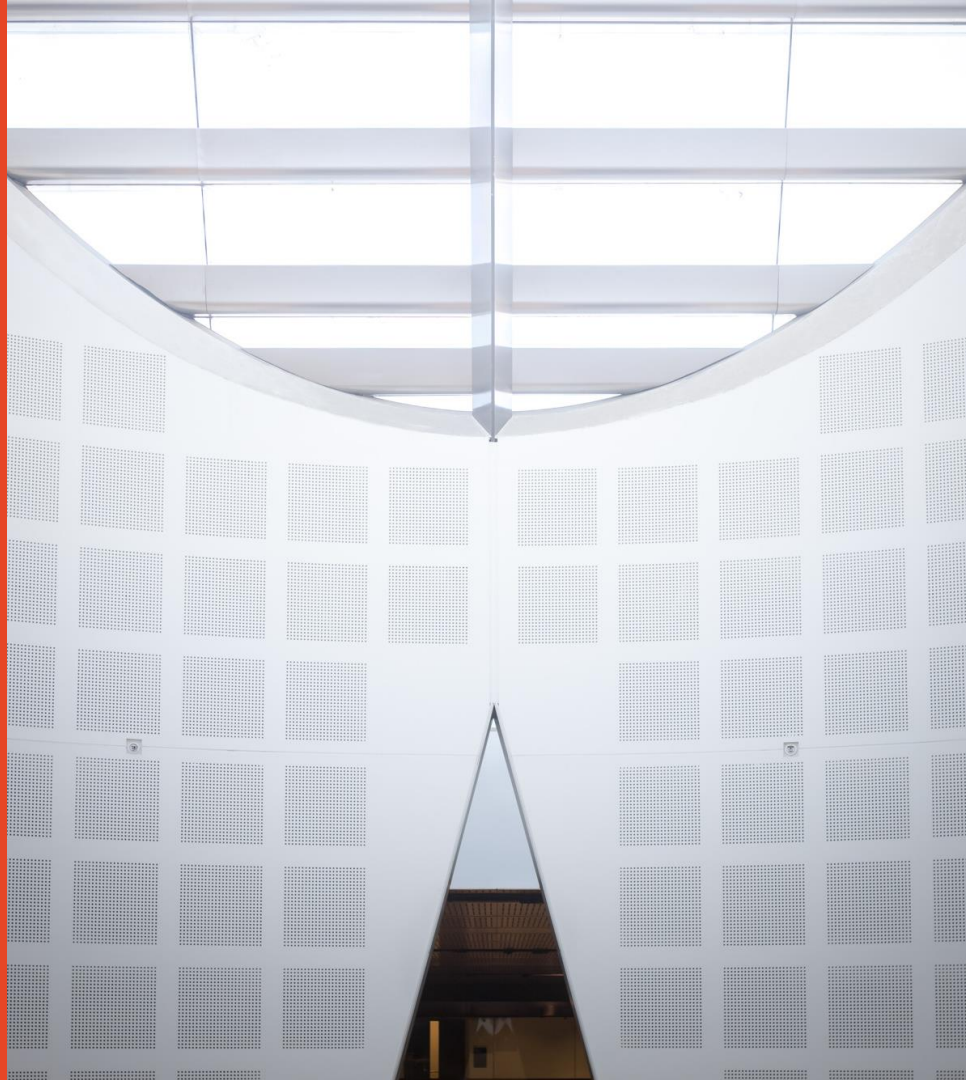
Software Construction and Design 2
SOFT3202 / COMP9202
Advanced Design Patterns (GoF & Enterprise)

Dr. Basem Suleiman

School of Information Technologies

THE UNIVERSITY OF SYDNEY



Copyright Warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

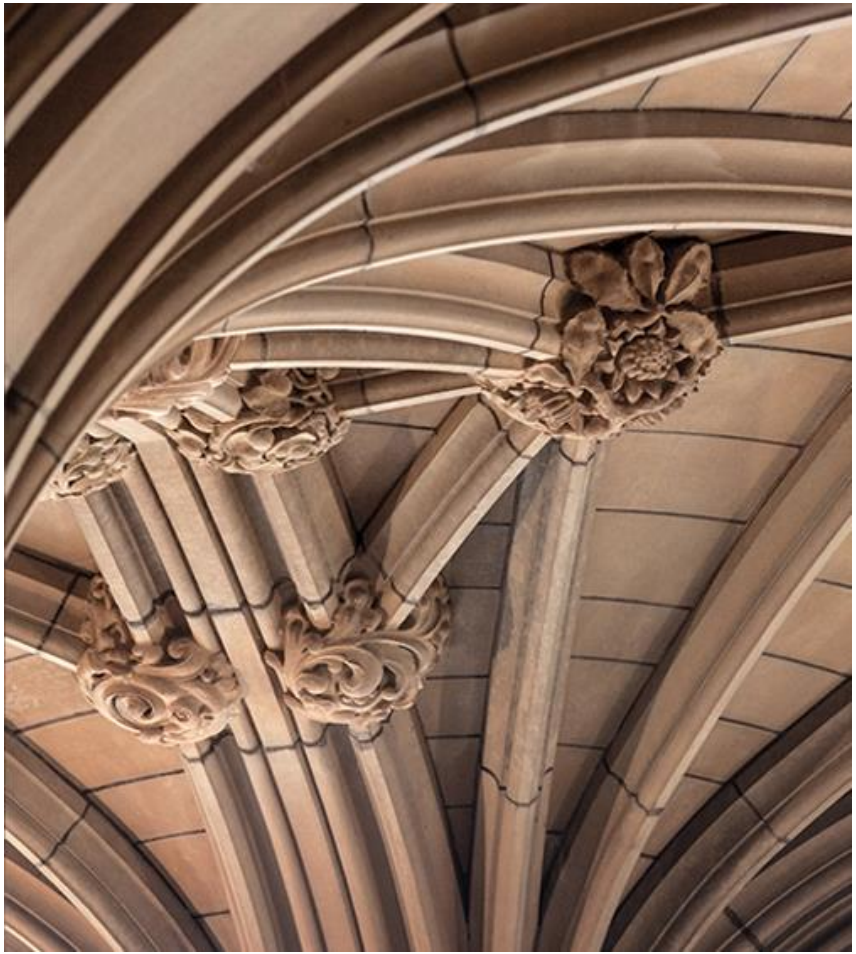
Do not remove this notice.

Agenda

- GoF Design Patterns
  - Visitor
  - Template Method
- Model-View-Controller
  - Page Controller
  - Template View

Visitor Design Pattern (GoF)

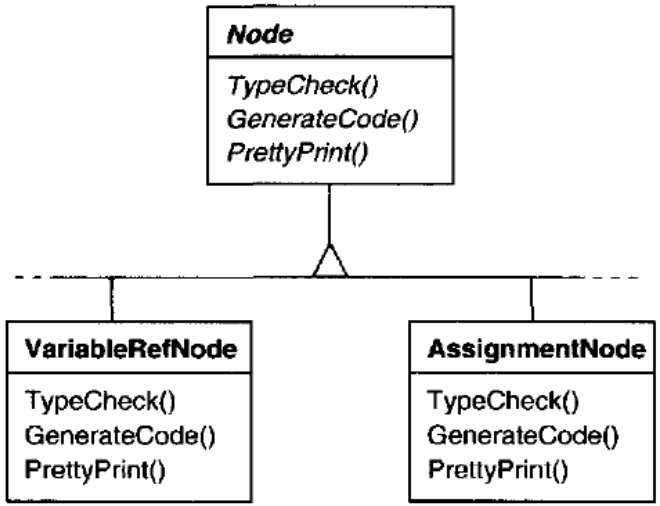
Object Behavioural



THE UNIVERSITY OF SYDNEY

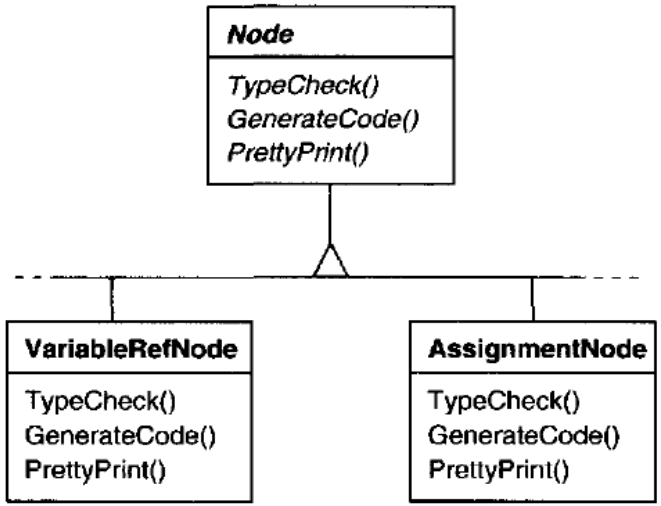
Motivation – A Compiler

- A compiler represents programs as abstract syntax trees
- Operations like type-checking, variable assignment and code generation
- Different classes (nodes) for different statements (e.g., assignment statement, arithmetic expressions
- *Discuss: is this a good design? Why/why not?*



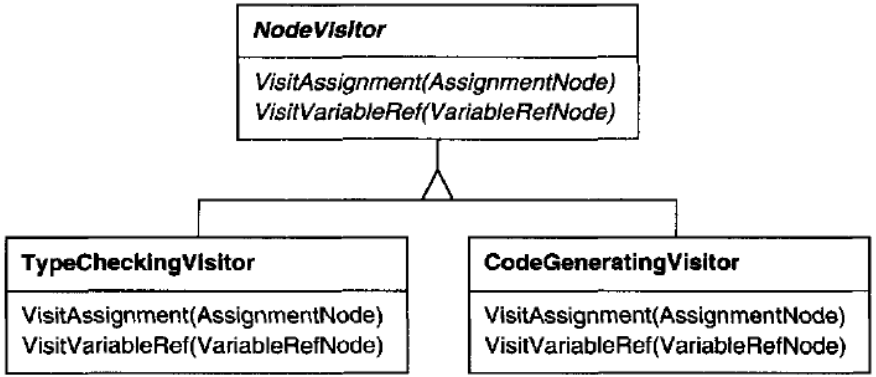
Motivation – A Compiler

- Problems:
  - Operations are distributed across various node classes
    - Difficult to understand, maintain and change design
  - Add new operations will require recompiling all of the classes
    - Difficult to extend

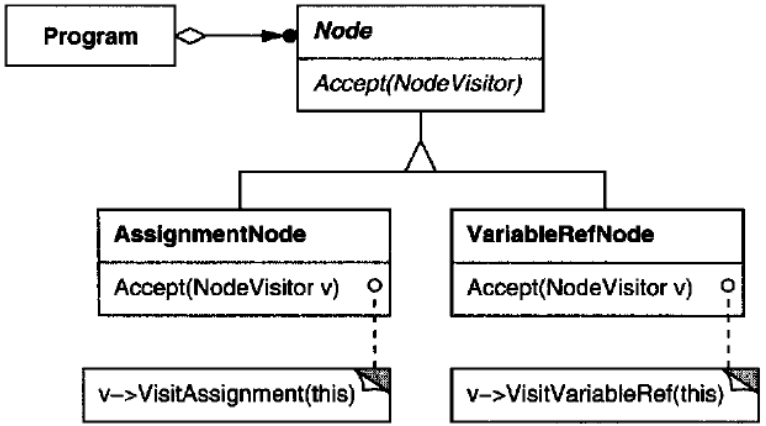


A Compiler Application – Improved Design

– .



Node Visitor Hierarchy



Node Hierarchy

Visitor Pattern – Class Hierarchies

- Node Hierarchy
  - For the elements being operated on
- Node Visitor Hierarchy
  - For the visitors that define operations on the elements
- To create a new operation, add a new subclass to the visitor hierarchy

Visitor Pattern – How it Works

- Group set of related operations from each class in a separate object (visitor)
- Pass this object to elements of the syntax tree as it is traversed
- An element accepts the visitor to be able to send request to (element passed as an argument)
- The visitor will run the operation for that element

Visitor Pattern

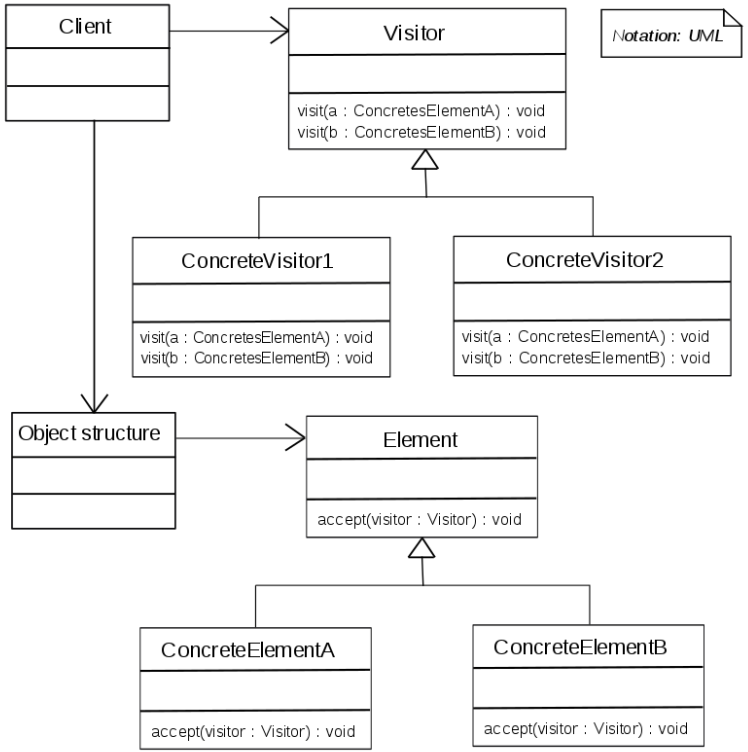
- Object behavioral

- Intent:
  - Modify a new operation without changing the classes of the elements on which it operates

- Applicability:
  - You want to perform operations on objects that depend on their concrete classes
  - You want to avoid mixing many distinct unrelated objects' operations in an object structure with their classes
  - The classes that define the object structure rarely change, but often it is needed to define new operations over the structure

Visitor Pattern – Structure

–



By Translated German file to English, CC BY 3.0, https://en.wikipedia.org/w/index.php?curid=52845911

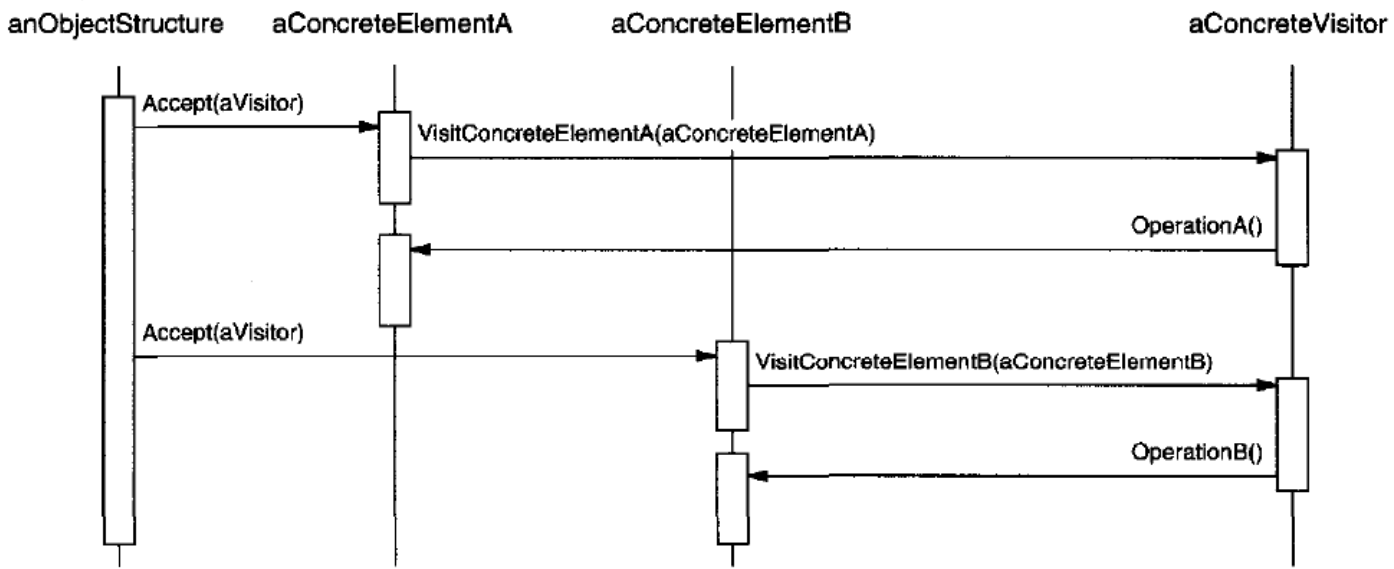
Visitor Pattern – Participants

- Visitor (NodeNisitor)
  - Declares a Visit operation for each class of ConcreteElement in the object structure
  - Classes identified by the operation's name and signature
- ConcreteVisitor (TypeCheckVistor)
  - Implement each operation declared by Visitor
  - Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure
- Element (Node)
  - Defines an "Accept" operation that takes a visitor as an argument

Visitor Pattern – Participants

- ConcereteElement (AssignmentNode,VariableRefNode)
  - Implements Accept operation (visitor as argument)
- ObjectStructure (Program)
  - Can enumerate its elements
  - May either be composite or a collection
  - May provide an interface to allow the visitor to visit its elements

Visitor Pattern – Collaboration



Visitor Pattern – Benefits

- Easy way to add new operations
  - Add a new Visitor
- Gather related behaviors (algorithms defined in the Visitors)
  - Specific data structure can be hidden in the visitor
- Visiting across class hierarchies
  - It can visit object structures with different types of elements (unlike iterator)
- State accumulation in the object structure
  - Otherwise, pass the state as an argument to the operations or declare it as global variables

Visitor Pattern – Drawbacks

- Violating encapsulation
  - It may enforce using public operations that access an element's internal state

- Difficult to add new concrete element classes
  - Adding new Concrete Element requires adding new abstract operation on Visitor and a corresponding implementation in every Concrete Visitor
    - Exception: default implementation to be provided or inherited by most Concrete Visitors

  - Consider the likelihood to change the algorithm applied over an object structure or classes of objects that make up the structure

Visitor Example – Element Implementation

```
1  interface Element {
2      void accept(Visitor v);
3  }
```

```
4  class ConcreteElementA implements Element {
5      public void accept(Visitor v) {
6          v.visit(this);
7      }
8      public String getElementA() {
9          return "Concrete Element A";
10     }
11 }
12 class ConcreteElementB implements Element {
13     public void   accept( Visitor v ) {
14         v.visit( this );
15     }
16     public String getElementB() {
17         return "Concrete Element B";
18     }
19 }
20 class ConcreteElementC implements Element {
21     public void accept(Visitor v) {
22         v.visit(this);
23     }
24     public String getElementC() {
25         return "Concrete Element C";
26     }
27 }
```

Visitor Example – Visitor Implementation

```
29 interface Visitor {
30     void visit(ConcreteElementA a);
31     void visit(ConcreteElementB b);
32     void visit(ConcreteElementC c);
33 }
```

```
34 class ConcreteVisitor1 implements Visitor {
35     public void visit(ConcreteElementA a) {
36         System.out.println("do visitor1 on " + a.getElementA());
37     }
38     public void visit(ConcreteElementB b) {
39         System.out.println("do visitor1 on " + b.getElementB());
40     }
41     public void visit(ConcreteElementC c) {
42         System.out.println( "do visitor1 on " + c.getElementC() );
43     }
44 }
45
46 class ConcreteVisitor2 implements Visitor {
47     public void visit(ConcreteElementA a) {
48         System.out.println("do visitor2 on " + a.getElementA());
49     }
50     public void visit(ConcreteElementb b) {
51         System.out.println("do visitor2 on " + b.getElementB());
52     }
53     public void visit(ConcreteElementC c ) {
54         System.out.println("do visitor1 on " + c.getElementC());
55     }
56 }
```

Visitor Example – Client Implementation

```
58 public class VisitorClient {
59     public static void main( String[] args ) {
60         Element[] list = {new ConcreteElementA(), new ConcreteElementB(), new ConcreteElementC()};
61         ConcreteVisistor1 v1 = new ConcreteVisistor1();
62         ConcreteVisistor2 v2 = new ConcreteVisistor2();
63         for (Element element : list) {
64             element.accept(v1);
65         }
66         for (Element element : list) {
67             element.accept(v2);
68         }
69     }
70 }
```

- What is the output of this code?

Visitor – Implementation (1)

- Each object structure will be associated with a Visitor (abstract class)
  - Declares *VisitConcreteElement* for each class of *ConcreteElements* defining the object structure
  - Visit operation declares a particular *ConcreteElemnt* as its argument to allow the visitor to access the interface of *ConcreteElement* directly
  - *ConcreteVistor* classes override each visit operation to implement visitor-specific behavior

- *ConcreteElement* classes implement their Accept operation that calls the matching Visit operation on the Visitor for that *ConcreteElement*

Visitor – Implementation (2)

- Double dispatch
  - The execution of an operation depends on the kind of request and the types of two receivers

- Visitor pattern allows adding operations to classes without changing them through the Accept method which is double dispatch
  - Accept method depends on Visitor and Element types which let visitors request different operations on each class of element

Visitor – Implementation (3)

- Traversing the object structure; a visitor must visit each element of the object structure – How?

- Can be a responsibility of
  - Object structure: a collection iterates over its elements calling the Accept operation on each (use *Composite*)

  - Separate iteration object: using an *Iterator* to visit the elements
    - Internal operator will call an operations on the visitor with an element as an argument (not using double dispatching)

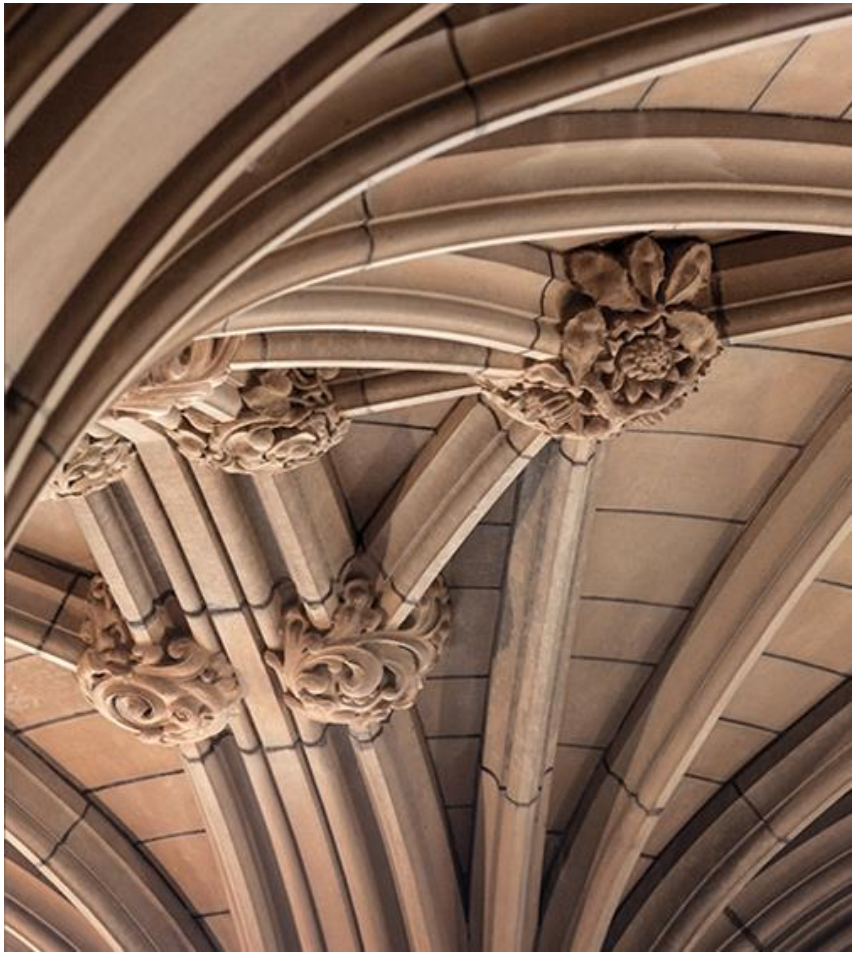
  - Visitor: implement the traversal logic in the Visitor
    - Allows to implement particularly complex traversal
    - Code duplication

Visitor – Related Patterns

- Composite
  - The composite pattern can be used to define an object structure over which a Visitor can iterate to apply an operation
- Interpreter
  - Visit or may be applied to do the interpretation

Template Method Pattern (GoF)

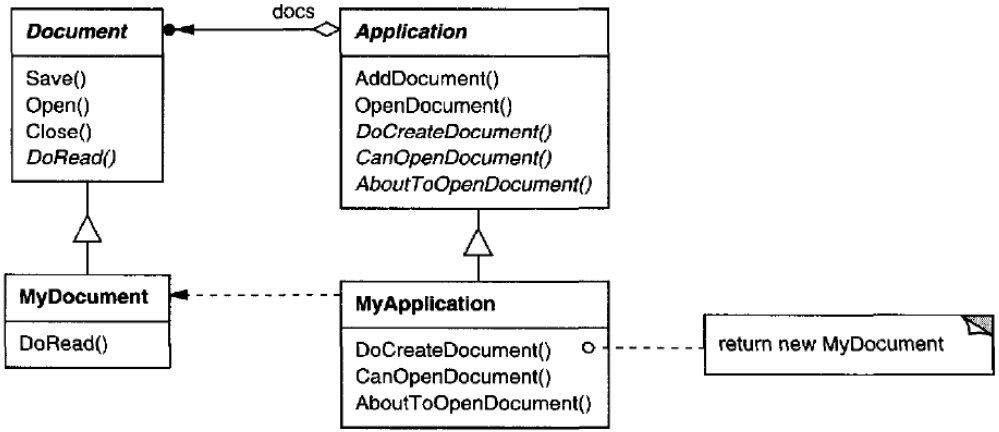
Class behavioural



THE UNIVERSITY OF SYDNEY

Motivating Scenario

- Application framework
- Application opens existing documents stored in external format
- Document represents the document's info once its read
- Sub-classing:
  - SpreadsheetDocument and SpreadsheetApplication are Spreadsheet Application
- *OpenDocument()* to define the algorithm for opening and reading a document (Template Method)



Motivating Scenario – Template Method

- A template method defines abstract operations (algorithm) to be concretely implemented by subclasses

- Application sub-classes
  - Some steps of the algorithm (e.g., for CanOpenDocument() and DoCreateDocument())

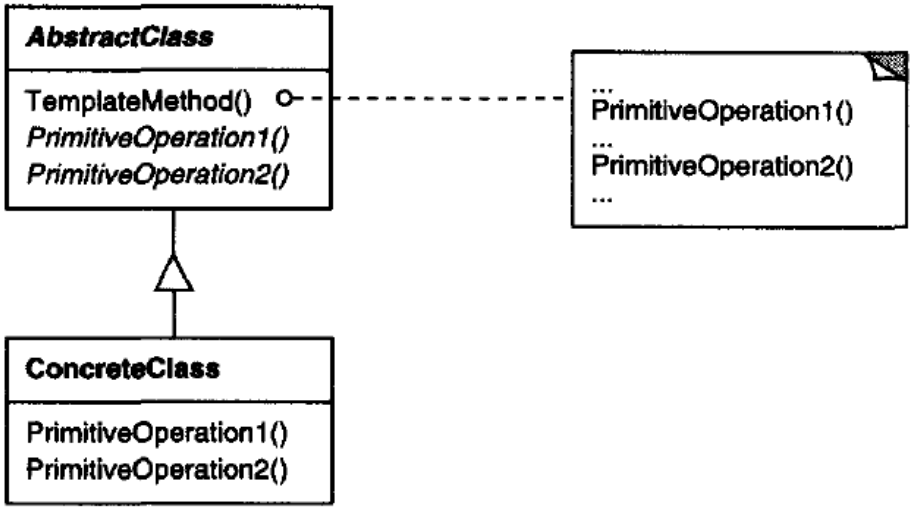
- Document sub-classes
  - Steps to carry on the operation (e.g., DoRead() to read the document)

- Let sub-classes know when something is about to happen in case they care
  - E.g., AboutToOpenDocument()

Template Method Pattern

- Class behavioral
- Intent
  - Let subclasses redefine certain steps of an algorithm without changing the algorithm's structure
- Applicability
  - Implement invariant parts of an algorithm once and let subclasses implement the varying behavior
  - Common behavior among subclasses to reduce code duplication ("*refactoring to generalize*")
  - Control subclasses extensions
    - Template method calls hook operations at specific points

Template Method Pattern – Structure



Template Method – Participants and Collaboration

- AbstractClass (Application)
  - Defines abstract primitive operations
  - Implement a template method defining an algorithm's skeleton
  - The template method calls primitive operations and AbstractClass' operations

- ConcreteClass (MyApplication)
  - Implements the primitive operations to perform sub-class specific steps of the algorithm
  - Relies on Abstract class to implement invariant algorithm's steps

Template Method – Consequences

- Code reuse (e.g., class libraries)
- Inverted control structure ("the Hollywood principle" – "Don't call us, we'll call you"
- Template methods can call:
  - Concrete operations (ConcreteClass or client classes)
  - Concrete AbstractClass operations
  - Primitive (abstract) operations (must be overridden)
  - Factory methods
  - Hook operations
    - Provides default behavior that subclass can extend if needed
    - Often does nothing by default, subclass override it to extend its behavior
  - Subclasses must know which operations are designed for overriding (hook or abstract/primitive)

Template Method Pattern – Implementation

- Minimize primitive operations a subclass must override
  - More primitive operations can increase client's tasks

- Naming conventions to identify operations that should be overridden
  - E.g., MacApp framework (Macintosh applications) prefixes template method names with "Do" (DoRead, DoCreateDocument)

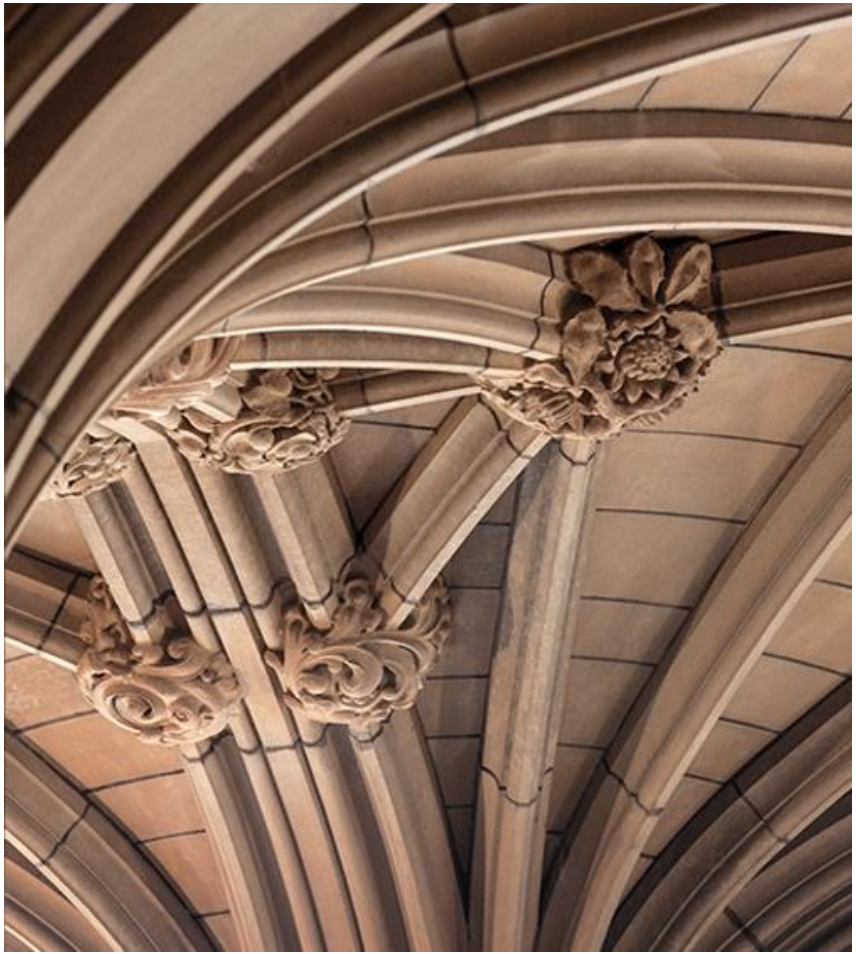
Template Method Pattern – Related Patterns

- Factory
  - Template Methods often call Factory Methods
  - e.g., DoCreateDocument called by OpenDocument

- Strategy
  - Strategy vary the entire algorithm using delegation
  - Template method vary part of an algorithm using inheritance

Model View Controller

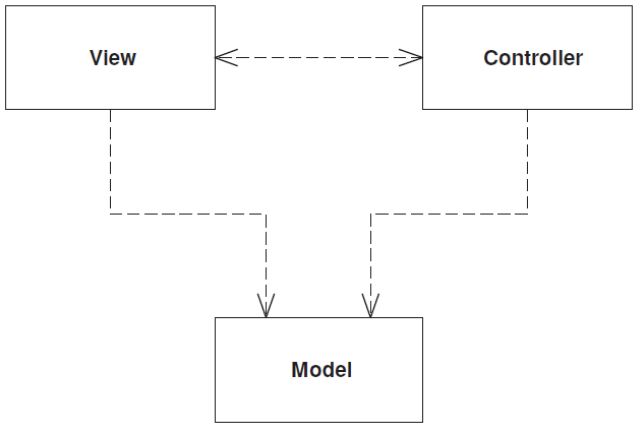
Enterprise (Web) Application



THE UNIVERSITY OF SYDNEY

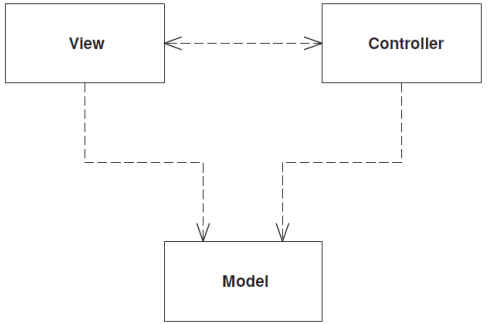
Model View Controller (MVC)

– *"Splits user interface interaction into three distinct roles"*



MVC – How it Works

- *Model*
  - An object represent information about the domain
  - E.g., customer object
- *View*
  - Representation of the model (data) in the UI
  - UI widget or HTML page rendered with info from the model
- *Controller*
  - Handle user interactions, manipulate the model and update the view accordingly



MVC – When to Use it

- MVC separate the presentation from the model and the controller from the view

- Benefits of separating the presentation from the model:
  - Separation of concerns (UI design vs. business policies/database interactions)
  - Multiple different views based on the same model
  - Easier to test the domain logic without irrespective to UI

MVC – View and Model

- Dependencies
  - The presentation depends on the model but not vice-versa
  - Maintain the presentation without changing the model

- Discuss:
  - What design consequences that might arise from the dependency of the presentation on the model? How to addressed it?

MVC – View and Model

- Dependencies
  - The presentation depends on the model but not vice-versa
  - Maintain the presentation without changing the model

- Discuss:
  - What design consequences that might arise from the dependency of the presentation on the model? How to addressed it?

- Several presentations of a model on multiple windows/UIs. If the user makes changes to the model through one of the presentations, this should be reflected in the other presentations
- Use the Observer pattern (GoF); presentation is observer of the model

MVC –View and Controller

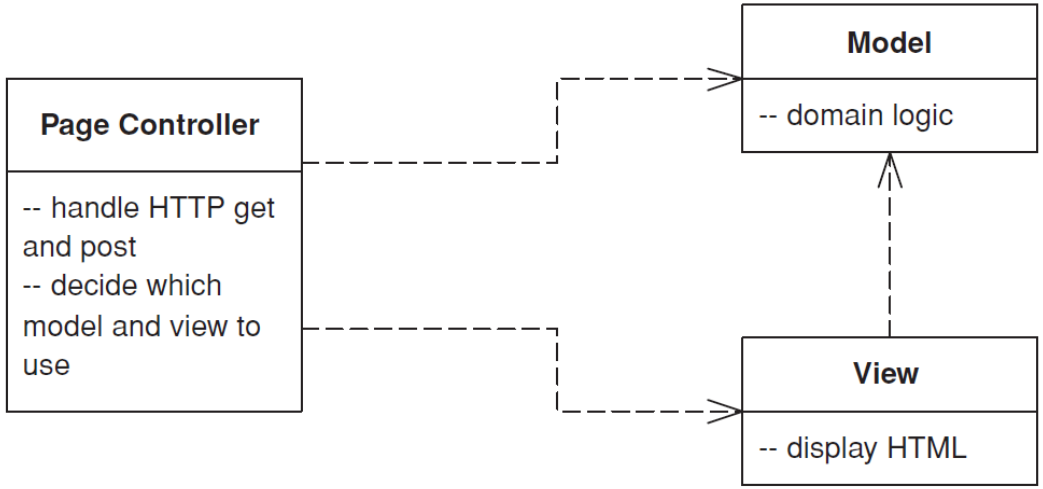
- Separation/dependency is less important

- Can be designed/implemented differently
  - One view and two controllers to support “editable” and “non-editable” behavior
    - Controllers as Strategies (GoF) for the view
  - In practice, one controller per view
  - Web interfaces help popularizing the separation

- Some GUI frameworks combine view and controller but led to confusion

MVC – Page Controller

"An object that handles a request for a specific page or action on a Web site"



Page Controller – How It works

- One module on the Web server act as the controller for each page on the Web site (ideally)
- With dynamic content different pages might be sent, controllers link to each action (e.g., click a link or button)
- Page controller can be structured as a script (e.g., servlet) or as a server page (e.g., PHP, JSP)

Page Controller – Responsibilities

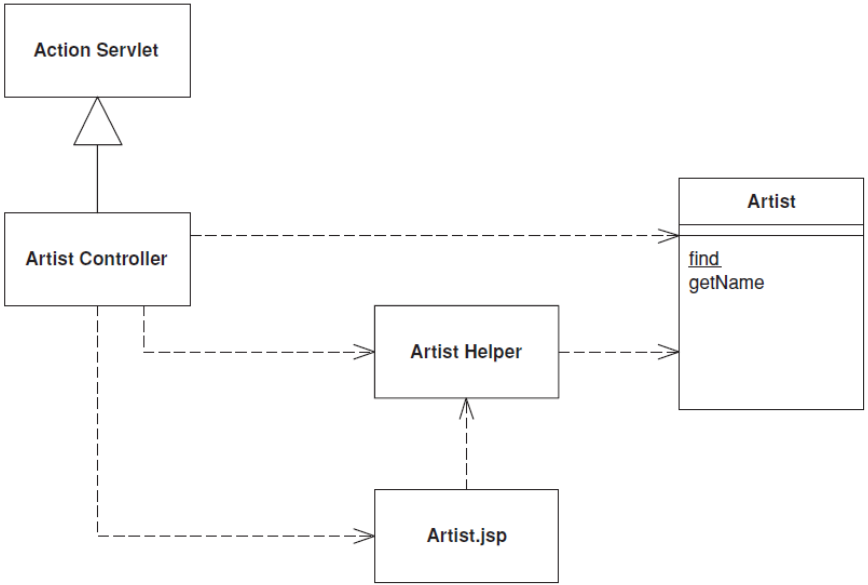
- Decode the URL and extract form data for the required request action
- Create and invoke any model objects to process the data
- Decide which view to display the result page and forward model information to it
- Invoke helper objects to help in handling a request
  - Handlers that do similar tasks can be grouped in one (reduce code duplication)

Page Controller – When to Use it

- Whether to use Page Controller or Front Controller?
- Page controller works particularly where most URLs can be handled with a server page or script, and more complex ones with helper objects

Page Controller – Example

- Simple display with a servlet Controller and a JSP view in Java
  - http://www.thingy.com/recordingApp/artist?name=danielaMercury



Page Controller – Example

- Mapping incoming requests (URLs) to corresponding controllers

```
<servlet>
    <servlet-name>artist</servlet-name>
    <servlet-class>actionController.ArtistController</servlet-class>
</servlet>
    <servlet-mapping>
    <servlet-name>artist</servlet-name>
    <url-pattern>/artist</url-pattern>
</servlet-mapping>
```

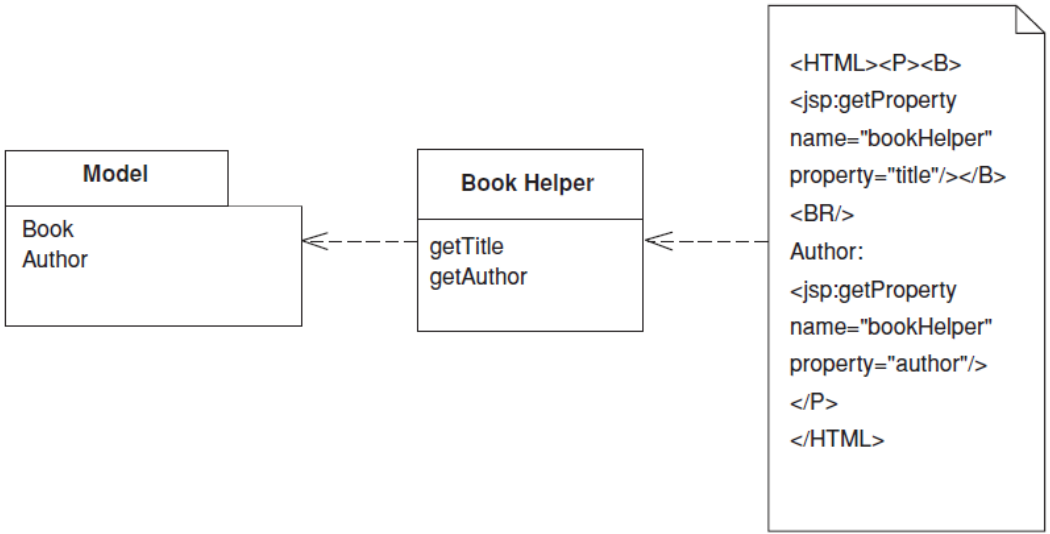
Page Controller – Example

- Class controller for handling /artist requests

```
1
2  class ArtistController {
3
4      //....
5      public void doGet(HttpServletRequest request, HttpServletResponse response)
6          throws IOException, ServletException {
7      Artist artist = Artist.findNamed(request.getParameter("name"));
8      if (artist == null)
9          forward("/MissingArtistError.jsp", request, response);
10     else {
11         request.setAttribute("helper", new ArtistHelper(artist));
12     forward("/artist.jsp", request, response);
13     }
14     }
15 }
```

MVC – Template View

– *"Renders information into HTML by embedding markers in an HTML page"*



Template View – Embedding the Markers

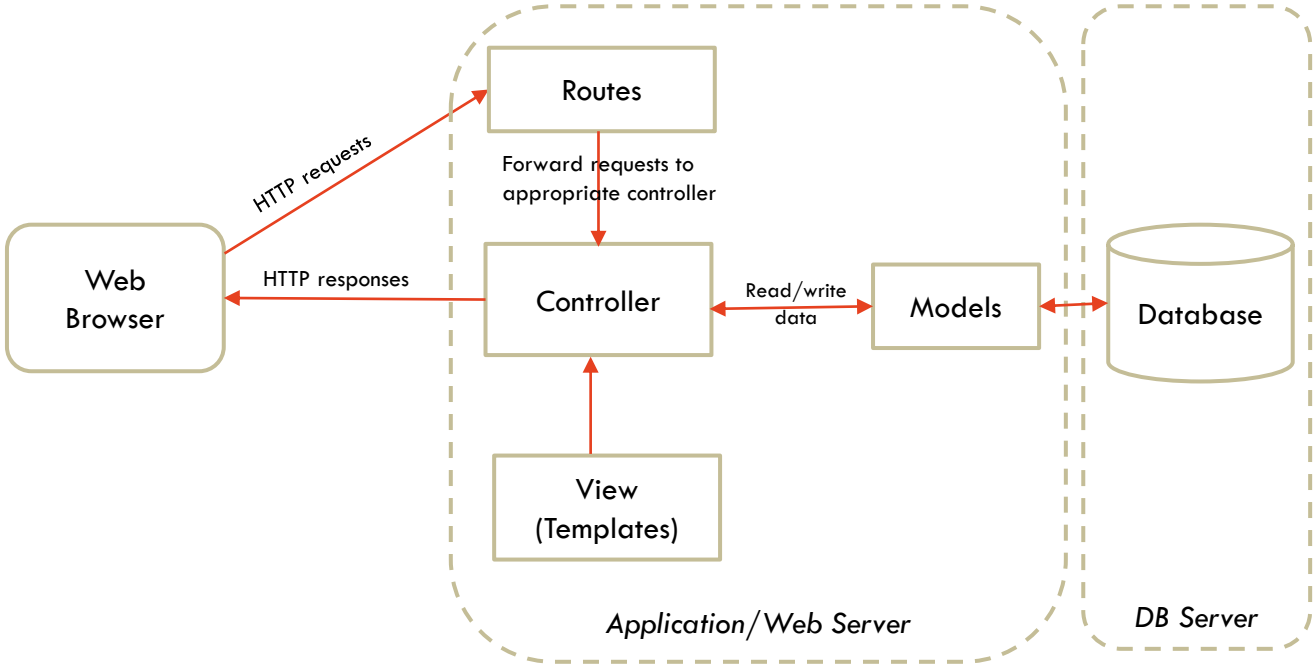
- HTML-like tags which will be treated differently (e.g., XML)
- Using special text markers not recognised by HTML editor but easier to understand and maintain
- Server pages (ASP, JSP, EJS) allows to embed programming logic (known as scriplets)
  - Not easy to understand when it becomes complex (programming logic + display)
  - E.g., conditional display, iteration

Template View – Helper Objects

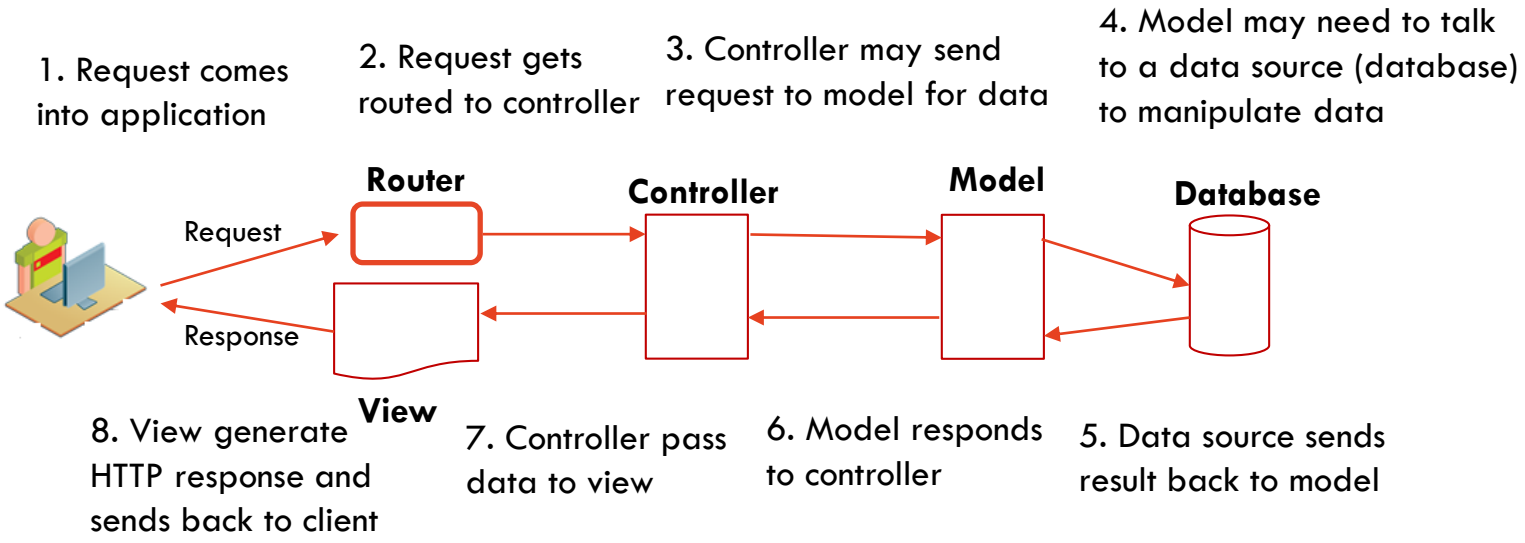
- Helper objects can be used to handle the programming logic where possible
- This can simplify the views by having those markers to be replaced with dynamic content
  - E.g., conditions and iterations logic in the helper object and called from the template view
- Developers can focus on the logic (helper objects) and designers on the views (template view)

MVC – Web Application

– Example of MVC design for Web application using Node.js/Express.js



MVC – Full Workflow (Example Node.js/Express.js)



- Data base related code should be put in model layer
- Controller should not have knowledge about the actual database
- Modularity allows easy switching between technologies
  - e.g. different view templates, different database management systems

References

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson.

- Martin Fowler (With contributions from David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford). 2003. *Patterns of Enterprise Applications Architecture*. Pearson.
- Alexander Shevts, Dive into Design Patterns. ebook
- Web Application Development (COMP5347) slides

W9 Tutorial: Practical Exercises/coding
W9 Lecture: Enterprise Design Patterns
Design Pattern Assignment

THE UNIVERSITY OF SYDNEY