INB370 / INN370 Software Development Lecture 5 — Maintainability and Metrics

Faculty of Science and Technology

Semester 1, 2010

Queensland University of Technology

Aims of the Week 5 lecture and practical session

- To appreciate the way in which object-oriented program design affects code maintainability
  - Modularity and program maintenance in general
  - How object-oriented program design aids maintenance
  - Things to do (and not to do) when developing object-oriented programs, to aid future maintenance
- To look at some software metrics that can help assess the maintainability of code
  - Coupling and cohesion measures
  - An Eclipse plug-in module for metrics

Part A — Modularity and maintainability

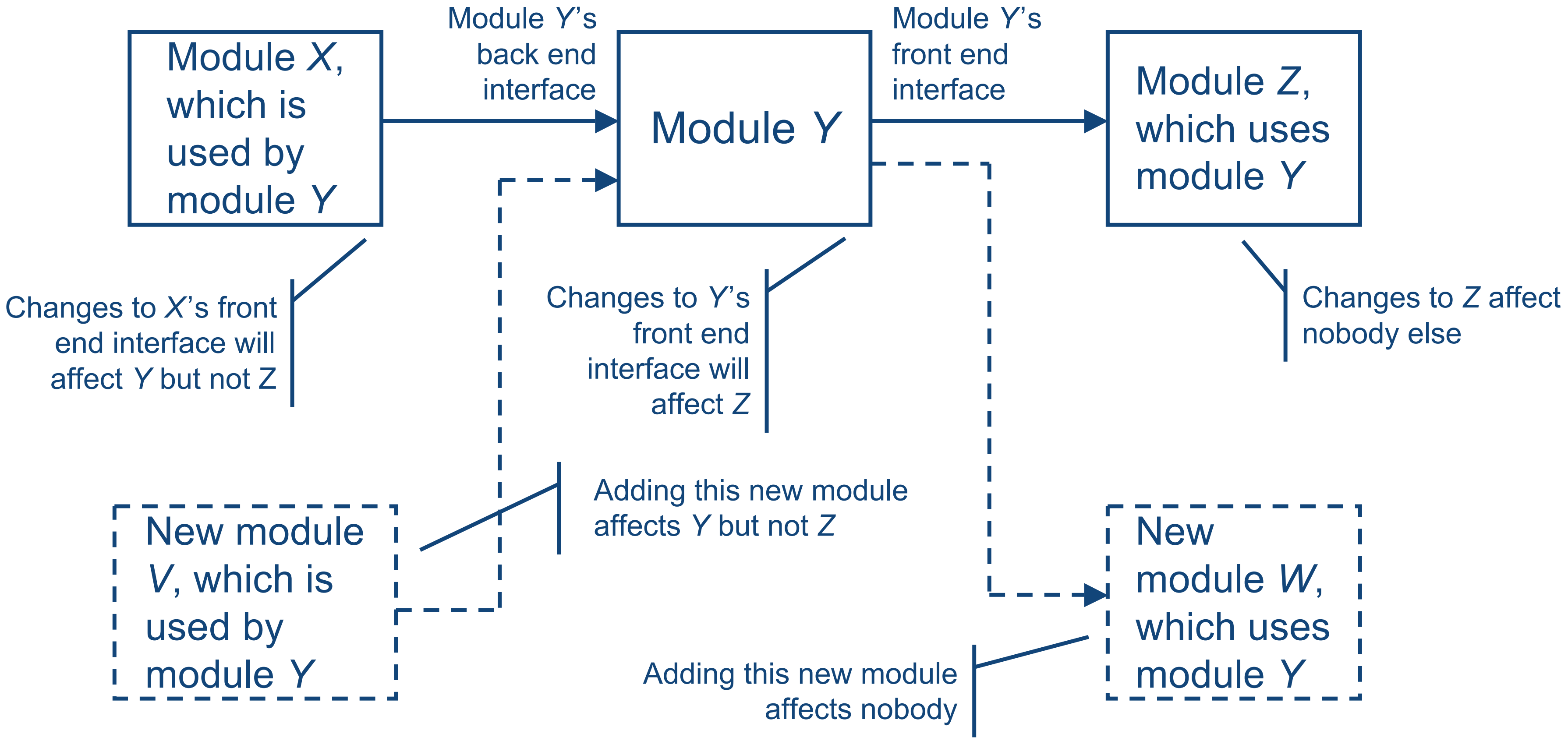
Modularisation and program maintenance

- In general a program ‘module’ is a construct that groups data structures or computations together
- In object-oriented code a module may be:
  - A *package*, which groups functionally-related classes
  - A *class*, which groups data structures and their related subroutines
  - A *method*, which performs one or more related computational processes
- The way a program is divided into modules affects our ability to:
  - *Understand* how the program works, to verify its correctness
  - Perform *corrective maintenance*, to fix errors
  - *Extend* or *enhance* the program, to add new functionality

Reference:
S. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill, 2005.

How changes to one module affect others

- In general a module interacts with other modules via a ‘front end’ and a ‘back end’:



Module *X*, which is used by module *Y*

Module *Y*’s back end interface

Module *Y*

Module *Y*’s front end interface

Module *Z*, which uses module *Y*

Changes to *X*’s front end interface will affect *Y* but not *Z*

Changes to *Y*’s front end interface will affect *Z*

Changes to *Z* affect nobody else

New module *V*, which is used by module *Y*

Adding this new module affects *Y* but not *Z*

New module *W*, which uses module *Y*

Adding this new module affects nobody

Coupling and cohesion

- Coupling and cohesion are two quantifiable measures of how well-designed, and hence how maintainable, modules are
  - An easily-maintainable program exhibits *high cohesion* and *low coupling* (aka *strong* cohesion and *weak* coupling)
- *Cohesion* is the degree of interaction *within* a module
  - High cohesion means the components of the module are closely related to one another — all of the code relevant to a particular feature of the program can be found in the same place
- *Coupling* is the degree of interaction *between* modules
  - Low coupling means modules are less dependent on one another — changes to one module affect fewer others

Categories of cohesion

- From worst to best:
  1. *Coincidental cohesion* — the module provides multiple, unrelated operations
  2. *Logical cohesion* — the module provides a single interface to multiple distinct operations, selected by the calling module
  3. *Temporal cohesion* — the module provides otherwise-unrelated operations which are all performed at the same time
  4. *Procedural cohesion* — the module provides operations which must be applied in a particular order
  5. *Communicational cohesion* — the module provides sequentially-applied operations which work on shared data
  6. *Functional cohesion* — the module provides a single operation or achieves a single goal
  7. *Informational cohesion* — the module performs multiple independent operations on an encapsulated data structure

Reference:
S. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill, 2005.

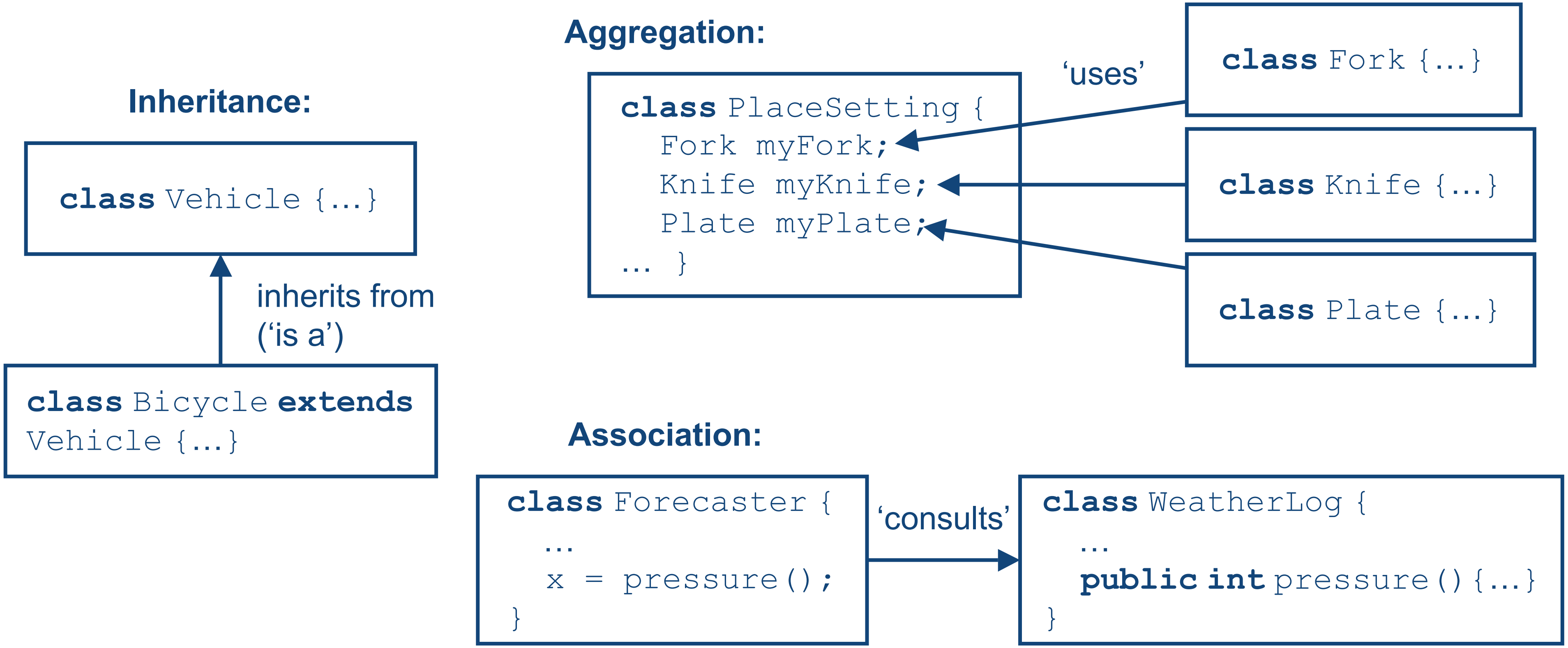
Categories of coupling

- From worst to best:
  1. *Content coupling* — one module directly accesses the contents of another
  2. *Common coupling* — two modules access the same shared data structure
  3. *Control coupling* — one module influences the actions of another by sending it ‘control signals’
  4. *Stamp coupling* — one module sends a whole data structure to another, but the second module only needs part of the data to perform its functions
  5. *Data coupling* — modules interact by exchanging homogeneous and necessary data only

Part B — Object orientation and maintainability

Modularisation in object-oriented programs

- Object orientation encourages us to produce programs with high cohesion and low coupling



Inheritance:

```
class Vehicle {…}
```

inherits from ('is a')

```
class Bicycle extends
Vehicle {…}
```

Aggregation:

```
class PlaceSetting {
  Fork myFork;
  Knife myKnife;
  Plate myPlate;
… }
```

'uses'

```
class Fork {…}
```

```
class Knife {…}
```

```
class Plate {…}
```

Association:

```
class Forecaster {
  …
  x = pressure();
}
```

'consults'

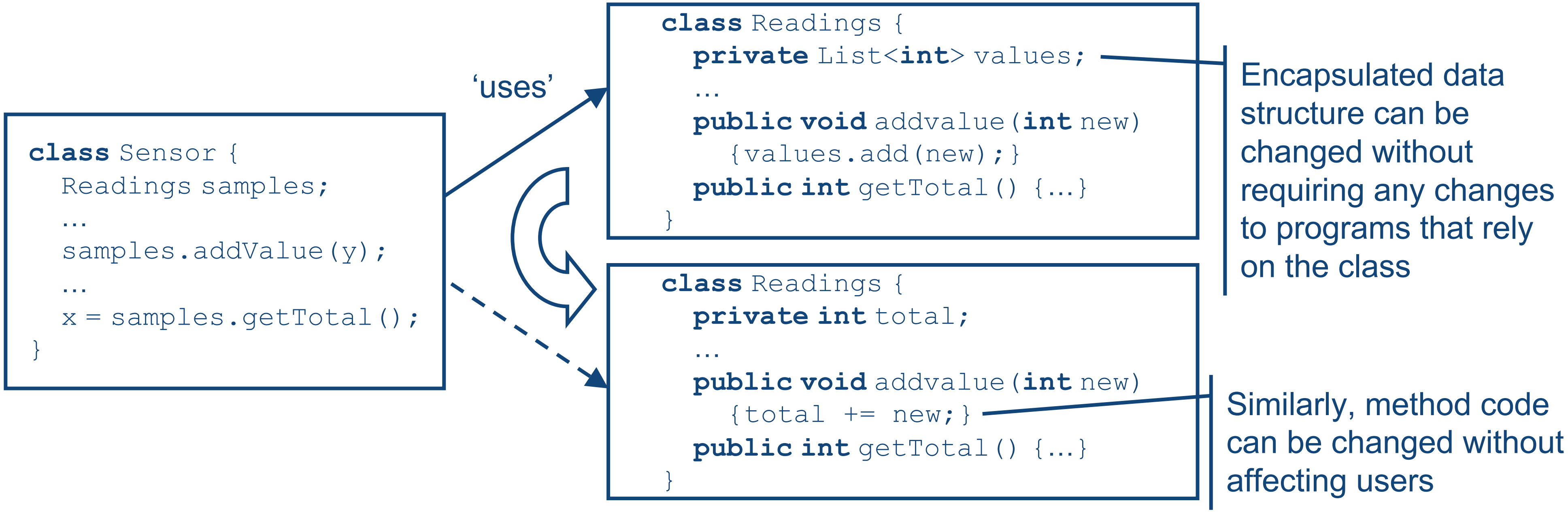
```
class WeatherLog {
  …
  public int pressure(){…}
}
```

QUT a university for the real world ®

Reference:
S. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill, 2005.

Data encapsulation and maintenance

- When all the fields in a class are **`private`**, and can be accessed only via **`public`** methods, changes to the data structure and methods that access it do not affect the class's users
  - Classes support both data and procedural abstraction
  - Public fields should be used for shared constants only



```
class Sensor {
  Readings samples;
  …
  samples.addValue(y);
  …
  x = samples.getTotal();
}
```

```
class Readings {
  private List<int> values;
  …
  public void addvalue(int new)
    {values.add(new);}
  public int getTotal() {…}
}
```

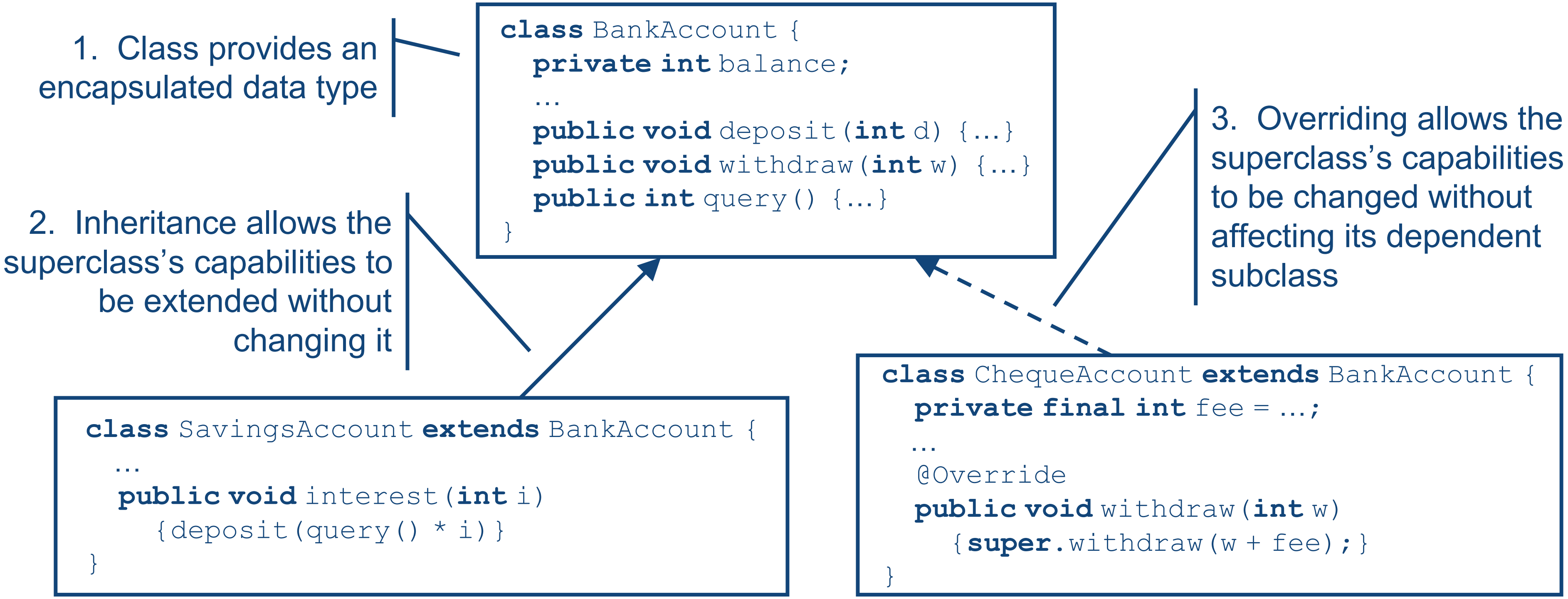
```
class Readings {
  private int total;
  …
  public void addvalue(int new)
    {total += new;}
  public int getTotal() {…}
}
```

Encapsulated data structure can be changed without requiring any changes to programs that rely on the class

Similarly, method code can be changed without affecting users

Inheritance, overriding and maintenance

- Class inheritance allows part of a program to be *extended* without changing it or affecting other parts that depend on it
- Method overriding allows part of a program to be *changed* without affecting other parts that depend on it



1. Class provides an encapsulated data type

```
class BankAccount {
  private int balance;
  …
  public void deposit(int d) {…}
  public void withdraw(int w) {…}
  public int query() {…}
}
```

2. Inheritance allows the superclass's capabilities to be extended without changing it

```
class SavingsAccount extends BankAccount {
  …
  public void interest(int i)
    {deposit(query() * i)}
}
```

3. Overriding allows the superclass's capabilities to be changed without affecting its dependent subclass

```
class ChequeAccount extends BankAccount {
  private final int fee = …;
  …
  @Override
  public void withdraw(int w)
    {super.withdraw(w + fee);}
}
```

Maintenance problems caused by object orientation

- Despite its benefits, object orientation also introduces some maintenance problems:
  - In a deeply-nested class hierarchy, where methods can be inherited and overridden, we may need to examine several classes in the hierarchy to find the declaration of a superclass method called in a subclass
  - Polymorphism and dynamic binding of identically-named methods in subclasses mean that when a run-time error occurs it's difficult to know which of the methods went wrong
  - Although we can change or add a subclass without affecting its superclasses, changes to a superclass potentially affect *all* of its subclasses

Reference:
S. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill, 2005.

Part C — Program design and coding for maintainability

Miscellaneous coding tips for making programs more maintainable

- Don’t comment obvious code
- Do use comments to explain the *purpose* of code (‘what’ not ‘how’)
- Keep comments up to date
- Use meaningful identifiers that explain the *purpose* of packages, classes, methods and fields
- Document *units* of measurement in comments:

```
    int length; // in metres
```

- Avoid *magic numbers* in the code — use named constants
- Avoid *arbitrary limits* on the size of data structures
- Use *assertions* during development and leave them in place when finished (disabled if necessary)
- Avoid *global variables*
- Don’t modify *pass-by-reference parameters*

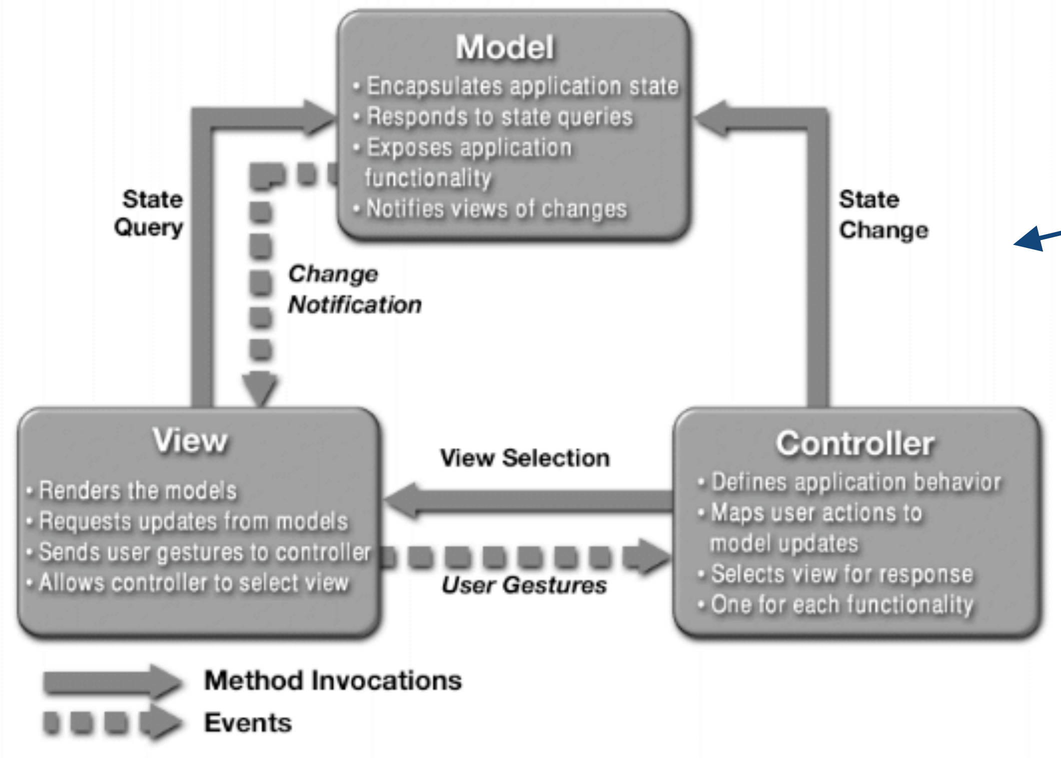
References:
Various programming discussion groups on the Internet

Miscellaneous coding tips for making programs more maintainable

- Check *return codes* and *catch exceptions*
- Strive to *write clearly*, in code and comments
  - What's convenient for the author is inconvenient and cryptic for the readers
  - Code is read more often than it's written
- Create *unit test cases*
- Keep your unit tests up to date

Design patterns

- Design patterns capture commonly-occurring program architectures, many of which aid program maintenance



Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
State Change
Change Notification
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

The 'MVC' pattern for interactive programs enforces low coupling

- Design patterns will be covered in a later lecture

Part D — Software metrics for assessing maintainability

Software metrics

- Software metrics attempt to quantify the ‘quality’ or ‘complexity’ of computer programs
  - And we can reasonably infer that poor quality or highly complex programs will be difficult to maintain
- To do this they measure static (syntactic) properties of the program’s source code
- Their usefulness is often debated for ‘coding in the small’:
  - Small, obscurely-written program fragments may be harder to understand than large, well-structured fragments
  - A program’s static structure may not be a good indicator of its dynamic (run time) properties
- However, they are clearly helpful for large-scale software development:
  - The more links there are between modules the harder a program is to understand and maintain

Software metrics in Eclipse

- Eclipse's Metrics plug-in module automatically calculates a wide-variety of software metrics that can be used to monitor your code's maintainability
- At the method level:
  - Number of parameters, lines of code, nested block depth, etc
  - *Cyclomatic Complexity*: the number of control flow paths through the code, with higher numbers indicating more 'complex' code
- At the class level:
  - Number of methods, attributes, overridden methods, etc
  - Depth in inheritance tree
  - *Specialization Index*: measures how the class overrides its superclass's methods
  - *Lack of Cohesion of Methods*: which (debatably) considers a class to be more cohesive if more methods access each attribute

Software metrics in Eclipse

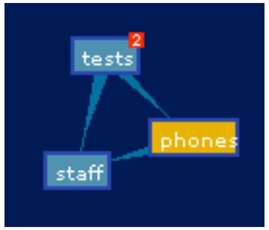
- At the package level:
  - Number of classes, number of interfaces, etc
  - *Afferent Coupling*: how many classes outside this package rely on one inside it, with higher values indicating that changes to this package will have a bigger impact on others
  - *Efferent Coupling*: how many classes in this package rely on one outside it, with higher values indicating an increased likelihood that changes to other packages will affect this one
  - *Abstractness*: the ratio of interfaces and abstract classes to concrete classes in the package, with higher values indicating that this package better protects its users from changes made within the package and can thus be changed more easily

Software metrics in Eclipse

  - *Instability*: the ratio of efferent coupling to all coupling, with higher values indicating that the package is more likely to be affected by changes to *other* packages
- At the project level:
  - Sums and averages of metrics for all the packages in the project
  - *Normalized Distance*: average of the differences between each package's abstractness and instability, with higher values indicating that the project is more brittle

Part E — Demonstration

Demonstration



- How inheritance and polymorphism allow us to extend a program's capabilities without disturbing the parts that are already in use:
  1. Create a class `Employee` for maintaing employee names in package `staff`
  2. Create classes `PhoneBook` and `PhoneNo` in package `phones` for maintaining phone directories with four-digit extensions
  3. Add a JUnit test class `PhoneBookTest` in package `tests`
  4. Check the project's metrics, especially for coupling
  5. Extend the program to allow new-style phone numbers with five-digit extensions by adding class `NewPhoneNo`
  6. Extend the program to allow for employees with a title via class `TitledEmployee`
  7. See which metrics have changed, especially method overriding and inheritance tree depth
  8. Test to show that old and new phone book entries can coexist

Homework

- Download and install Eclipse’s `Metrics` module and reads its (rather meagre) documentation:

  `http://metrics.sourceforge.net/`

- Read Doug Lea’s *Draft Java Coding Standard*

  `http://g.oswego.edu/dl/html/javaCodingStd.html`

- Read Roedy Green’s how-to guide for producing *Unmaintainable Java Code*, keeping in mind that these are things you should *not* do!

  `http://mindprod.com/jgloss/unmain.html`

  - In particular, read the *General Principles*, *Program Design*, *Testing*, *Ambiguity*, *Documentation*, *Coding Obfuscation* and *Miscellaneous Techniques* sections