokes = 8

W1.pop()

Encapsulation

- Encapsulating or hiding variables with the private keyword, adds in some complexity
- We need to consider how to handle hidden or encapsulated fields.

Accessing the state

Intrusive approaches

- use language constructs (C++ friend classes)
- add inspector methods
- *in both cases we break encapsulation and we may produce undesired results*

Equivalent scenarios approach:

- generate equivalent and non-equivalent sequences of method invocations
- compare the final state of the object after equivalent and non-equivalent sequences

Inheritance

- Inheritance
  - One class can inherit properties and behaviours from another class
  - Any issues in the base class can propagate to the derived classes.

- Testing of derived classes
  - When a class inherits from a base class, it can add or modify behaviours.
  - Test the derived classes to ensure that they behave as intended (if changed or used in a different context)

- Refactoring
  - Refactoring a base class can affect all derived classes, making it necessary to retest them all.

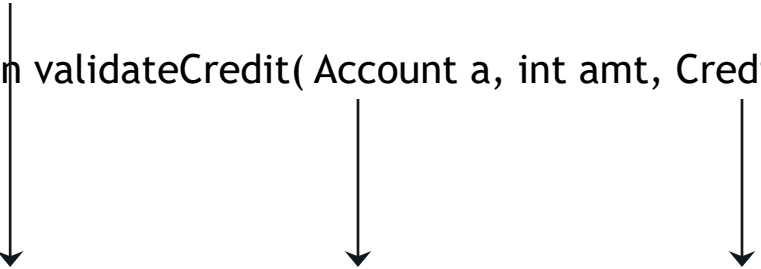
Polymorphism and dynamic binding

- Polymorphism
  - Ability of objects of different classes to be treated as if they were objects of a common superclass
  - Single method can be used to operate on objects of different classes, without needing to know the specific class of each object.

- Dynamic binding
  - A single method call may be dynamically bound to different methods depending on the state of the computation.
  - Overridden method that acts different based on the parameters it receives.
  - Tests must exercise different bindings to reveal failures that depend on a particular binding or on interactions between bindings for different calls.

Polymorphism and dynamic binding

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



EduCredit	USAccount	VISACard
BizCredit	UKAccount	AmExpCard
IndividualCredit	EUAccount	StoreCard
	JPAccount	
	OtherAccount	

The combinatorial problem: 3 x 5 x 3 = 45 possible combinations of dynamic bindings (just for this one method!)

The combinatorial approach

Identify a set of combinations that cover all pairwise combinations of dynamic bindings

Account	Credit	creditCard
USAccount	EduCredit	VISACard
USAccount	BizCredit	AmExpCard
USAccount	individualCredit	ChipmunkCard
UKAccount	EduCredit	AmExpCard
UKAccount	BizCredit	VISACard
UKAccount	individualCredit	ChipmunkCard
EUAccount	EduCredit	ChipmunkCard
EUAccount	BizCredit	AmExpCard
EUAccount	individualCredit	VISACard
JPAccount	EduCredit	VISACard
JPAccount	BizCredit	ChipmunkCard
JPAccount	individualCredit	AmExpCard
OtherAccount	EduCredit	ChipmunkCard
OtherAccount	BizCredit	VISACard
OtherAccount	individualCredit	AmExpCard

Same motivation as pairwise specification-based testing (module 8, next week)

Abstract and generic classes

- Abstract classes cannot be directly instantiated
  - serve as a blueprint for other classes.
  - Testing an abstract class alone is not feasible
  - Therefore, testing the concrete classes that implement the abstract class becomes important, as they provide the full implementation of the abstract class.
- Generic classes are classes/ methods that are parameterized by one or more types
  - Testing generic classes requires testing them with different types of parameters to ensure that they work correctly in all possible scenarios.

Exception handling

Exception handling

- Attempt to handle run time errors
- separate handling of error cases from the primary program logic

Try catch block etc..

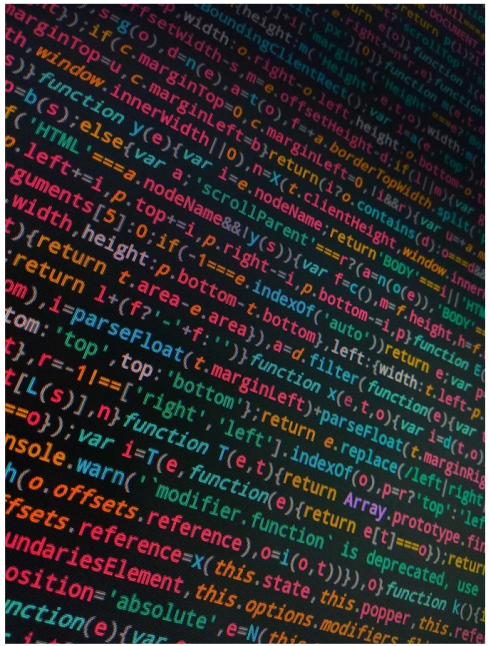
- Alters the flow of code

We need to..

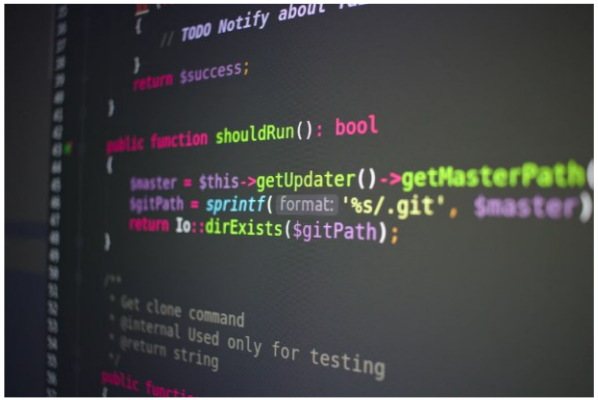
- Test where exceptions are thrown
- Test where exceptions are handled
- Test for unhandled exceptions

CFG - Procedural vs OOP

Differences



Procedural…BIG!

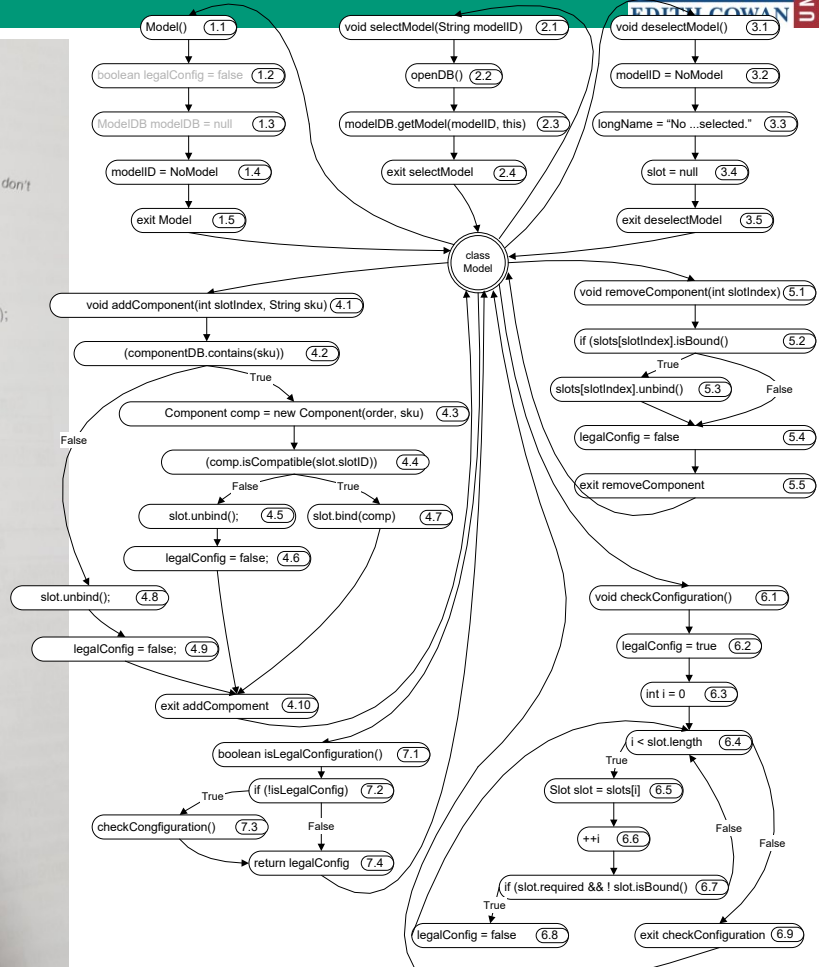


OOP…Small

Control flow graph for a class

```
1   public class Model extends Orders.CompositeItem {
2       public String modelID;   // Database key for slots
3       private int baseWeight; // Weight excluding optional components
4       private int heightCm, widthCm, depthCm; // Dimensions if boxed
5       private Slot[] slots;    // Component slots
6
7       private boolean legalConfig = false; // memoized result of isLegalConf
8       private static final String NoModel = "NO MODEL SELECTED";
12  ...
13      /** Constructor, which should be followed by selectModel */
14      public Model(Orders.Order _order) {
15          super(_order);
16          modelID = NoModel;
17      }
99  ...
100     /** Is the current binding of components to slots a legal
101      * configuration? Memo-ize the result for repeated calls */
102     public boolean isLegalConfiguration() {
103         if (! legalConfig) {
104             checkConfiguration();
105         }
106         return legalConfig;
107     }
108
109     /** Are all required slots filled with compatible components?
110      *  It is impossible to assign an incompatible component,
111      * so just to check that every required slot is filled.    */
112     private void checkConfiguration() {
113         legalConfig = true;
114         for (int i=0; i < slots.length; ++i) {
115             Slot slot = slots[i];
116             if (slot.required && ! slot.isBound()) {
117                 legalConfig = false;
118             }
119         }
120     }
241 ...
242 }
```

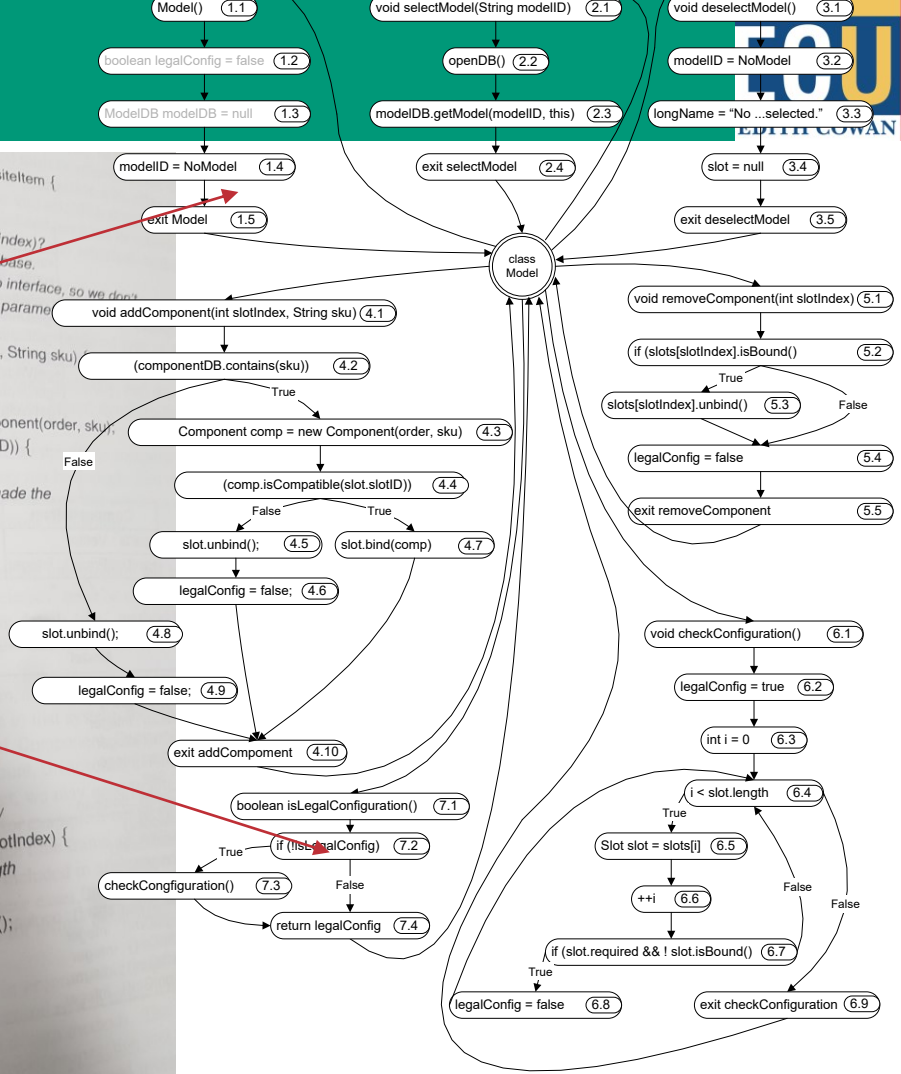
```
1   public class Model extends Orders.CompositeItem {
61  ...
62      /** Bind a component to a slot.
63       * @param slotIndex Which slot (integer index)?
64       * @param sku    Key to component database.
65       * Choices should be constrained by web interface, so we don't
66       * need to be graceful in handling bogus parameters.
67       */
68      public void addComponent(int slotIndex, String sku) {
69          Slot slot =slots[slotIndex];
70          if (componentDB.contains(sku)) {
71              Component comp = new Component(order, sku);
72              if (comp.isCompatible(slot.slotID)) {
73                  slot.bind(comp);
74                  // Note this cannot have made the
75                  // configuration illegal.
76              } else {
77                  slot.unbind();
78                  legalConfig = false;
79              }
80          } else {
81              slot.unbind();
82              legalConfig = false;
83          }
84      }
85
86
87      /** Unbind a component from a slot. */
88      public void removeComponent(int slotIndex) {
89          // assert slotIndex in 0..slots.length
90          if (slots[slotIndex].isBound()) {
91              slots[slotIndex].unbind();
92          }
93          legalConfig = false;
94      }
215 ...
216 }
```



Control flow graph for a class

```
1   public class Model extends Orders.CompositeItem {
2       public String modelID;    // Database key for slots
3       private int baseWeight; // Weight excluding optional components
4       private int heightCm, widthCm, depthCm; // Dimensions if boxed
5       private Slot[] slots;     // Component slots
6
7       private boolean legalConfig = false; // memoized result of isLegalConf
8       private static final String NoModel = "NO MODEL SELECTED";
12  ...
13      /** Constructor, which should be followed by selectModel */
14      public Model(Orders.Order _order) {
15          super(_order);
16          modelID = NoModel;
17      }
99  ...
100     /** Is the current binding of components to slots a legal
101      * configuration? Memo-ize the result for repeated calls */
102     public boolean isLegalConfiguration() {
103         if (! legalConfig) {
104             checkConfiguration();
105         }
106         return legalConfig;
107     }
108
109     /** Are all required slots filled with compatible components?
110      * It is impossible to assign an incompatible component,
111      * so just to check that every required slot is filled.    */
112     private void checkConfiguration() {
113         legalConfig = true;
114         for (int i=0; i < slots.length; ++i) {
115             Slot slot = slots[i];
116             if (slot.required && ! slot.isBound()) {
117                 legalConfig = false;
118             }
119         }
120     }
241 ...
242 }
```

```
1   public class Model extends Orders.CompositeItem {
61  ...
62      /** Bind a component to a slot.
63       * @param slotIndex Which slot (integer index)?
64       * @param sku    Key to component database.
65       * Choices should be constrained by web interface, so we don't
66       * need to be graceful in handling bogus parame
67       */
68      public void addComponent(int slotIndex, String sku)
69          Slot slot =slots[slotIndex];
70          if (componentDB.contains(sku)) {
71              Component comp = new Component(order, sku);
72              if (comp.isCompatible(slot.slotID)) {
73                  slot.bind(comp);
74                  // Note this cannot have made the
75                  // configuration illegal.
76              } else {
77                  slot.unbind();
78                  legalConfig = false;
79              }
80          } else {
81              slot.unbind();
82              legalConfig = false;
83          }
84      }
85
86
87      /** Unbind a component from a slot. */
88      public void removeComponent(int slotIndex) {
89          // assert slotIndex in 0..slots.length
90          if (slots[slotIndex].isBound()) {
91              slots[slotIndex].unbind();
92          }
93          legalConfig = false;
94      }
215 ...
216 }
```



Control flow graph for a class

Control flow for each method

+

node for class

+

edges

- from node *class* to the start nodes of the methods
- from the end nodes of the methods to node *class*

=> control flow through *sequences* of method calls