Design and performance considerations

Reusing this material

Object Dependency Example

- What's undesirable about this?


```
// UserInterface
public void compute() {
    engine.compute();
}
```

```
// Engine
public void compute() {
    ...
    ui.updateGraphics();
}
```

Two-Way Interdependency

- As well as the UserInterface being dependent on the Engine, the Engine is dependent on the UI
- So the Engine can't run in isolation
  - difficult to slot in a different interface (eg. command line)
  - difficult to test the Engine component in isolation
- If there's a display problem, is it in the UserInterface or the Engine?
  - obvious place to look is the UserInterface
  - with this design the problem might be in the Engine

Functional separation

- Knowledge of the screen display is the responsibility of the UI *not* the Engine
- The Engine does not need to know about the UI
- ‘Knowledge Localisation’
  - if one component doesn’t need to know about another keep it that way!
  - Client-server is okay
  - This is server-server

Object Dependency Separation

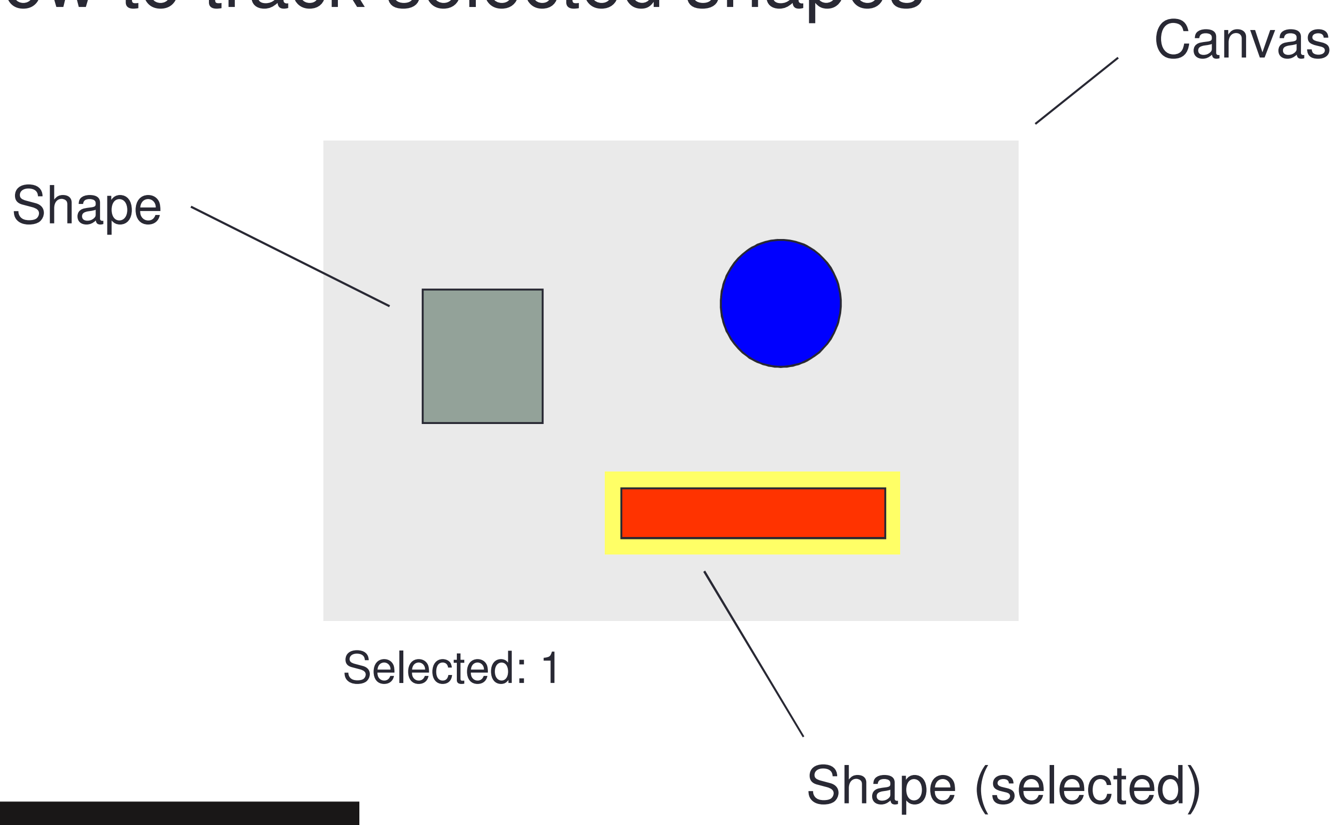
- Better solution


```
// UserInterface
public void compute() {
     engine.compute();
     unpdateGraphics();
}
```

```
// Engine
public void compute() {
     ...
}
```

Trade-Off Example

- Drawing package
  - How to track selected shapes



Slow Version


```
// Shape
public void select() {
  selected = true;
}

public void deselect() {
  selected = false;
}

public boolean isSelected() {
  return selected;
}
```

```
// Canvas
public int numSelected() {
  // Return the number of selected shapes.
  int n = 0;
  foreach shape in shapes {
    if (shape.isSelected()) {n = n + 1;}
  }
  return n;
}
```

Fast Version


```
// Shape
public void select {
   selected = true;
   canvas.incSelected();
}

public void deselect {
   selected = false;
   canvas.decSelected();
}
```

```
// Canvas
public void incSelected() {
   numSelected++;
}

public void Selected() {
   numSelected--;
}

public int numSelected() {
   // Return the number of selected shapes.
   return numSelected;
}
```

Storage of State

- Advantages
  - the changes allow numSelected() to run in constant time
- Disadvantages
  - they result in a bit more memory being consumed
    - pointers from shapes to the canvas
    - numSelected variable on Canvas
  - more seriously they create interdependency!
    - a Shape cannot run without a Canvas - may impede testing
    - ‘algorithm dependency’ - future algorithms must be very careful to update numSelected

Storage of State

- Suppose someone adds simple methods for adding and removing shapes

<u>:Canvas</u>

shapes: Collection of Shape

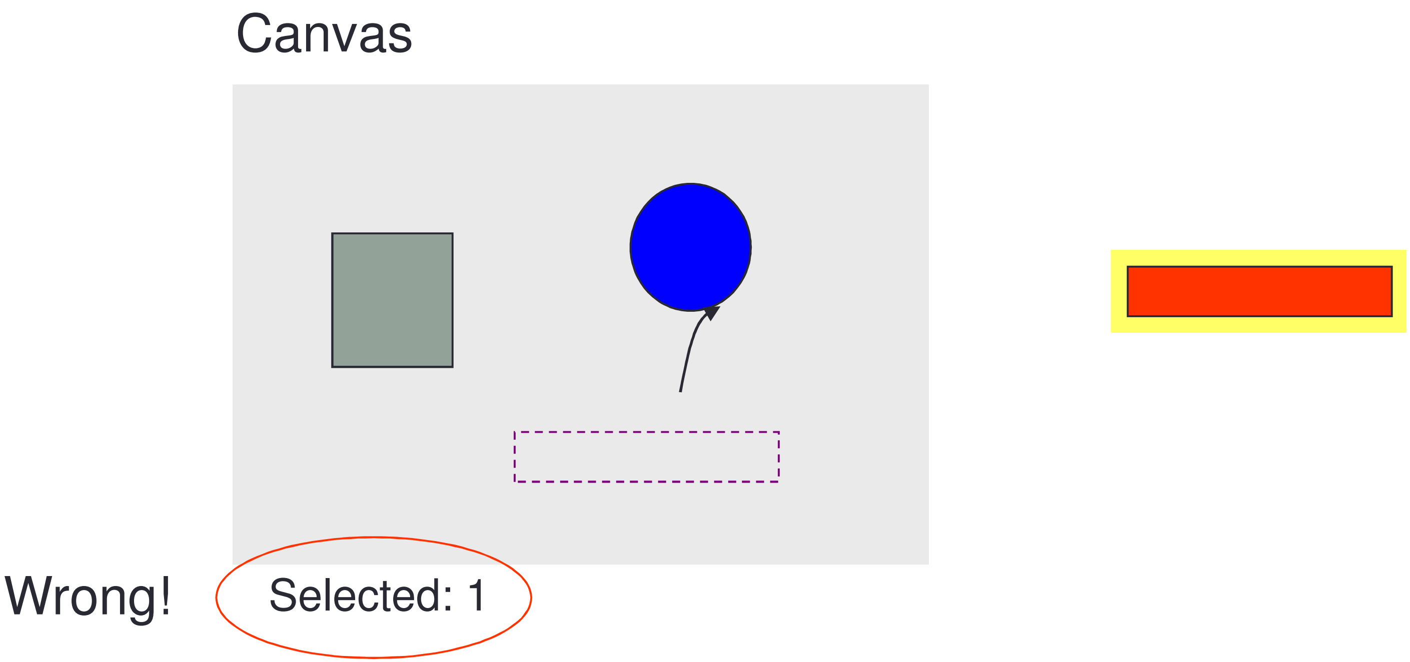
numSelected(): Integer
incrSelected()
decSelected()
addShape(s: Shape)
removeShape(s: Shape)

```
// Canvas
public void addShape(Shape s) {
  shapes.add(s);
}

public void removeShape(Shape s) {
  shapes.remove(s);
}
```

- What of the case where the shape being removed is also currently selected?

Trade-Off Example



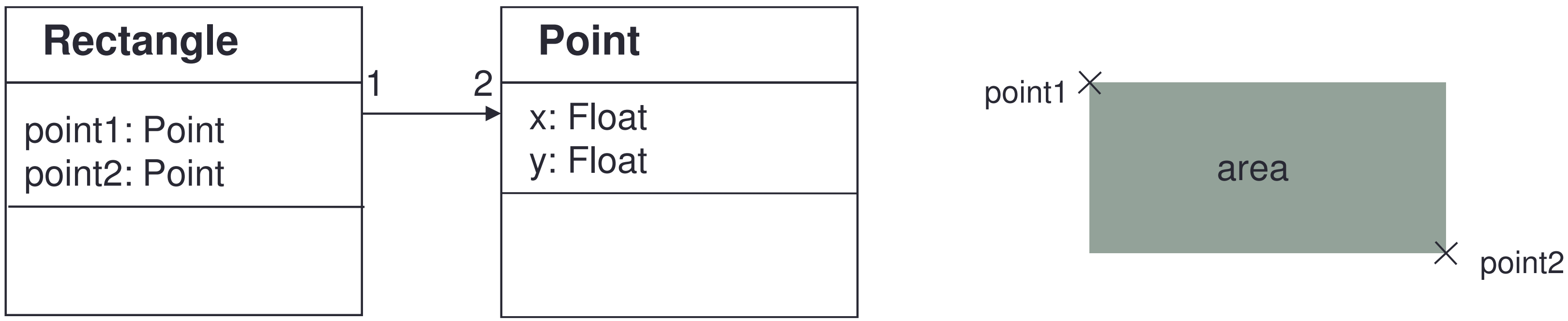
- The `numSelected` variable has not been updated
- It's a bug!

Trade-Off Summary

- Storage of state here is a trade-off between
  - speed
  - memory
  - complexity
    - specifically, the ease of extension and maintenance
- Only resolve in favour of speed if it really is a speed-critical section
- Try to minimise the *potential* for bugs

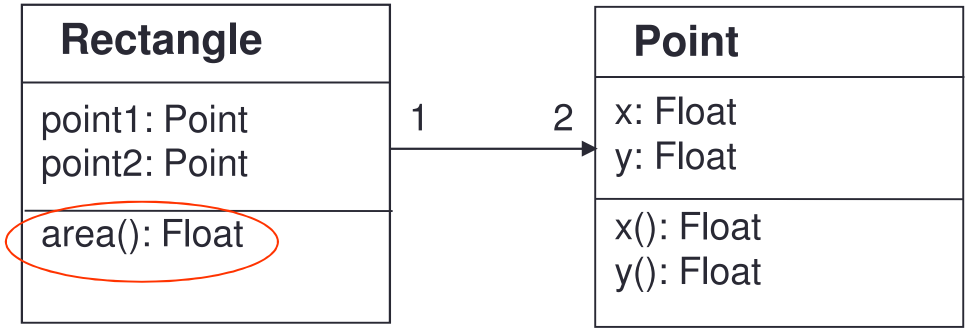
Derived Attributes

- A derived attribute is a piece of data which can be calculated from more-fundamental attributes
  - a rectangle's points are its fundamental attributes
  - a rectangle's area is a derived attribute



Derived Attributes

- Easiest plan is to implement area() as a method only

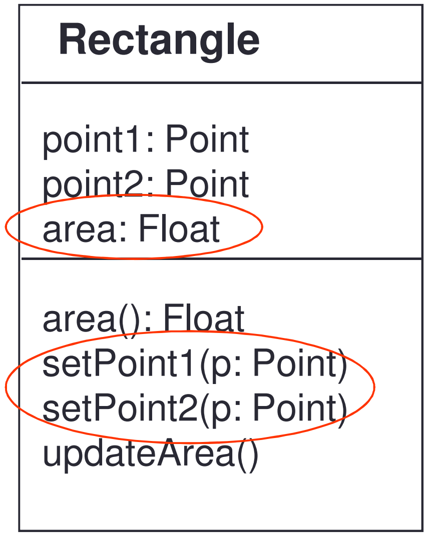



```
// Rectangle
public float area() {
  int w = point1.x() - point2.x();
  int h =  point1.y() - point2.y();
  return Math.abs(w * h);
}
```

- But if speed is critical, may have to store the area

Derived Attributes

- Storing the area creates interdependency amongst the variables
  - so ensure point1 and point2 can be modified *only* via an interface




```
// Rectangle
public void setPoint1(Point p) {
  point1 = p;
  updateArea();
}

public void setPoint2(Point p) {
  point2 = p;
  updateArea();
}

private void updateArea() {
  int w = point1.x() - point2.x();
  int h =  point1.y() - point2.y();
  area = Math.abs(w * h);
}
```

Factoring Out Side-Effects

- Often good practice to divide a method into a calculation part and a side-effect part
  - localising side-effects makes debugging easier
  - making methods smaller and simpler is a worthwhile goal anyway

```
// Rectangle
public void updateArea() {
  // Calculate new area and store it.
  int w = point1.x() - point2.x();
  int h =  point1.y() - point2.y();
  area = Math.abs(w * h);
}
```

```
// Rectangle
public void updateArea() {
  area = calculateArea();
}

private float calculateArea() {
  // Return the area. No side-effects.
  int w = point1.x() - point2.x();
  int h =  point1.y() - point2.y();
  return Math.abs(w * h);
}
```

Method location

- The `calculateArea()` method needs to access point data

```
// Rectangle
public float calculateArea() {
  int w = point1.x() - point2.x();
  int h =  point1.y() - point2.y();
  return Math.abs(w * h);
}
```

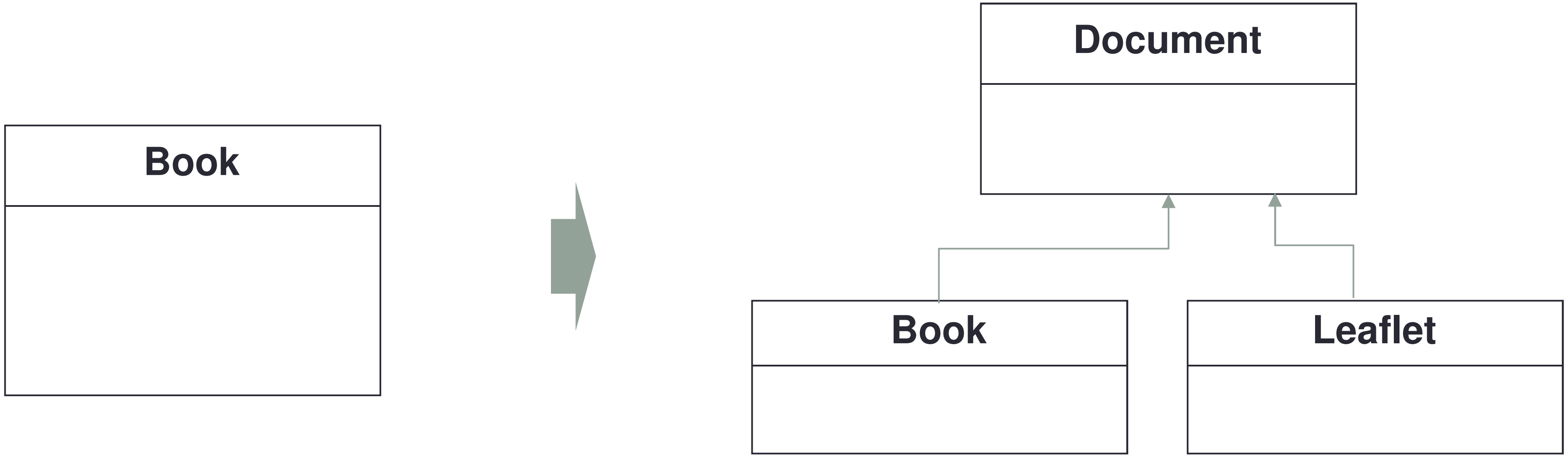
- This is a clue to think about implementing it elsewhere e.g. greater flexibility if it's on Point

```
// Point
public float rectangularAreaTo(Point p) {
  int w = p.x() - x;
  int h =  p.y() - y;
  return Math.abs(w * h);
}
```

```
// Rectangle
public float calculateArea() {
  return point1.rectangularAreaTo(point2);
}
```

Designing Classes for Reuse

- When designing a class, consider splitting it into a hierarchy even if there's no current necessity
  - if you can identify a sensible concept, factor it out
  - lots of small classes are better than one big class
    - keeps complexity down
    - makes subsequent reuse much easier



Arguments are Healthy

- If a method relies on some global 'constant', try to build it from a method which doesn't

```
// Event
public void process() {
  …
  t = t + Simulator.getClockStep();
  …
}
```

```
// Event
public void process() {
  process(Simulator.getClockStep());
}

public void process(float clockStep) {
  …
  t = t + clockStep;
  ...
}
```

- Turns implicit dependency into an explicit one
- Improves flexibility and reuse opportunities

Let the Compiler do the Work

- Consider an options component within a system
  - provides a mapping of string names to integer values

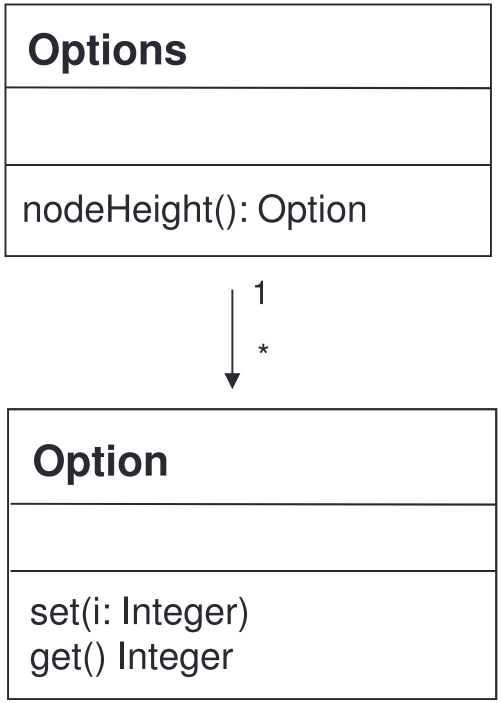
Options
name: String number: Integer
set(name: String, i: Integer) get(name: String): Integer

```
options.set("NodeHeight", 45);
options.get("NodeHeeght");
```

- What happens if you make a spelling mistake?
  - runtime error

Let the Compiler do the Work

- Much safer to add a class for an option, and access the “NodeHeight” option instance via a method



- What happens if you make a spelling mistake now?
  - compilation error - excellent!

Reuse

- Reuse
  - The process of creating software systems from exisiting software assets
- Object-oriented programming *encourages* reuse
  - Encapsulation: specifies which operations access which data
  - Polymorphism: restricts the assumptions of an object to a well defined protocol
  - Inheritance: allows re-use of a class whose behaviour provides some of the behaviour of a new class
- However successful reuse also requires
  - Better ways to describe behaviour + interfaces
  - Easier ways to plug parts together

Reuse

- Classes
  - The traditional entity of reuse
  - However reuse of a set of classes is more useful
- Frameworks
  - Set of classes with well defined interactions
  - Designed to solve specific problems
  - *Customisable*, implementation may only be partially defined
- Components
  - Independent software entities (e.g. a set of classes) which can be integrated *unchanged* into larger systems
- Component Based Development
  - Describes software developed by assembling exisiting components

Reuse

- Frameworks and Patterns relate to software reuse
- Expertise reuse is also important
- Patterns
  - A general solution to a problem
  - Abstracting common practice in solving a similar set of problems
- Process Patterns
  - Describe rules which can be followed when building software systems
- Design Patterns
  - Describe a set of classes and objects which solve a general design problem, for you to customise

Development and performance

- OO programming is focussed on code reuse and safe development
- OO programming does not guarantee good code
- Poor design will lead to poor code
- For Fortran the correct split of modules and classes is key
- Provided objects are not too low-level performance shouldn't be affected
  - Accessing individual array elements through methods in computational kernels will damage performance
  - Constantly calculating data that could be stored may damage performance
  - Storing data that could be calculated on the fly may damage performance
  - OO functionality itself will not add much overhead

Performance

- Example performance investigation
  - Very old
  - F90 features not F2003

Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March, 1997, SIAM Press

High Performance Object-Oriented Scientific Programming in Fortran 90

Charles D. Norton* Viktor K. Decyk† Boleslaw K. Szymanski‡

Language	Compiler	Particles	Time (sec.)
Two-Dimensional Program			
Fortran 77	IBM xlf90	3,571,712	193.52
Fortran 77	IBM xlf	3,571,712	195.08
Fortran 90	IBM xlf90	3,571,712	202.88
C++	IBM xlC	3,571,712	359.00
Three-Dimensional Program			
Fortran 77	IBM xlf90	7,962,624	1548.71
Fortran 77	IBM xlf	7,962,624	1550.14
Fortran 90	IBM xlf90	7,962,624	1339.91
C++	IBM xlC	7,962,624	2797.00

- Performance ultimate dependent on implementation

Operations	Fortran 77	Fortran 90	C++
Advance	225.18	168.49	348.24
Deposit	66.38	69.52	223.19

Summary

- Make life easy for yourself
  - make the code easy to read and understand
  - minimise dependencies among objects
  - make a dependency explicit if you can’t avoid it
  - decide on a class’s responsibilities and adhere to them
  - don’t add complexity to gain speed unless you really have to
  - design code with a careful eye on flexibility and re-use
  - design methods that encapsulate small pieces of functionality
  - give as much of the testing burden to the compiler as you can