

Software Design and Construction 2

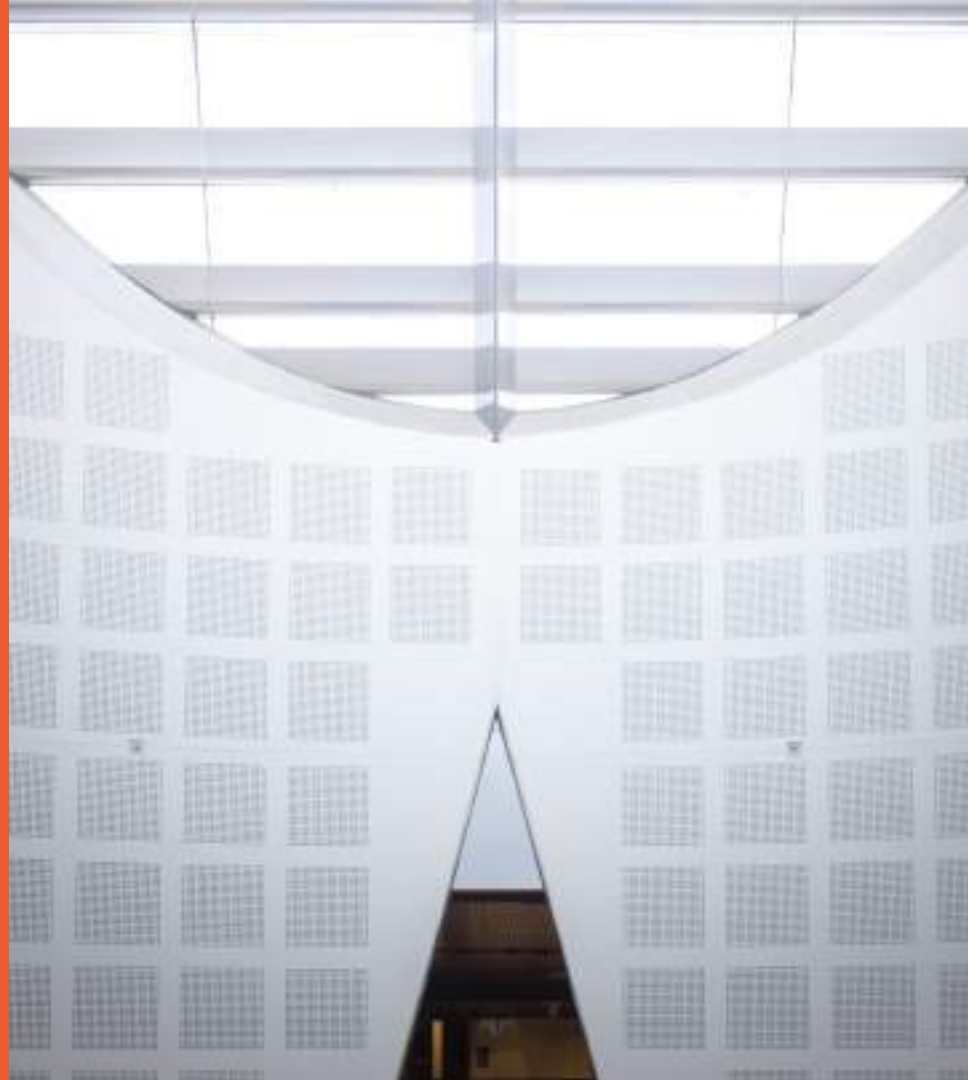
SOFT3202 / COMP9202

Enterprise Patterns

- Unit of Work
- Lazy Load
- Value Object

Dr. Rahul Gopinath

School of Computer Science



Enterprise Applications

- Payroll, patient records, shipping tracking, insurance, accounting, insurance
- Lots of Data, typically managed in DBMS
- Persistent for many years
- Concurrent data access with various presentation modes (web, cli, API)
- Components can change (and should be changeable, not monolithic)

Presentation tier

The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.



Logic tier

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.



GET LIST OF ALL
SALES MADE
LAST YEAR



ADD ALL SALES
TOGETHER

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.



QUERY



SALE 1
SALE 2
SALE 3
SALE 4



Database



Storage

Enterprise Applications: Mapping Objects to DB

- Data is modelled as objects in domain model
- Data is modelled as tables in relational DBMS
- How to map between in-memory objects and DB tables

Enterprise Applications: Transactions

- A segment of code executed as a whole (indivisible)
- Book a seat
- Transfer money
- Withdraw cash
- Borrow a book
- Should support ACID

Unit of Work: Object Relational Behavioural Pattern

Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems

Transactions in the client side.

Unit of Work: Object Relational Behavioural Pattern

Keeps track of every thing that can affect a DB (but not write)

- New objects created/existing objects updated
- Objects being read (consistency)

When you are done, alter the DB in a single transaction

Unit of Work
registerNew(object) registerDirty(object) registerClean(object) registerDeleted(object) commit rollback

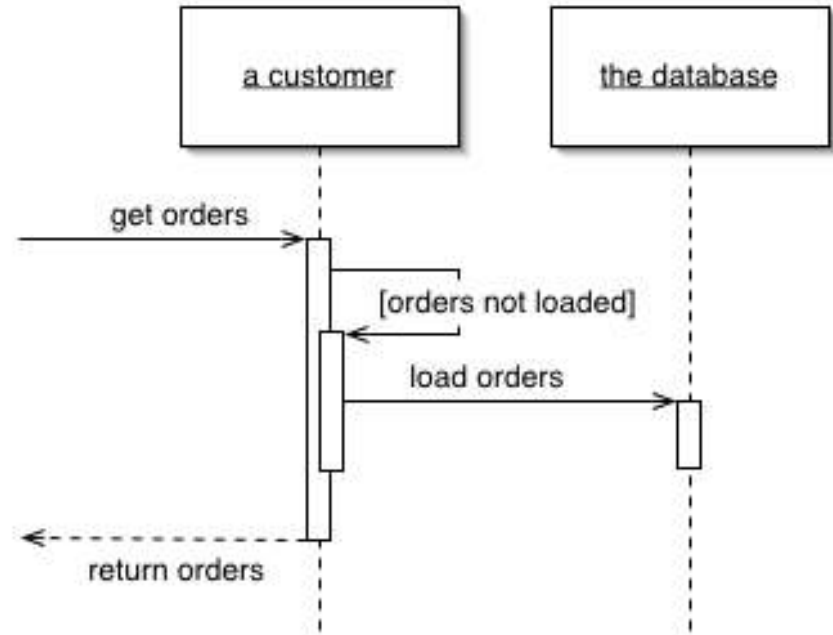
<https://www.martinfowler.com/eaCatalog/unitOfWork.html>

Unit of Work: Consequences

- All information in one place
- Reduce DB calls
- Provide transaction semantics over larger sequences

Lazy Load: Object Relational Behavioural Pattern

An object that doesn't contain all of the data you need but knows how to get it



Lazy Load: Object Relational Behavioural Pattern

- Useful if many DB calls are needed to populate an object.
- Implementations:
 - Lazy initialization
 - Check for a sentinel in each access, and load if sentinel (null) found.
 - Virtual Proxy
 - Use a light copy of object instead of the full until needed.
 - Value Holder
 - An object wrapper that loads the object on the first try
 - Ghost
 - A real object in partial state

Lazy Load: Consequences

- Allows one to defer loading of expensive objects
- Can mess with type checks (details may be unknown until loading)
- Can lead to more data base access than needed

Value: Object Relational Behavioural Pattern

A small simple object whose equality isn't based on identity (typically immutable)

- Custom equals implementation based on the domain
- E.g. money, date range

Value Object: Consequences

- Can improve performance

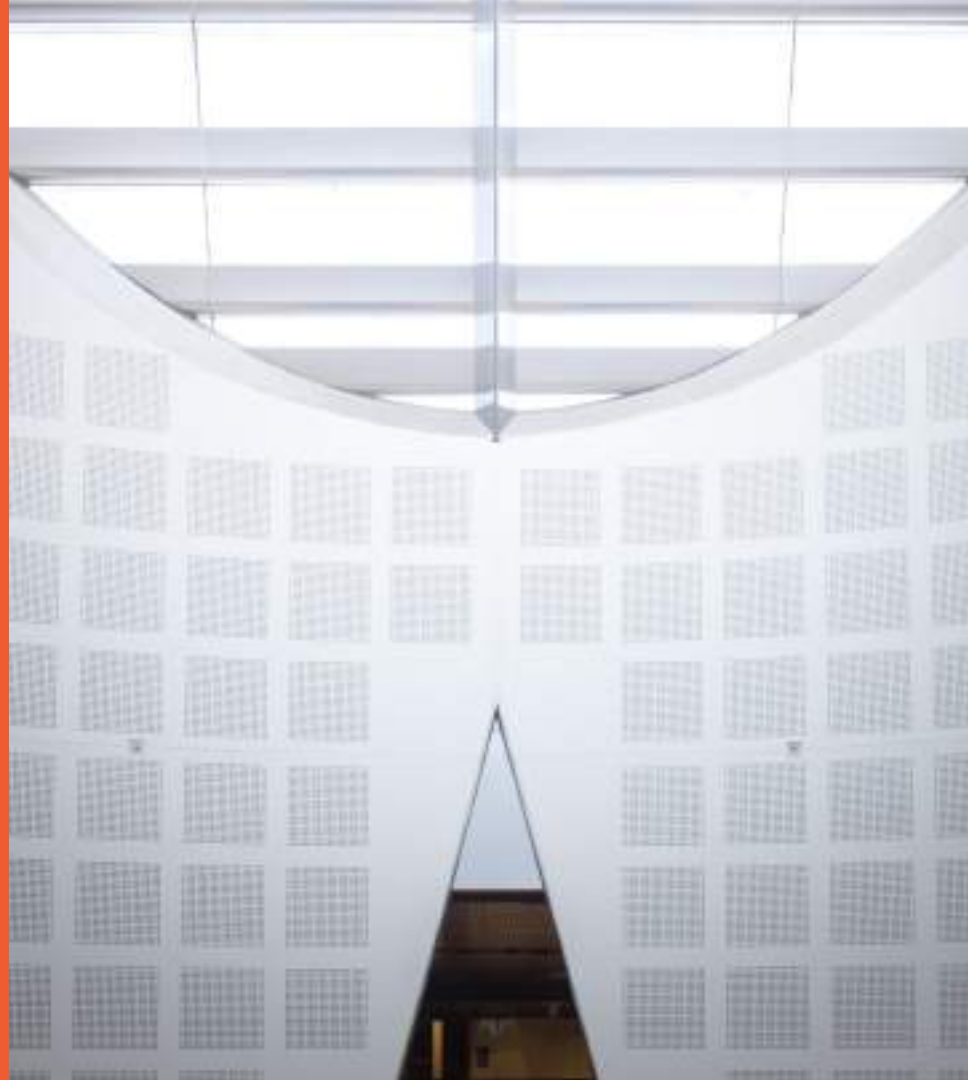
Software Construction and Design 2

SOFT3202 / COMP9202

GoF Design Patterns (Visitor)

Dr. Rahul Gopinath

School of Information Technologies

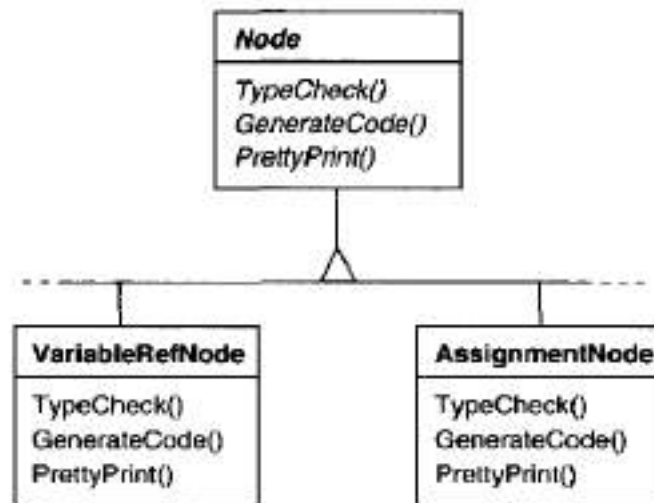


Visitor



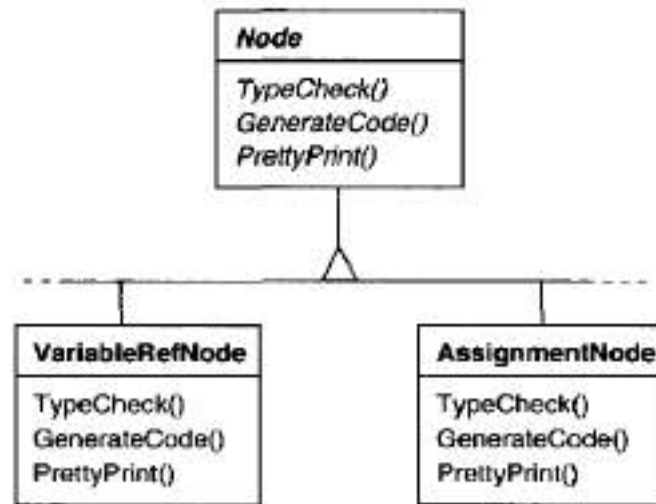
Motivation – A Compiler

- 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)



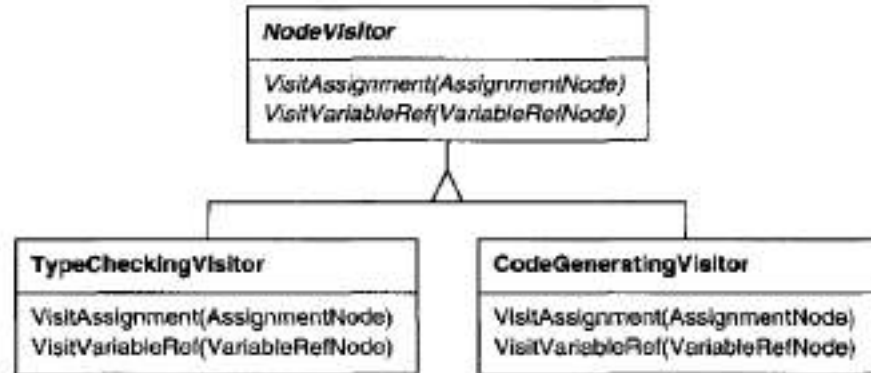
Motivation – A Compiler

- 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

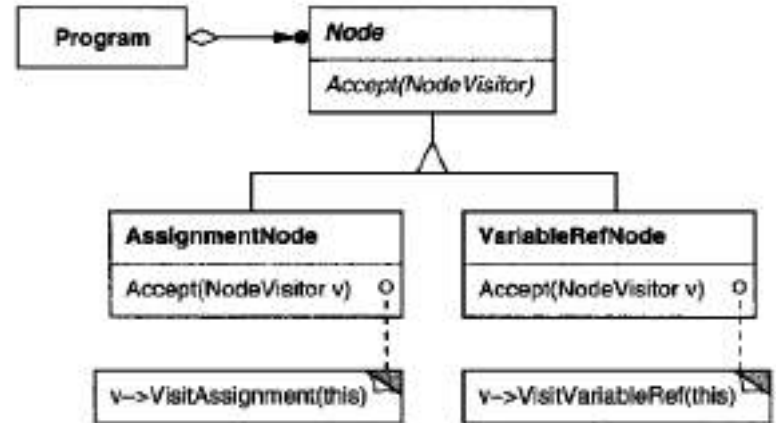


A Compiler Application – Improved Design

– .



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

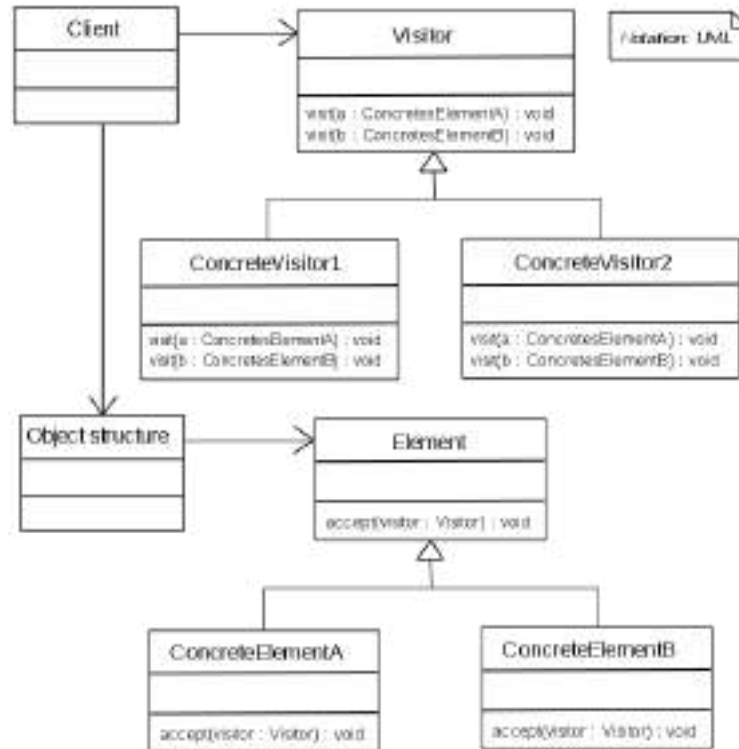
Visitor Pattern – How it Works

- 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 (Separate algorithms from the objects on which they operate.)
- 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>

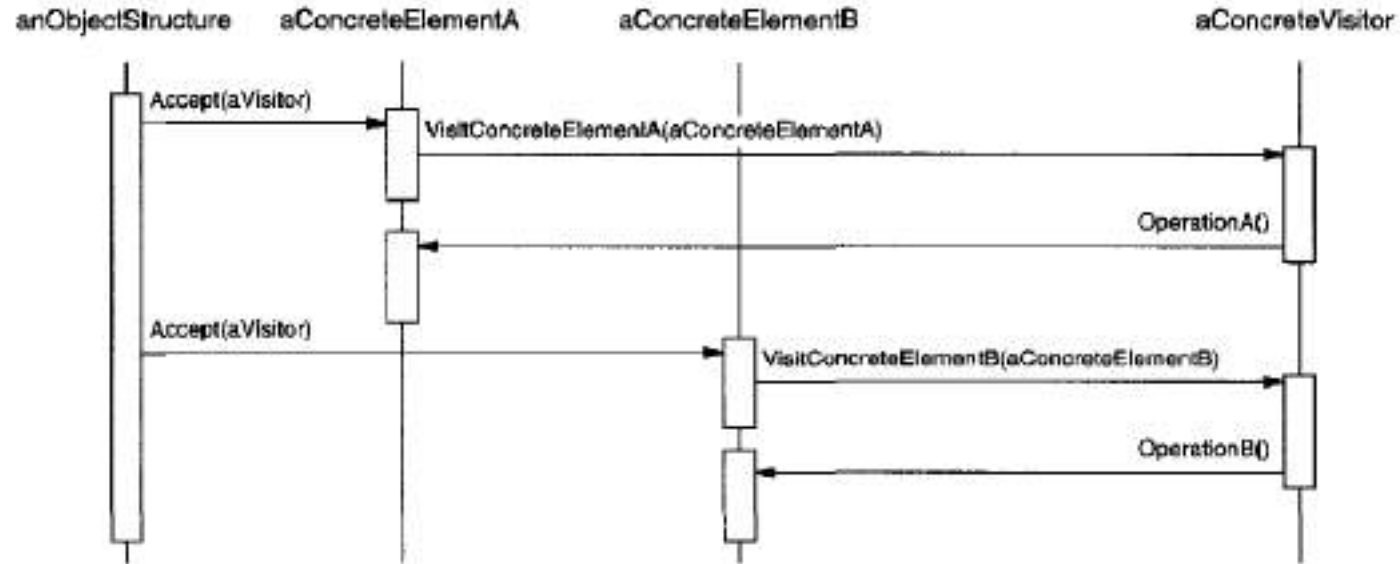
Visitor Pattern – Participants

- 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

- ConcreteElement (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

```
public interface Visitor {
    void visit(ConcreteElementA elementA);
    void visit(ConcreteElementB elementB);
}
```

```
public interface Element {
    void accept(Visitor visitor);
}
```

```
public class ConcreteVisitor1 implements Visitor {
    public void visit(ConcreteElementA elementA) {
        System.out.println("ConcreteVisitor1 visited ConcreteElementA");
    }

    public void visit(ConcreteElementB elementB) {
        System.out.println("ConcreteVisitor1 visited ConcreteElementB");
    }
}
```

```
public class ConcreteElementA implements Element {
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
```

```
public class ConcreteVisitor2 implements Visitor {
    public void visit(ConcreteElementA elementA) {
        System.out.println("ConcreteVisitor2 visited ConcreteElementA");
    }

    public void visit(ConcreteElementB elementB) {
        System.out.println("ConcreteVisitor2 visited ConcreteElementB");
    }
}
```

```
public class ConcreteElementB implements Element {
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        List<Element> elements = new ArrayList<>();
        elements.add(new ConcreteElementA());
        elements.add(new ConcreteElementB());

        Visitor visitor1 = new ConcreteVisitor1();
        Visitor visitor2 = new ConcreteVisitor2();

        for (Element element : elements) {
            element.accept(visitor1);
            element.accept(visitor2);
        }
    }
}
```

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 *ConcreteElement* as its argument to allow the visitor to access the interface of *ConcreteElement* directly
 - *ConcreteVisitor* 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

```

// ShapeVisitor interface
public interface ShapeVisitor {
    void visit(Circle circle);
    void visit(Rectangle rectangle);
}

// Shape interface
public interface Shape {
    void accept(ShapeVisitor visitor);
}

// Circle class
public static class Circle implements Shape {
    private int radius;

    public Circle(int radius) {
        this.radius = radius;
    }

    public int getRadius() {
        return radius;
    }

    @Override
    public void accept(ShapeVisitor visitor) {
        visitor.visit(this);
    }
}

// Rectangle class
public static class Rectangle implements Shape {
    private int width, height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }

    @Override
    public void accept(ShapeVisitor visitor) {
        visitor.visit(this);
    }
}

// AreaCalculatorVisitor class
public static class AreaCalculatorVisitor implements ShapeVisitor {
    @Override
    public void visit(Circle circle) {
        double area = Math.PI * Math.pow(circle.getRadius(), 2);
        System.out.println("Area of Circle: " + area);
    }

    @Override
    public void visit(Rectangle rectangle) {
        int area = rectangle.getWidth() * rectangle.getHeight();
        System.out.println("Area of Rectangle: " + area);
    }
}

// PerimeterCalculatorVisitor class
public static class PerimeterCalculatorVisitor implements ShapeVisitor {
    @Override
    public void visit(Circle circle) {
        double perimeter = 2 * Math.PI * circle.getRadius();
        System.out.println("Perimeter of Circle: " + perimeter);
    }

    @Override
    public void visit(Rectangle rectangle) {
        int perimeter = 2 * (rectangle.getWidth() + rectangle.getHeight());
        System.out.println("Perimeter of Rectangle: " + perimeter);
    }
}

public class DoubleDispatchDemo {

    public static void main(String[] args) {
        List<Shape> shapes = new ArrayList<>();
        shapes.add(new Circle(5));
        shapes.add(new Rectangle(4, 5));

        ShapeVisitor areaCalculator = new AreaCalculatorVisitor();
        ShapeVisitor perimeterCalculator = new PerimeterCalculatorVisitor();

        System.out.println("Calculating areas:");
        for (Shape shape : shapes) {
            shape.accept(areaCalculator);
        }

        System.out.println("\nCalculating perimeters:");
        for (Shape shape : shapes) {
            shape.accept(perimeterCalculator);
        }
    }
}

```

double dispatch is used when the accept() method is called on a Shape object with a ShapeVisitor object as an argument. The accept() method, in turn, calls the visit() method on the visitor object, passing this (the current Shape object) as an argument. This two-step process ensures that the correct method is called based on both the type of the Shape object and the type of the ShapeVisitor object.

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


```
// FileSystemElement interface
public interface FileSystemElement {
    void accept(FileSystemVisitor visitor);
}

// File class
public static class File implements FileSystemElement {
    private String name;

    public File(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public void accept(FileSystemVisitor visitor) {
        visitor.visit(this);
    }
}

// Directory class
public static class Directory implements FileSystemElement {
    private String name;
    private List<FileSystemElement> children;

    public Directory(String name) {
        this.name = name;
        this.children = new ArrayList<>();
    }

    public String getName() {
        return name;
    }

    public void add(FileSystemElement element) {
        children.add(element);
    }

    public List<FileSystemElement> getChildren() {
        return children;
    }

    @Override
    public void accept(FileSystemVisitor visitor) {
        visitor.visit(this);
    }
}
```

```
// FileSystemVisitor interface
public interface FileSystemVisitor {
    void visit(File file);
    void visit(Directory directory);
}

// FileSystemTraversalVisitor class
public static class FileSystemTraversalVisitor implements FileSystemVisitor {
    private int indent = 0;

    @Override
    public void visit(File file) {
        printIndent();
        System.out.println("File: " + file.getName());
    }

    @Override
    public void visit(Directory directory) {
        printIndent();
        System.out.println("Directory: " + directory.getName());
        indent++;
        for (FileSystemElement element : directory.getChildren()) {
            element.accept(this);
        }
        indent--;
    }

    private void printIndent() {
        for (int i = 0; i < indent; i++) {
            System.out.print(" ");
        }
    }
}
```

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

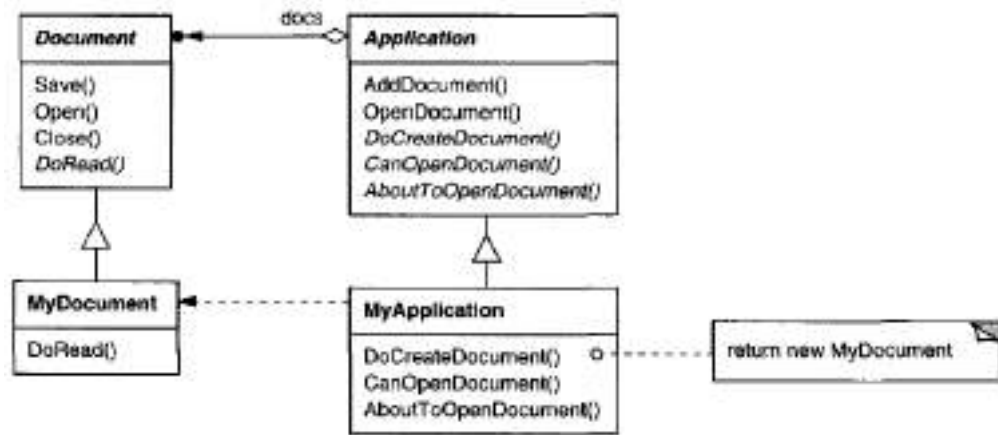
Template Method Pattern (GoF)

Class behavioural



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)



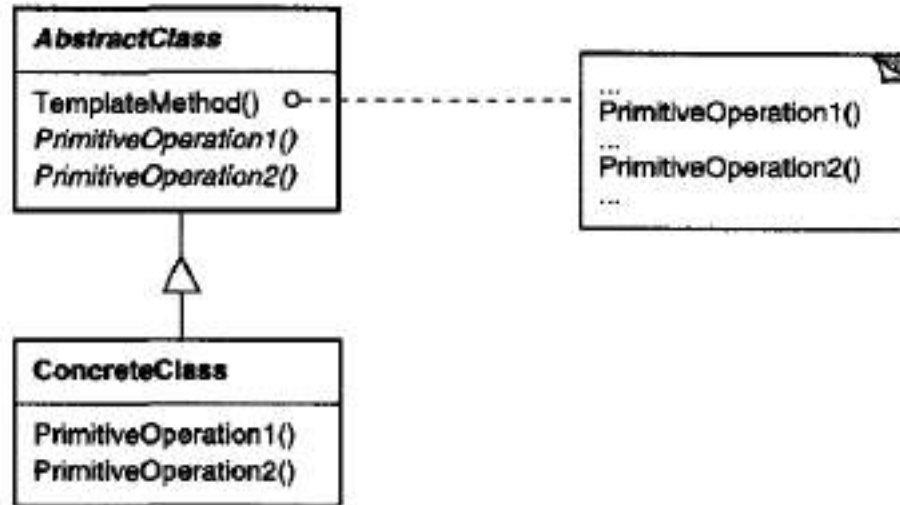
Motivating Scenario – Template Method

- 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

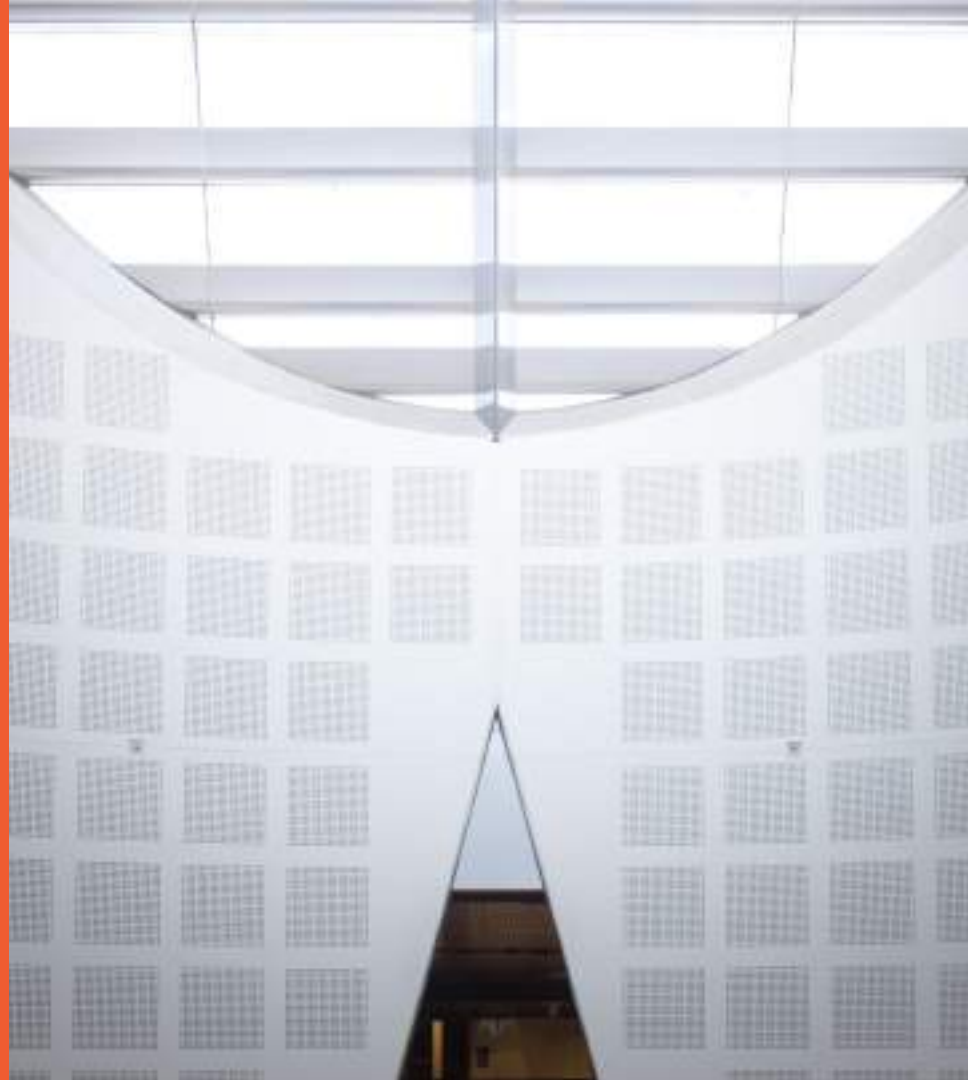
Software Construction and Design 2

SOFT3202 / COMP9202

Enterprise Design Patterns
(Concurrency)

Dr. Rahul Gopinath

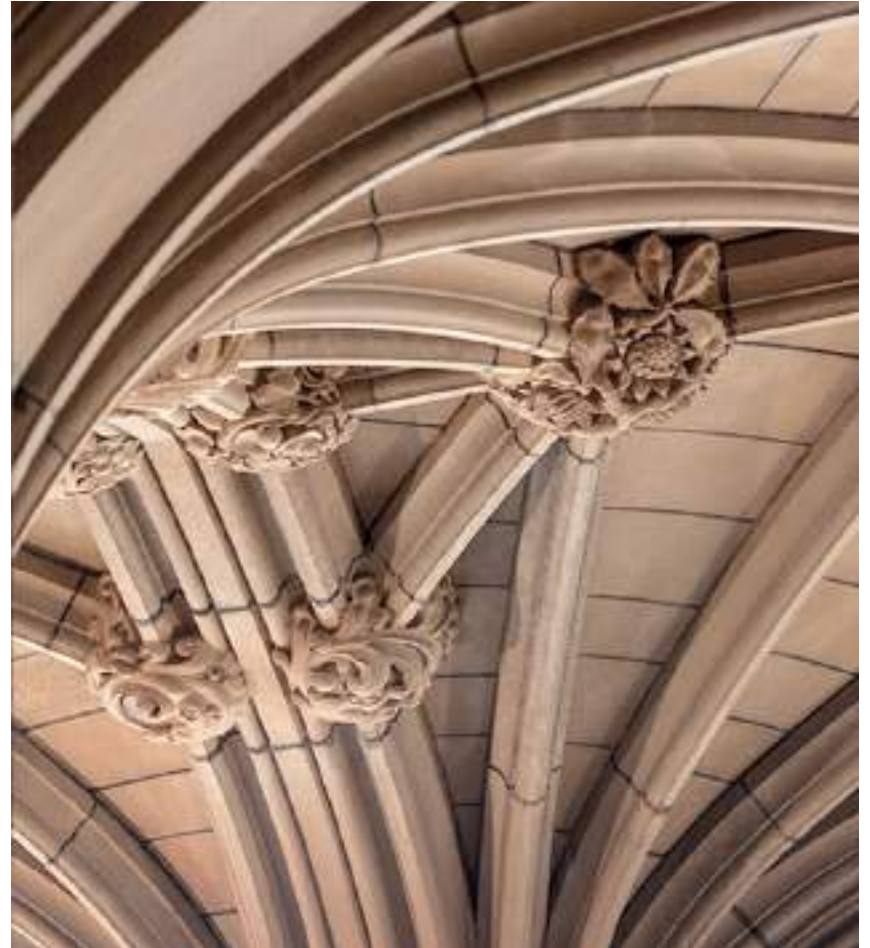
School of Information Technologies



Agenda

- Introduction
 - Concurrency in Enterprise Applications
- Enterprise Application Design Patterns
 - Lock
 - Optimistic Lock
 - Pessimistic Lock
 - Thread Pool
 - Active Objects

Enterprise Applications



Enterprise Applications

- Manipulate lots of persistent data
 - DBMS (relational DB and NoSQL)
- Concurrent data access and manipulation
 - Design with concurrent programming thinking not only transactions management systems
- Data access and manipulation via different views and contexts
- Integrate with other enterprise applications
 - Legacy and modern ones
 - Using different software and hardware technologies

State

- All imperative programming involves altering state (the values linked to names, for example, variables)
- In enterprise applications we distinguish several varieties of data that has state which is important
 - Infrastructure state
 - Resource state
 - Session state
 - Local computation state
 - Other

Type of States

- Infrastructure: provides general support for computation (connection pool)
- Resource: data about the real-world domain (salary, manger)
- Session: set of related operations over a period of time (business transaction)
 - Operation may change values of resource state
 - Client session state, server session state or database session state
- Local computation state: temp variables created in programs

Lifetime of State

- Resource state lasts from creation to explicit deletion, through many sessions
- Session state lasts from start of session to completion, through many operations
- Local computation state lasts during one operation

Sharing of State

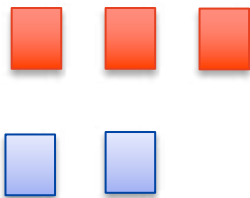
- Resource state means something about the domain
 - It will be accessed by many sessions
 - Possibly at the same time
- Session state is used by different steps of the session
 - Usually, one after another (but with gaps)
- Local computation state is not shared

Concurrency & Parallelism

- Concurrency: When a system can support execution of two tasks at the same time in some fashion (interleaving is OK).



- Parallelism: When the tasks can run simultaneously. Can support concurrency.



Concurrency

- Techniques and mechanisms that enable several parts of a program be executed simultaneously without affecting the outcome
 - Several computations that share data run at the same time
 - Concurrent execution of program tasks

Why Concurrency?

- Allow simultaneous access/usage of enterprise applications
- Improve performance (speed)!
 - Allow for parallel execution of the concurrent parts
 - Doing more than one operation at a time
 - Processing while waiting for I/O
 - Using multiple processes

Mechanisms for Concurrency

- Multiple machines/nodes
- One machine with separate processes
- Threads within a process

Example: preparing a salad



Tasks to do:



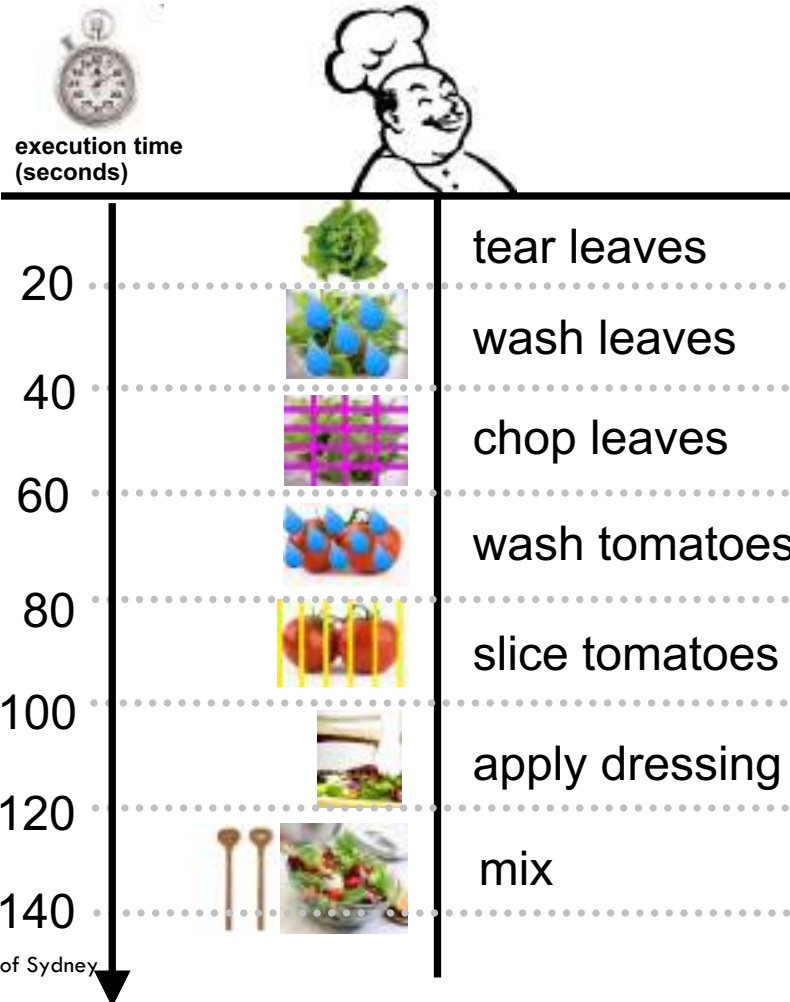
- tear leaves
- wash leaves
- chop leaves



- wash
- slice



- apply dressing
- mix

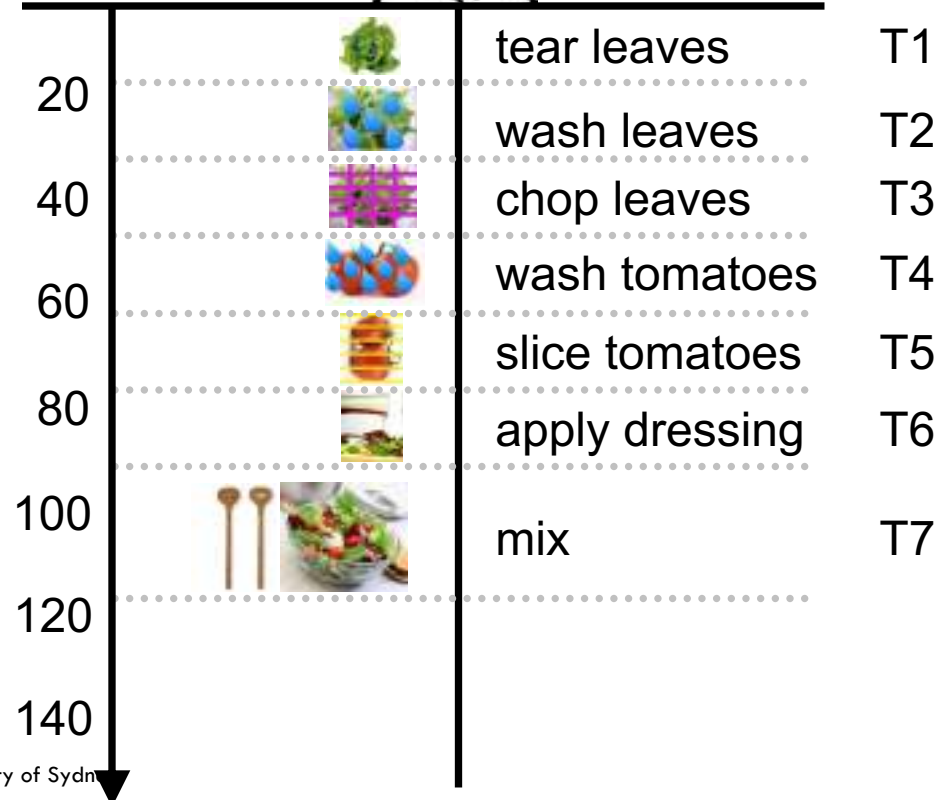


Single Chef Salad (sequential)

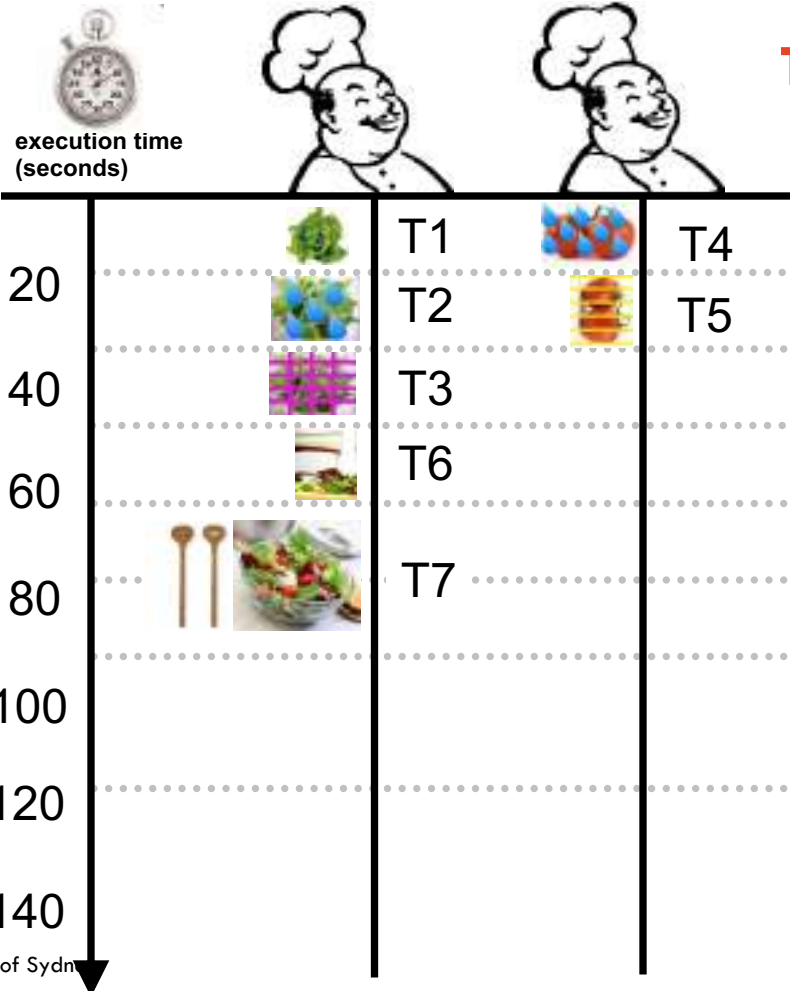
- T1 ■ Preparing a salad consists of **7 tasks** (T1-T7).
- T2 ■ A single chef prepares a salad in processing the 7 tasks **sequentially** (one after the other).
- T3 ■ It takes a single chef 142 seconds to prepare a salad.
- T4 ■ We can speed up the process by making the chef work **faster**.
- T5
- T6
- T7



Fast Single Chef Salad (sequential)



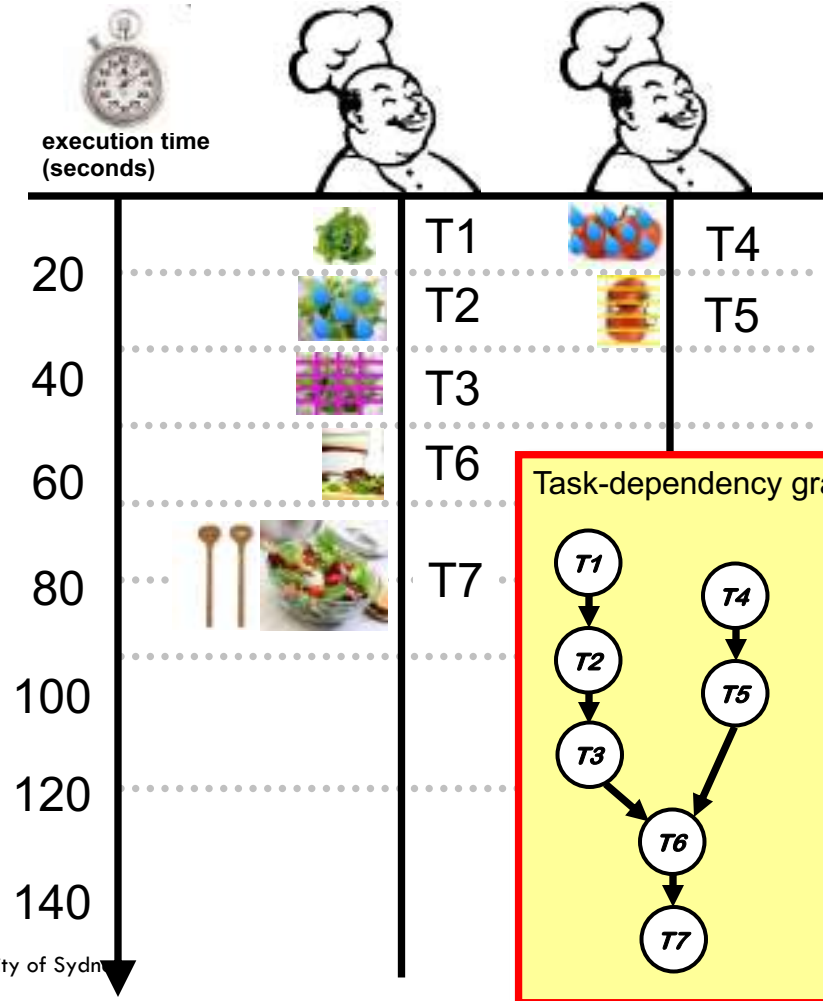
- If we can convince the Chef to finish every task in 18 instead of 20 seconds, we will reduce the process to $7 \times 18 = 126$ seconds.
- The chef agrees for Tasks T1-T6.
- However, he cannot speed up Task T7 without spoiling the entire kitchen.
- He refuses to work any faster than that.
- We can speed up the process by bringing in an additional Chef that works **in parallel** to Chef 1.



Two Chefs in Parallel Salad

- Chef 2 washes the tomatoes (Task T4) while Chef 1 tears the lettuce (Task T1).
- Task T1 is processed in parallel to Task T4. Processing tasks in parallel is called **task-parallelism**.
- Another form of task-parallelism occurs between tasks T2 and T5.
- There exist **dependencies** between tasks:
 - Vegetables must be washed *before* they are chopped/sliced.
 - Mixing (T7) can only be done *after* the dressing has been applied (T6).
 - etc. (see next slide)

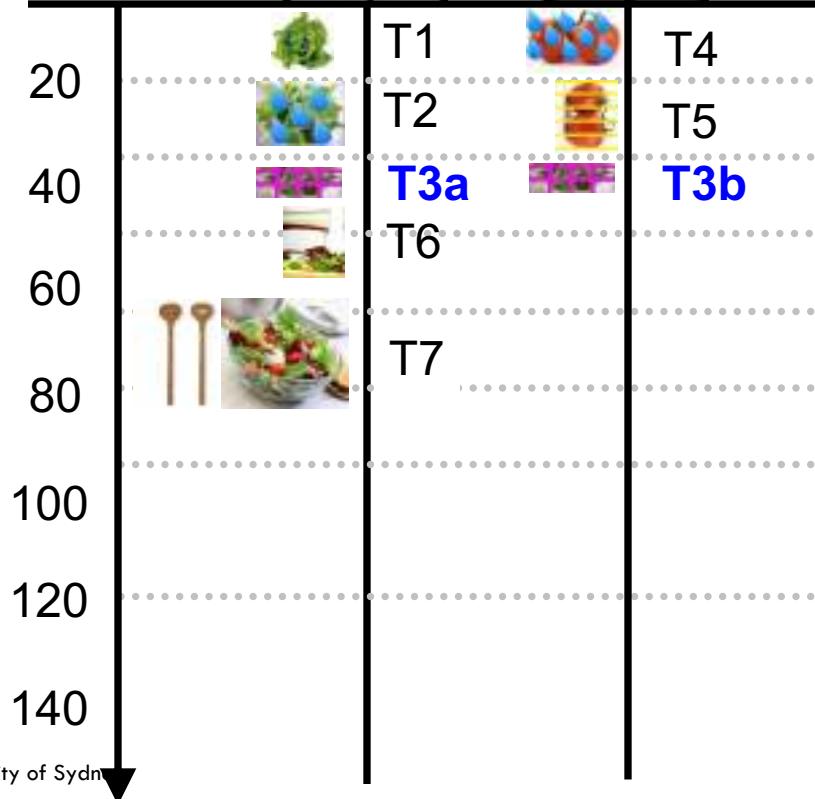
Two Chefs in Parallel Salad



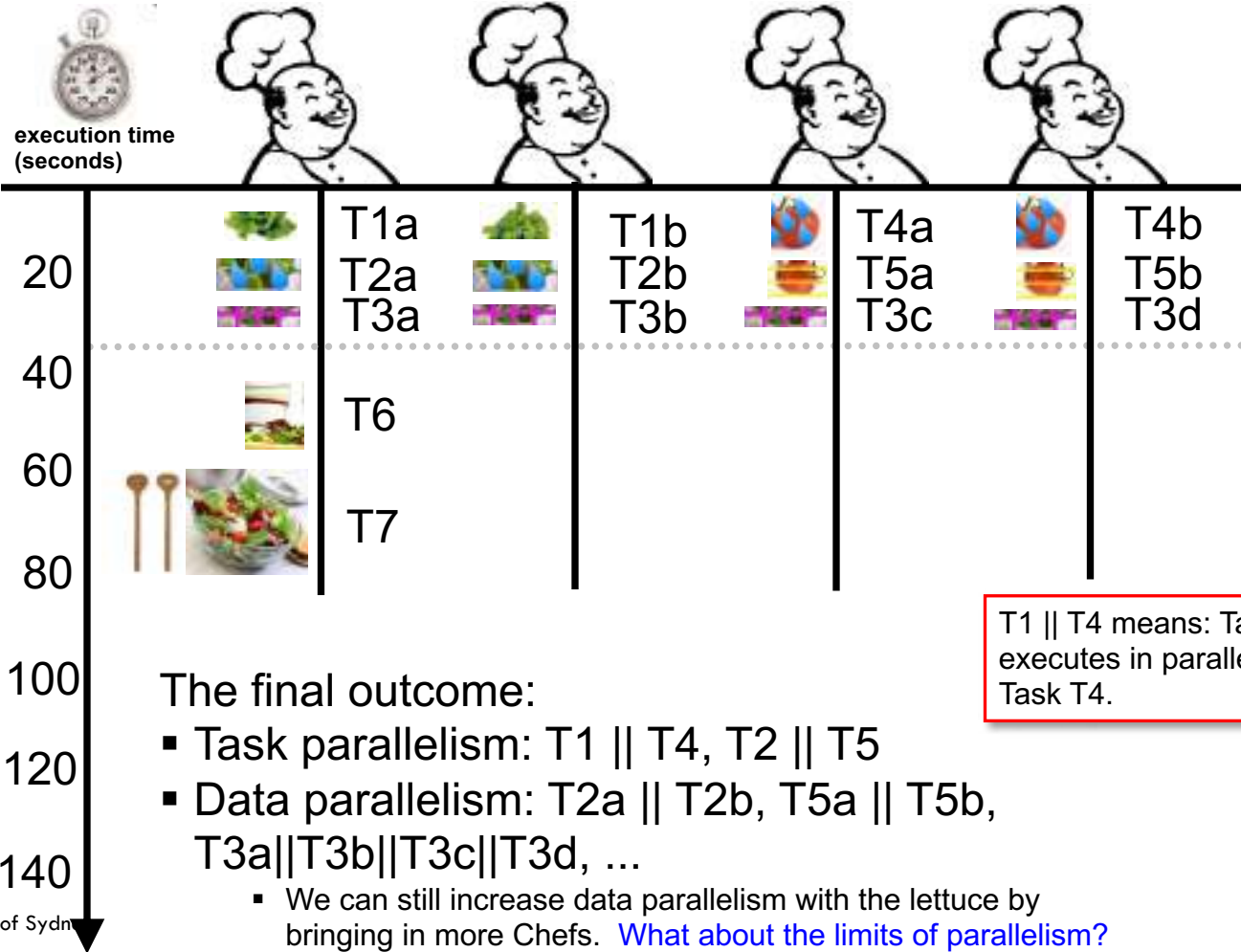
- Dependencies between tasks do not allow further task-parallelism with this example.
- A **task-dependency graph** shows **dependencies** $T_a \rightarrow T_b$ between tasks.
 - directed acyclic graph (DAG)
 - graph nodes represent tasks
 - directed edge indicates that T_b can only be processed after T_a has completed.
 - Task T_b depends on task T_a
 - See example graph to the left.



Two Chefs in Parallel (task and data parallelism)



- Dependencies between tasks do not allow further task-parallelism with this example.
- However, we can have several Chefs work in parallel on the “data” of a task.
- This form of parallelism is called **data-parallelism**.
 - Example1: Chef 1 and Chef 2 chop one half of the lettuce each (Tasks **T3a** and **T3b**).
 - Example2: On the next slide more data-parallelism is introduced.



Definitions

Task: a computation that consists of a sequence of instructions. The computation is a *distinct* part of a program or algorithm. (That is, programs and algorithms can be **de-composed** into tasks).

- Examples (salad): “washing lettuce”, “initialize data-structures”, “sort array”, ...
- Examples (software): initialize data-structures, load data from disk, search, sort, display results to the user, ...

Task-parallelism: parallelism achieved from executing different tasks at the same time (i.e., in parallel).

- The amount of task-parallelism in a program is limited by the number of tasks in the program, and by the dependencies between tasks.
- Tasks can sometimes be de-composed into sub-tasks to increase task-parallelism.
- Task-parallelism is sometimes referred to as **functional parallelism**, because different functionality (or parts) of the program execute in parallel.
- The independent program parts can be single statements, basic-blocks, loop-bodys, or function calls.

Definitions (cont.)

Data-parallelism: performing the same task to different data-items at the same time.

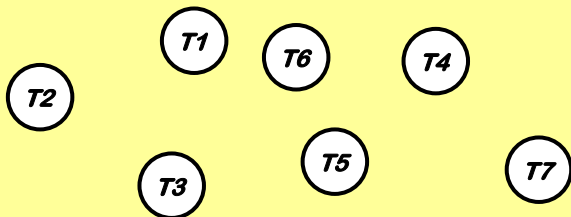
- Example (salat): 2 Chefs slicing 1 tomato each. (tomato = data, slicing ~ task).
- Example (software): two ``workers" searching one half of an array each.
- Unlike task-parallelism, data-parallelism is **not limited** by the properties (number of tasks) of the underlying program.
- Data-parallelism thus scales better to a higher number of cores.

Definitions (cont.)

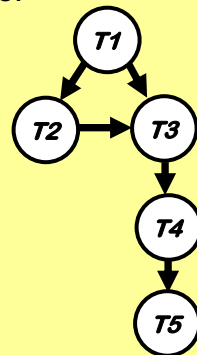
Dependencies: an execution order between two tasks T_a and T_b . T_a must complete before T_b can execute. Notation: $T_a \rightarrow T_b$.
Dependencies limit the amount of parallelism in an application.

Example task dependency graphs:

no dependencies, maximum parallelism:



dependencies, no parallelism possible:

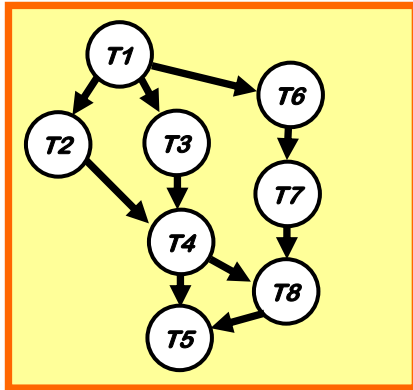


Definitions (cont.)

Dependencies impose a **partial ordering** on the tasks:

Two tasks **T_a** and **T_b** can execute in parallel iff

- 1) there is no path in the dependence graph from **T_a** to **T_b**
- 2) there is no path in the dependence graph from **T_b** to **T_a**

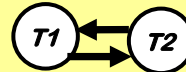


Dependency is transitive:

$T_a \rightarrow T_b$ and $T_b \rightarrow T_c$ implies $T_a \rightarrow T_c$

Say: T_a must complete before T_b , and T_b must complete before T_c , therefore T_a must complete before T_c .


What about this?



Concurrency Problems

- When interleaved computations have any shared state that they operate on, things can go badly wrong
 - Interference between programs/transactions
- Conflicts over shared resources
 - Cannot sell the same seat twice
- The state on which the computation depends can be changed unexpectedly
 - By another thread, between steps of the computation

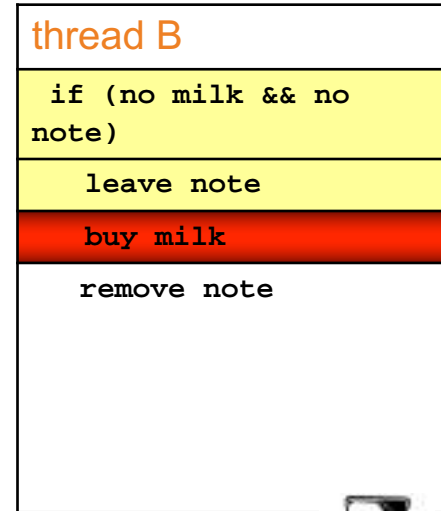
Example 1: Too much milk

Time	You	Your flat mate
5:00	Arrive at home	
5:05	Look in fridge, no milk	
5:10	Leave for supermarket	
5:15		Arrive at home
5:20	Arrive at supermarket	Look in fridge, no milk
5:25	Buy milk	Leave for supermarket
5:30	Arrive at home,	
5:35	Put milk in fridge	Arrive at supermarket
5:40		Buy milk
5:45:00		Arrive at home, open fridge
5:45:01		Oh no!

- We need to **synchronize** parallel activities!



The Need for Locks

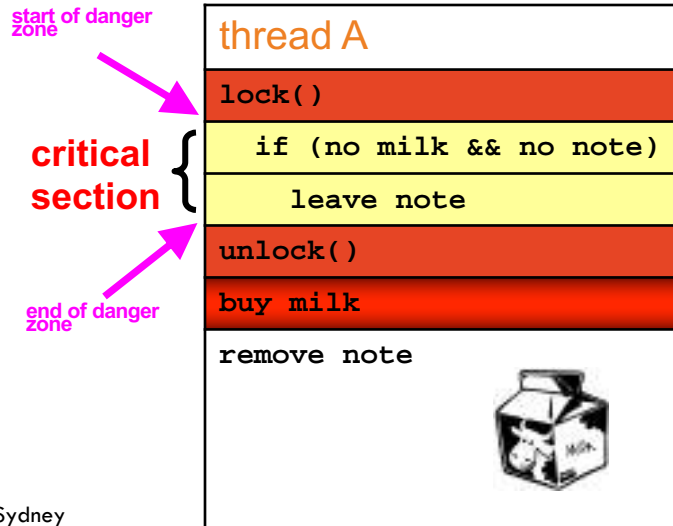


The University of Sydney – Does this solve the problem?

Mutual Exclusion

Only one thread must be in the **critical section** at any time.

- Once a thread is in the critical section, no other thread can enter that critical section until the first thread has left that critical section.
- Serializes access to critical section.



General Rules

- If instances of two code segments can interleave,
 - and both access the same item of shared state
 - and at least one of the instances modifies that item
- Then some mechanism is needed to avoid concurrency problems
- Also, same applies for two instances of the same code segment
- Do not try to reason out why some particular case is safe, just use proper protective measures

Concurrency Control

- For realistic applications, concurrency and shared data are both unavoidable
- So need mechanisms to restrict the interleaving and prevent bad things happening
 - Without reducing to a single-threaded computation
 - So allow harmless concurrency, but not harmful concurrency

What is a Transaction?

- A collection of one or more operations on one or more data resources, which reflect a *discrete unit of work*
 - In the real world, this happened (completely) or it didn't happen at all (Atomicity)
 - Can be read-only (e.g. RequestBalance), but typically involves data modifications
- Database technology allows application to define a transaction
- A segment of code that will be executed as a whole
 - System infrastructure should prevent problems from failure and concurrency
- Transaction should be chosen to cover a complete, indivisible business operation

What is a Transaction?

- Examples
 - Transfer money between accounts
 - Purchase a group of products
- Why are transactions and transaction processing interesting?
 - They make it possible to build high performance, reliable and scalable computing systems
 - Transactions aim to maintain integrity despite failures

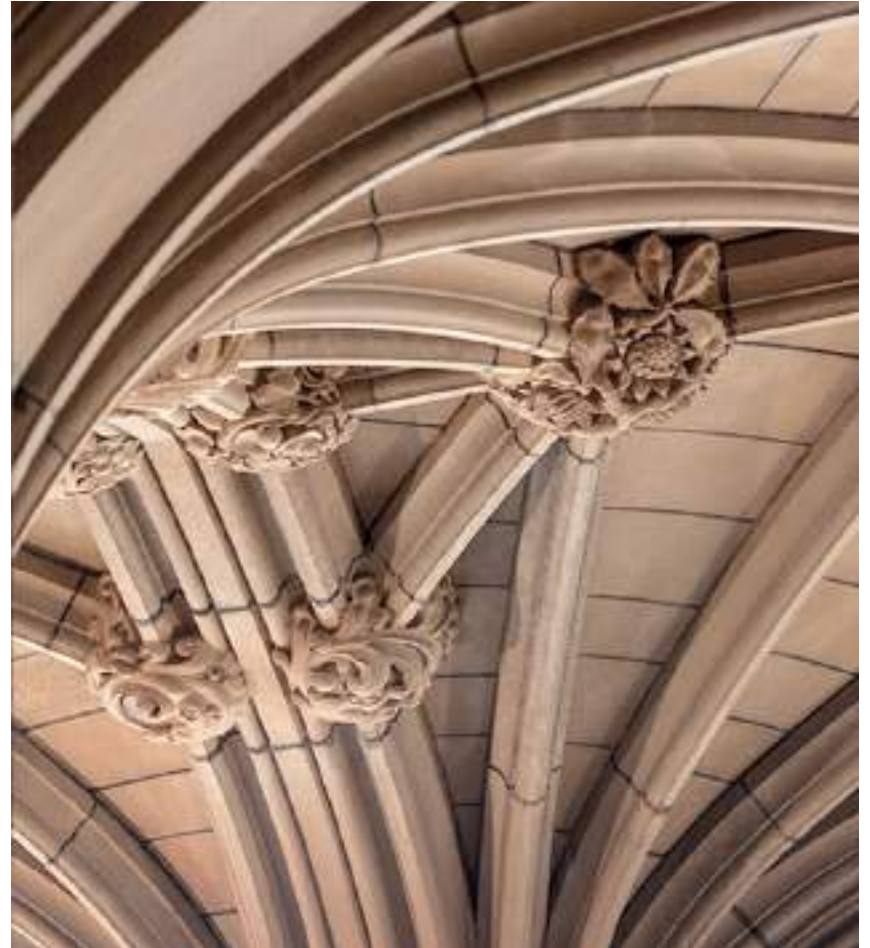
Transaction – ACID Test

- The transaction support should pass the ACID test
 - Atomic (all, or nothing)
 - Either transaction commits (all changes are made) or else it aborts (no changes are made)
 - Abort may be application-requested rollback, or it may be system initiated
 - Consistent
 - Isolated (no problems from concurrency; details are complicated)
 - DBMS provides choices of “isolation level”
 - Usually good performance under load is only available with lower isolation
 - But lower isolation level does not prevent all interleaving problems
 - Durable (changes made by committed transactions are not lost in failures)

Transaction Systems

- Efficiently and quickly handle high volumes of requests
- Avoid errors from concurrency
- Avoid partial results after failure
- Grow incrementally
- Avoid downtime
- And ... never lose data

Concurrency in Java



Concurrency in Java

- Java can run multiple threads of execution inside application
- Thread is like a separate CPU executing application but in SW
- Reasons
 - Better utilization of CPUs
 - Better responsiveness / fairness for application
- Recall/Learn
 - Thread / Runnable class
 - How to create threads / manipulate threads
 - Enforce critical sections with synchronized

Creating a New Thread

- Subclass thread

```
public class TestThread extends Thread {  
    public void run(){  
        System.out.println("TestThread running");  
    }  
}
```

- Create a new thread

```
Thread thread = new TestThread();
```

- Start/Spawn a new thread

```
thread.start();
```

- Also, runnable interface available

Thread Methods

- Current thread: `thread.currentThread()`
- Get thread name: `thread.getName()`
- Pause a thread: `thread.pause(3); // 3ms`
- Example:

```
public class ThreadExample {
    public static void main(String[] args){
        System.out.println(Thread.currentThread().getName());
        for(int i=0; i<5; i++){
            new Thread(" " + i){
                public void run(){
                    System.out.println("T:" + getName());
                    this.pause(3);
                }
            }.start();
        }
    }
}
```

Synchronization / Critical Section

- Use synchronized construct
 - only one thread allowed in
- Example:

```
public class Account {  
    private int acc = 0;  
    public void add(int val){  
        // critical section  
        synchronized(this){  
            this.acc += val;  
        }  
    }  
}
```

// critical section

Concurrency Patterns

Enterprise Application Patterns



Enterprise Applications – Concurrency Patterns

- Design patterns concerned with the multi-threaded programming paradigm including:
 - Active Object
 - Thread pool
 - Read write lock
 - Optimistic Lock
 - Pessimistic lock
 - Others

Active Object

- Separate Method Invocation from Method Execution
- Allows the system to remain responsive, even during time-consuming operations
- Provides concurrency and asynchronous processing

Active Object: Components

1. **Proxy:** Provides an interface for clients to interact with the Active Object. It receives client requests and forwards them to the Scheduler.
2. **Scheduler:** Manages and prioritizes client requests and dispatches them to the corresponding Servant.
3. **Servant:** Performs the actual work or method execution.
4. **Activation List (Dispatch Queue):** A queue that holds client requests until they are processed by the Servant.

Active Object: Components

```
// Proxy
class Proxy {
    private final Scheduler scheduler;

    public Proxy(Scheduler scheduler) {
        this.scheduler = scheduler;
    }

    @Override
    public Future<String> doTask(String data) {
        return scheduler.schedule(data);
    }
}
```

```
// Servant - Performs the actual work
class Servant {
    public String processData(String data) {
        // Perform a long-running operation
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "Processed: " + data;
    }
}
```

```
// Scheduler - Schedules the work
class Scheduler {
    private final ExecutorService executorService;
    private final Servant servant;

    public Scheduler(Servant servant) {
        this.servant = servant;
        this.executorService = Executors.newSingleThreadExecutor();
    }

    public Future<String> schedule(String data) {
        return executorService.submit(() -> servant.processData(data));
    }
}
```

```
Servant servant = new Servant();
Scheduler scheduler = new Scheduler(servant);
Proxy proxy = new Proxy(scheduler);

Future<String> result = proxy.doTask("Sample Data");
System.out.println("Request submitted, waiting for result...");

System.out.println(result.get());
```


Active Object: Components

```
// Proxy
class Proxy {
    private final Scheduler scheduler;

    public Proxy(Scheduler scheduler) {
        this.scheduler = scheduler;
    }

    @Override
    public void doTask(String data, OpCallback callback) {
        scheduler.schedule(data, callback);
    }
}

// Scheduler
class Scheduler {
    private final ExecutorService eService; // activation list
    private final Servant servant;

    public Scheduler(Servant servant) {
        this.servant = servant;
        this.eService = Executors.newSingleThreadExecutor();
    }

    public void schedule(String data, OpCallback callback) {
        executorService.submit(() -> {
            String result = servant.processData(data);
            callback.onComplete(result);
        });
    }
}
```

The University of Sydney

```
// Callback interface
interface OpCallback {
    void onComplete(String result);
}
```

```
// Servant
class Servant {
    public String processData(String data) {
        // Perform a long-running operation
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "Processed: " + data;
    }
}
```

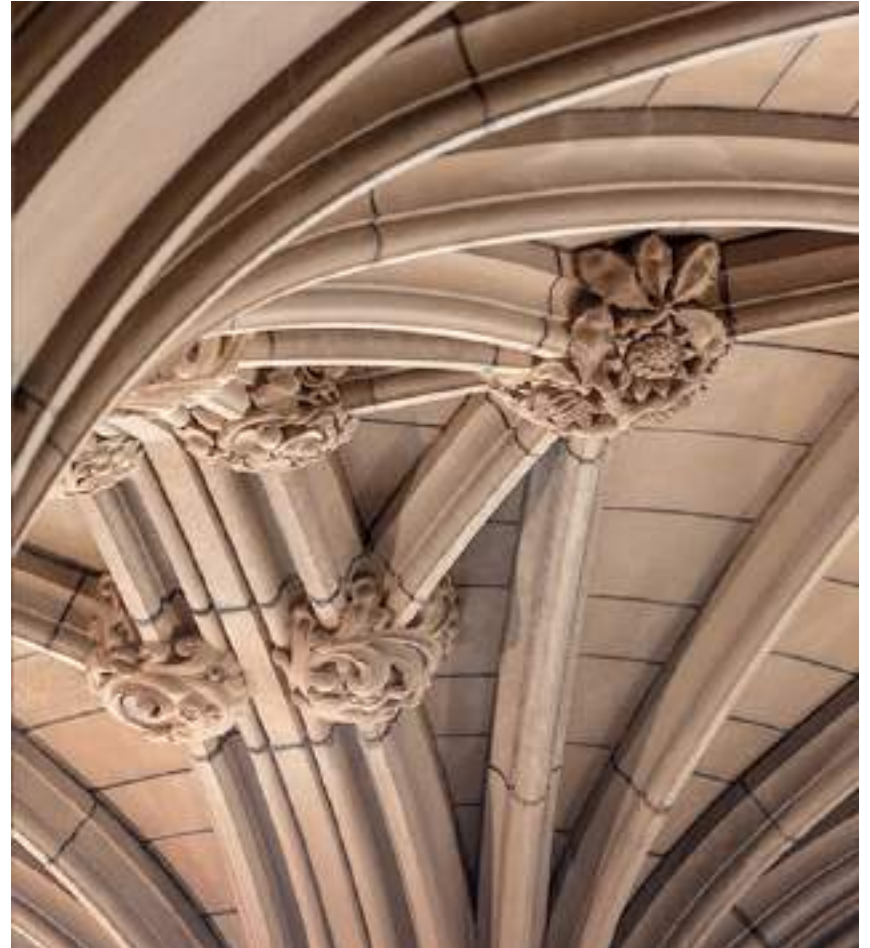
```
Servant servant = new Servant();
Scheduler scheduler = new Scheduler(servant);
Proxy proxy = new Proxy(scheduler);
```

```
OpCallback callback = new OpCallback() {
    public void onComplete(String result) {
        System.out.println(result);
    }
};
```

```
System.out.println("Request submitted, waiting for result...");
proxy.doTask("Sample Data", callback);
System.out.println("Continuing with other tasks...");
```


Thread Pool Pattern

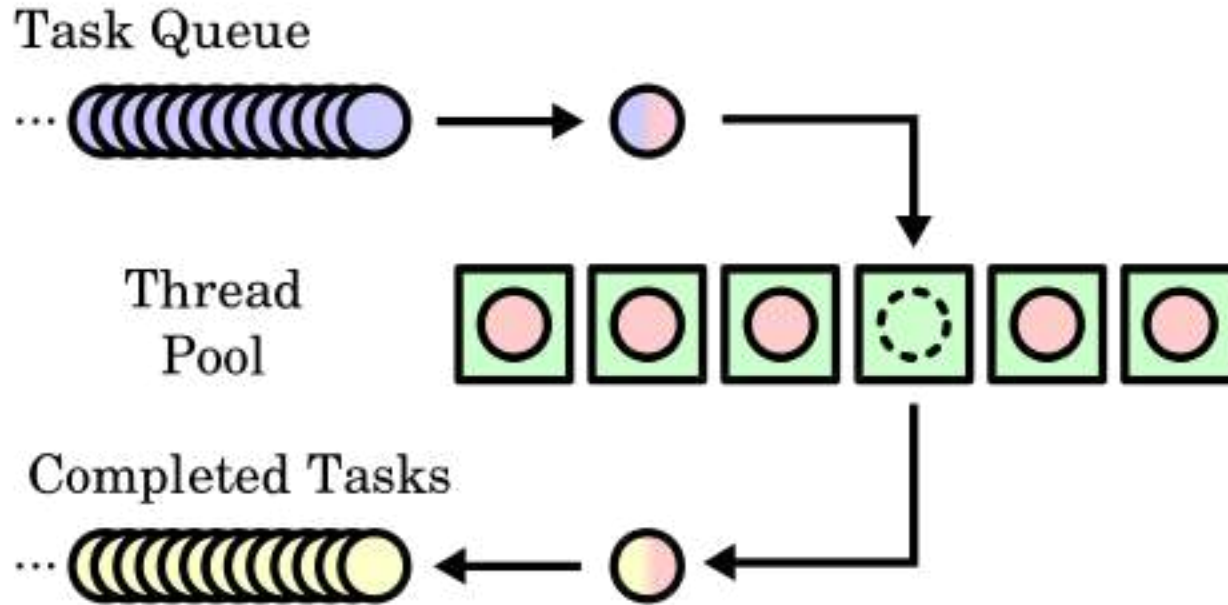
Enterprise/Web Application Patterns



Thread Pool Pattern

- Design pattern for achieving concurrent execution in an application
- Also known as replicated works
- Executing tasks sequentially leads to poor performance
 - Some tasks may take longer time than others
 - Some tasks may require communicating with other components (e.g., database)
- Several of tasks need to be executed concurrently to improve performance

Thread Pool Pattern – How it Works



By I, Cburnett, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=2233464>

Thread Pool Pattern – How it Works

- Multiple *threads* are maintained by a *thread pool*
- Threads wait for *tasks* to be allocated for concurrent execution
- Tasks are scheduled into a *task queue* or *synchronized queue*
- A thread picks a task from the task queue and once the execution of the task is completed it places it in a *completed task queue*

Thread Pool – Performance

- Thread creation and destruction
 - Expensive in terms of times and resources
 - Threads that are initially created can reduce the overhead
- Thread pool size
 - Number of pre-created threads reserved for executing tasks
 - Should be tuned carefully to achieve best performance
 - Right number reduce time and impact on resources
 - Excessive number would waste memory and processing power

Thread Pool – Configuration and Performance

- Number of threads can be adapted dynamically
 - E.g., A web server can add threads if it receives numerous web requests and remove them when number of requests drop
- Performance affected by the algorithm used to create and destroy threads
 - Creating/destroying too many unnecessary threads
 - Creating too many threads consumes resources
 - Destroying many threads require more time later when they are needed again
 - Creating small number of threads gradually may result in long waiting times
 - Destroying small number of threads gradually may waste resources

Thread Pool Pattern – Implementation

- Typically implemented on a single computer
- Can be implemented to work on server farms
 - Master process (thread pool) distributes tasks to work processes on different computers

Thread Pool: Components

- Worker Threads
- Task Queue
- Thread Pool Executor

```

public class ThreadPool {
    private final int numThreads;
    private final WorkerThread[] workerThreads;
    private final BlockingQueue<Runnable> taskQueue;

    public ThreadPool(int numThreads) {
        this.numThreads = numThreads;
        this.workerThreads = new WorkerThread[numThreads];
        this.taskQueue = new LinkedBlockingQueue<>();

        for (int i = 0; i < numThreads; i++) {
            workerThreads[i] = new WorkerThread();
            workerThreads[i].start();
        }
    }

    public void submit(Runnable task) {
        taskQueue.add(task);
    }

    public void shutdown() {
        for (int i = 0; i < numThreads; i++) {
            taskQueue.add(() -> {});
        }

        for (WorkerThread worker : workerThreads) {
            try {
                worker.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class WorkerThread extends Thread {
    @Override
    public void run() {
        while (true) {
            try {
                Runnable task = taskQueue.poll(1, TimeUnit.SECONDS);
                if (task != null) {
                    if (task == ThreadPool.this::shutdown) {
                        break;
                    }
                    task.run();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        ThreadPool threadPool = new ThreadPool(4);

        for (int i = 0; i < 10; i++) {
            int userNum = i;
            Runnable task = () -> System.out.println("Hello, User " + userNum + "!");
            threadPool.submit(task);
        }

        threadPool.shutdown();
    }
}

```

Thread Pool Advantages

1. Improved performance: Thread pools minimize the overhead of creating and destroying threads by reusing worker threads.
2. Resource management: Better control over the number of threads being used.
3. Simplified concurrency: Allows developers to focus on implementing the tasks rather than dealing with thread synchronization and lifecycle management.
4. Throttling and load balancing: Thread pools can be used to limit the number of concurrent tasks being executed, providing a way to throttle resource usage and balance the load on a system.

Thread Pool Disadvantages

1. Limited parallelism: The maximum level of parallelism is limited by the number of worker threads in the pool.
2. Complexity: Thread pools introduce complexity in terms of configuration, tuning, and management.
3. Resource contention, starvation and deadlock: Although thread pools can improve resource management, they can also lead to resource contention if too many threads are competing for the same resources. Improper use of thread pools can lead to thread starvation or deadlock if tasks submitted to the pool depend on the completion of other tasks in the pool, and there are not enough worker threads to execute them.

Web/Enterprise Applications

- Most enterprise applications deployed in distributed environment
- Accessed online by employees/staff within an enterprise and/or by end clients/customers from the Web
- Developed by web programming languages and deployed on Web servers (e.g., Nginx)
- Web servers uses process-based or thread-based execution of requests/tasks to achieve better performance
- Understanding performance of such systems require understanding I/O

Optimistic (Offline) Lock

Enterprise Application



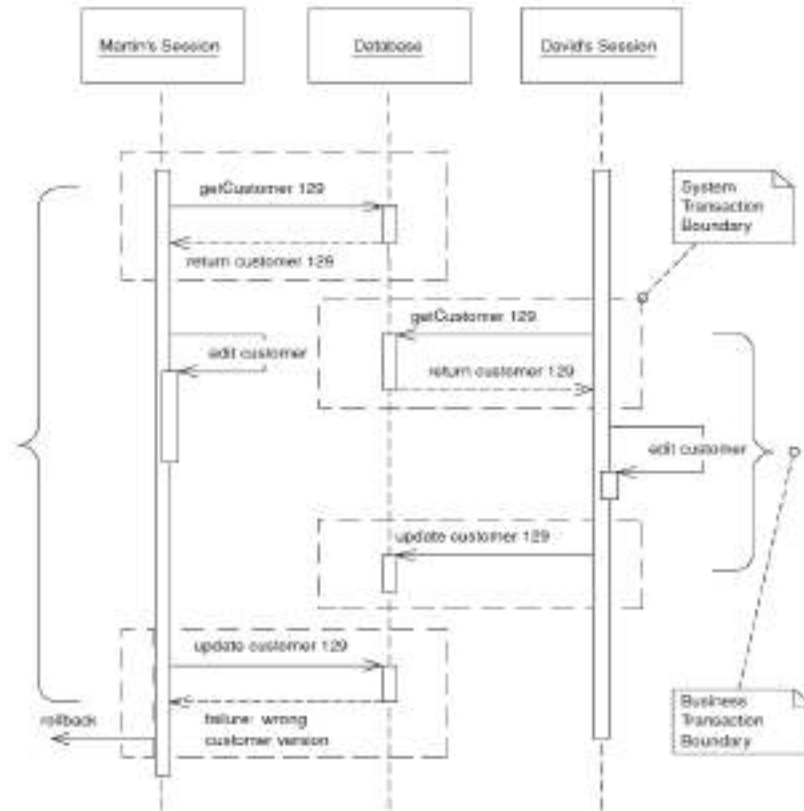
Optimistic (Offline) Lock Pattern

- *“Prevents conflicts between **concurrent** business transactions by detecting a conflict and rolling back the transaction.”*

Optimistic (Offline) Lock – When to Use it

- Optimistic concurrency management depends on the chance of conflict
 - Low (unlikely): use optimistic offline lock
 - High (likely): use pessimistic lock

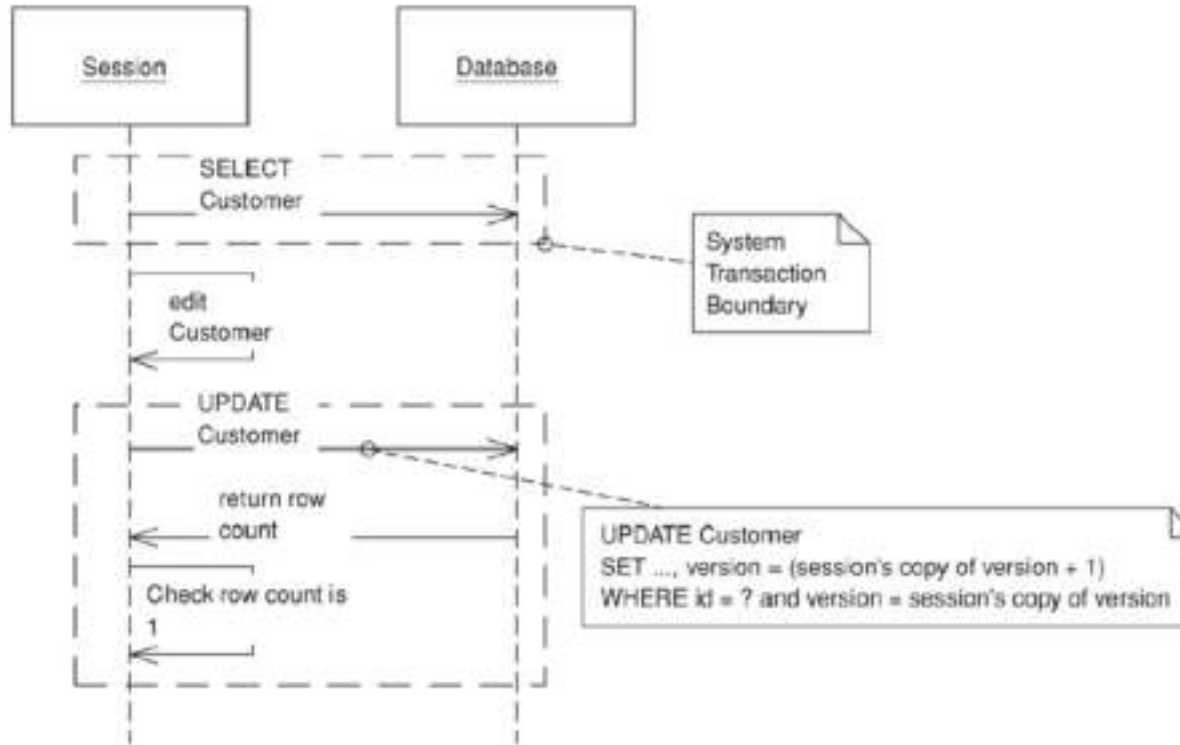
Optimistic (Offline) Lock – How



Optimistic Lock – How it Works

- Validate when a session loaded a record, another session hasn't altered it
- Associate a version number (increment) with each record
- When a record is loaded that number is kept by the session along with all other session states
- Compare the session version with the current version in the record data
 - If successful, commit the changes and update the version increment
 - A session with an old version cannot acquire the lock

Update Optimistic Check



Optimistic Lock – Issues

- The version included in Update and Delete statements
- Scenario:
 - A billing system creates charges and calculate appropriate sales tax based on customer's address
 - A session creates the charge and then locks up the customer's address to calculate the tax
 - Another session edits the customer's address (during the charge generation session)
 - Any issues?

Optimistic Lock – Inconsistent Read

- Optimistic lock fails to detect the inconsistent read
- Scenario:
 - The rate generated by the charge generation session might be invalid
 - The conflict will be gone undetected (charge session did not edit the customer address)
 - How to solve this issue?

Optimistic Lock – Inconsistent Read

- Recognize that the correctness of a session depends on the value of a certain data field
 - E.g., charge session correctness depends on the customer's address
- Add the customer address to the change set
- Alternatively, maintain a list of items to be version-checked (bit more code but clearer implementation)

Optimistic Lock – Dynamic Query

- Inconsistent read could be more complex
- E.g., a transaction depends on the results of a dynamic query
 - Save the initial results and compare them to the results of the same query at commit time

Optimistic Concurrency Control

The screenshot shows a web browser window with the address bar displaying `localhost:1274/Manage/UpdateUser`. The page has a dark navigation bar with links for Home, User Media, About, Contact, Hello, and Log off. The main content area is titled "Manage." and "Change your account settings". It contains a form with two input fields: "Email:" with the value "firstensal@ensaloom" and "Address:" with the value "1 First Street, First". Below these fields is an "Update" button. The footer of the page displays "© 2018 - Video Store Co."

localhost:1274/Manage/UpdateUser

Home User Media About Contact Hello Log off

Manage.

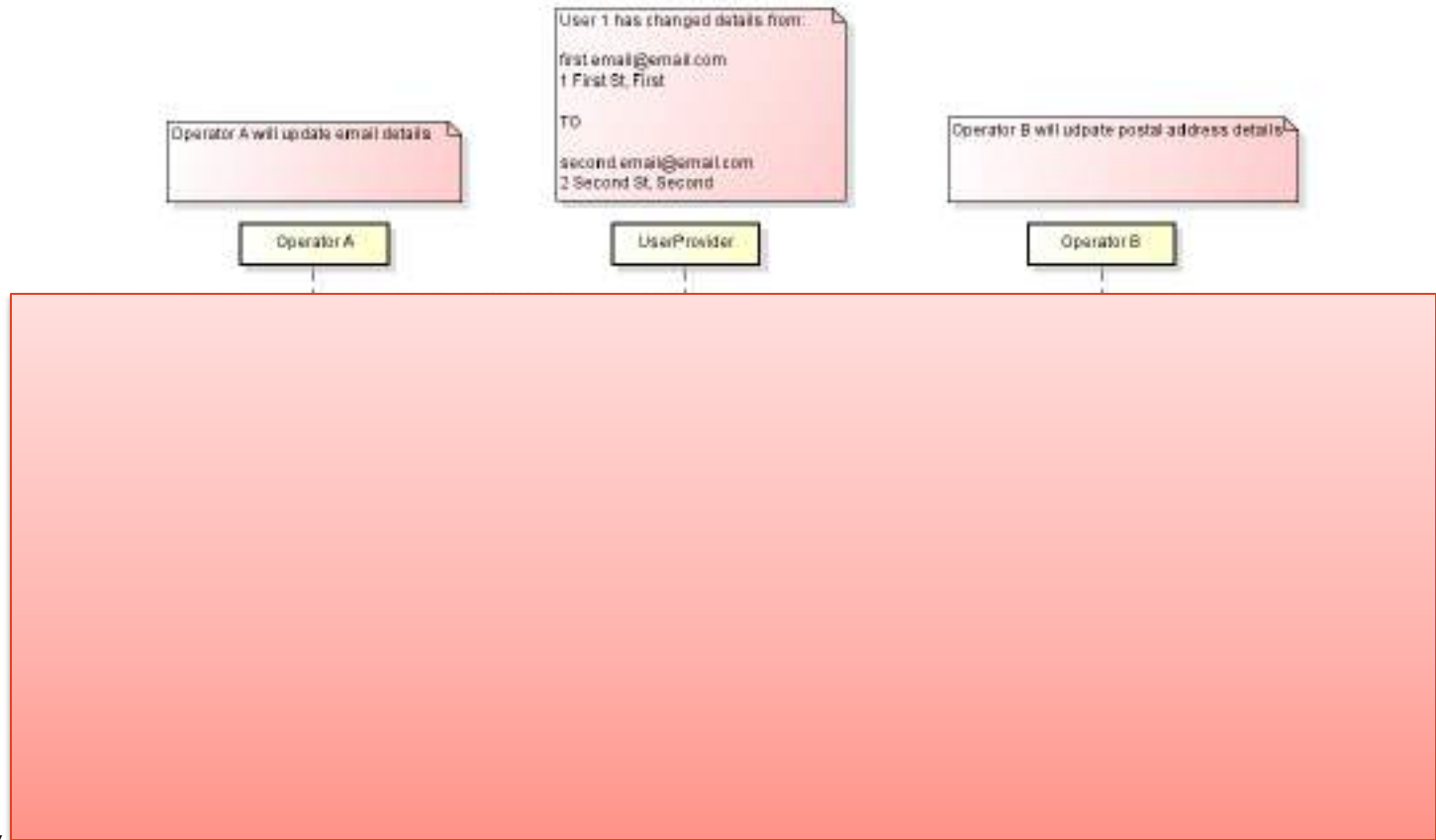
Change your account settings

Email:

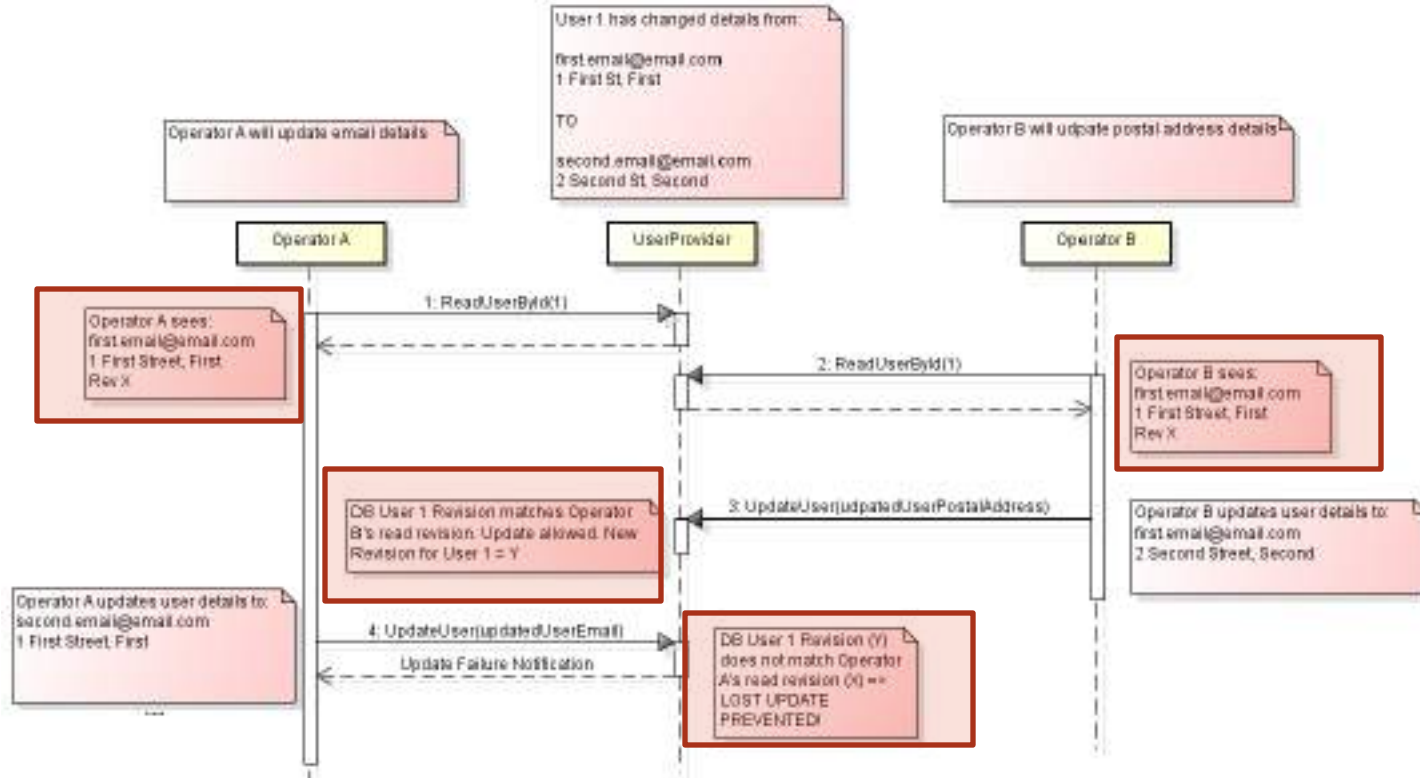
Address:

© 2018 - Video Store Co.

Optimistic Concurrency Control



Optimistic Concurrency Control



Pessimistic (Offline) Lock

Enterprise Application Pattern



Concurrency and Long Transactions

- Concurrency involves manipulating data for a business transaction that spans multiple requests
- Transaction management systems cannot help with long transactions, so multiple system transactions should be used
- Why not using optimistic lock pattern?

Concurrency and Long Transactions

- Concurrency involves manipulating data for a business transaction that spans multiple requests
- Transaction management systems cannot help with long transactions, so multiple system transactions should be used
- Why not using optimistic lock pattern?
 - Several users access the same data within a business transaction, only one will commit but the others will fail
 - Other transaction processing time will be lost (conflict is only detected at the end)

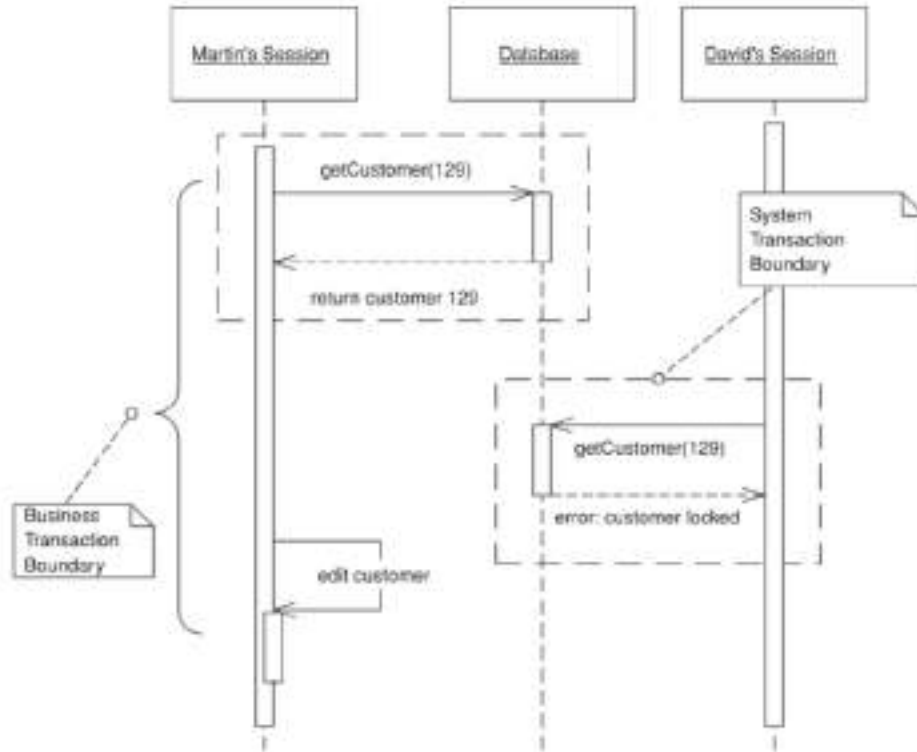
Pessimistic Lock Pattern

*“Prevents conflicts between **concurrent business transactions** by allowing only one business transaction at a time to access data.”*

Pessimistic Lock – When to Use it

- When the chance of conflict between two concurrent business transactions is high
- Cost of conflict is too high regardless of its likelihood
- Use when it is really required (it has high performance costs)
- Business transactions spans across multiple system transactions

Pessimistic Lock – How it Works



Pessimistic Lock – How

1. Determine type of locks needed
2. Build a lock manager
3. Define Procedures for a business transaction to use locks

Note: determine which record types to lock if using pessimistic lock to complement optimistic lock

Pessimistic Lock – Types of Lock

- **Exclusive write Lock:** a transaction acquire a lock to edit a session
 - Does not allow 2 transactions to write to the same record concurrently
 - Does not address reading data
 - Allow much more concurrency
- **Exclusive read Lock:** a transaction acquire a lock to load a record
 - Obtain most recent data regardless of the intention to edit data
 - Restrict concurrency
- **Read/Write Lock:** combines both read and write locks
 - Restrict concurrency by Read lock and increased concurrency by Write lock

Pessimistic Lock – Types of Lock

- **Read/Write Lock:** combines the benefit of both lock types:
 - *Read and write locks are mutually exclusive:* a record can't be write-lock if any other transaction owns a read lock on it and vice versa
 - *Concurrent read locks are acceptable:* several sessions can be readers once one has been allowed as read-lock
 - Increase system concurrency
 - Hard to implement

Pessimistic Lock – Which Lock to Choose

- Factors to consider:
 - Maximize system concurrency
 - Satisfy business needs
 - Minimize code complexity
 - Should be understood by data molders and analysts
- Avoid choosing the wrong locking type, locking everything, or locking the wrong records

Pessimistic Lock – Lock Manager

- Define your lock manager to coordinate acquiring/releasing lock requests
 - Need to maintain which transaction (session) and what is being locked
 - Use only one lock table (hash or database table)
 - Use Singleton (GOF) for in-memory lock table
 - Database-based lock manager for clustered application server environment
 - Transaction should interact only with the lock manager but never with a lock object

Pessimistic Lock – Lock Manager

- In-memory lock table
 - Serialize access to the entire lock manager
- Database lock table
 - Interact via transaction-based system
 - Use database serialization
 - Read and Write lock serialization can be achieved by setting a unique constraint on the column storing lockable item's ID
 - Read/write locks are more difficult
 - Reads the lock table and insert into it so you need to avoid inconsistent read
- Use a separate serializable system transaction for lock acquisition to improve performance

Pessimistic Lock – Lock Manager

- **Lock protocol** defines how a transaction can use the lock manager including:
 - What to lock?
 - When to lock and release?
 - How the lock should behave when a lock cannot be acquired?

Lock Manager – When to Lock

- General rule: a transaction should acquire a lock before loading the data
- Order of lock and load is not a matter in some cases:
 - E.g., Frequent read transaction
 - Perform an optimistic check after you acquire the Pessimistic lock on an item to ensure having the latest version of it

Lock Manager – What to Lock

- Lock an ID, or a primary key, of the object/record
- Obtain the lock before you load the object/record (to ensure the object is current)
- Lock release
 - When transaction is complete
 - May release before transaction completion in some cases (understand what is the lock)

Lock Manager – Lock behaviour

- Abort when a transaction cannot acquire a lock
 - Should be acceptable by end users (early warning)

References

- Martin Fowler (With contributions from David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford). 2003. *Patterns of Enterprise Applications Architecture*. Pearson.
- Enterprise-Scale Software Architecture (COMP5348) slides
- Web Application Development (COMP5347) slides
- Basarat Ali Syed 2014, Beginning Node.js. E-book, accessible online from USYD library
- Wikipedia, Concurrency Pattern https://en.wikipedia.org/wiki/Concurrency_pattern
- Wikipedia, Thread Pool, https://en.wikipedia.org/wiki/Thread_pool

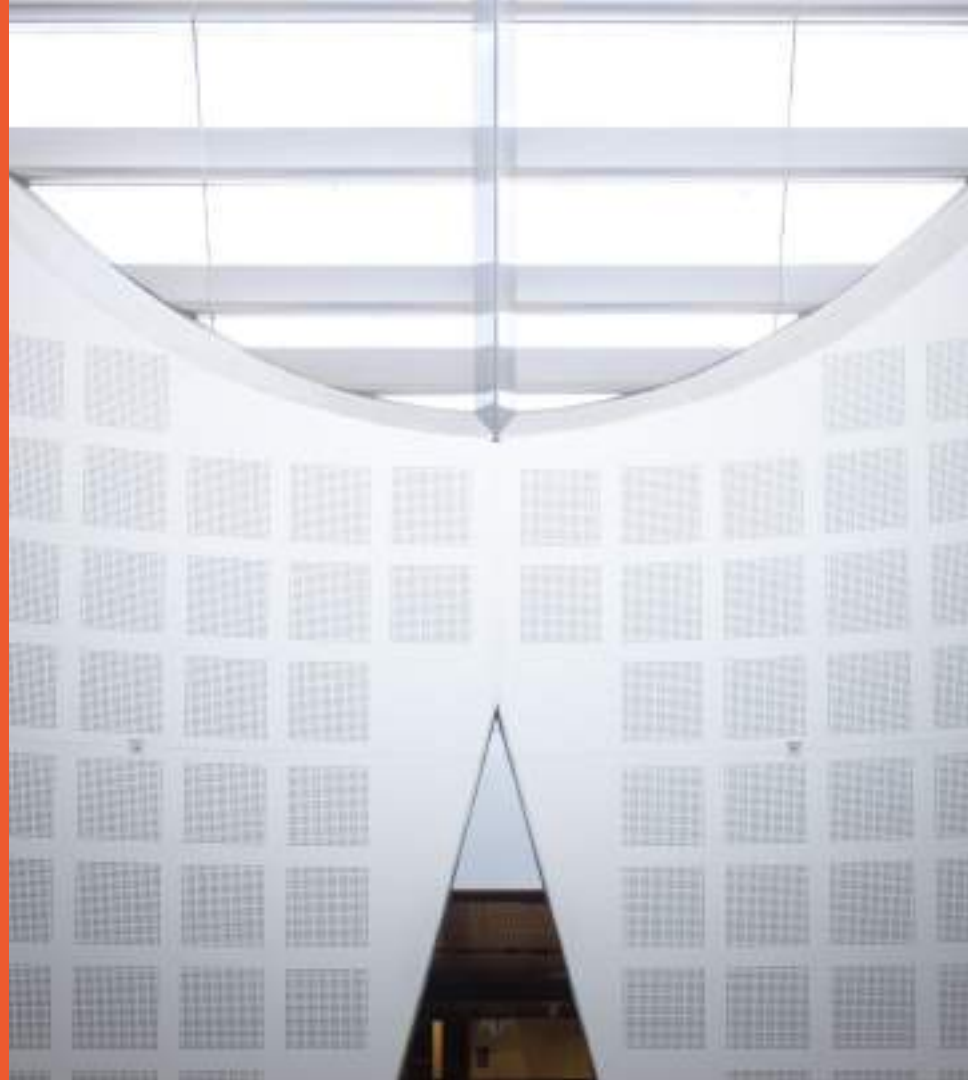
Software Construction and Design 2

SOFT3202 / COMP9202

Abstract Factory, State,
Command, Mediator

Dr. Rahul Gopinath

School of Computer Science

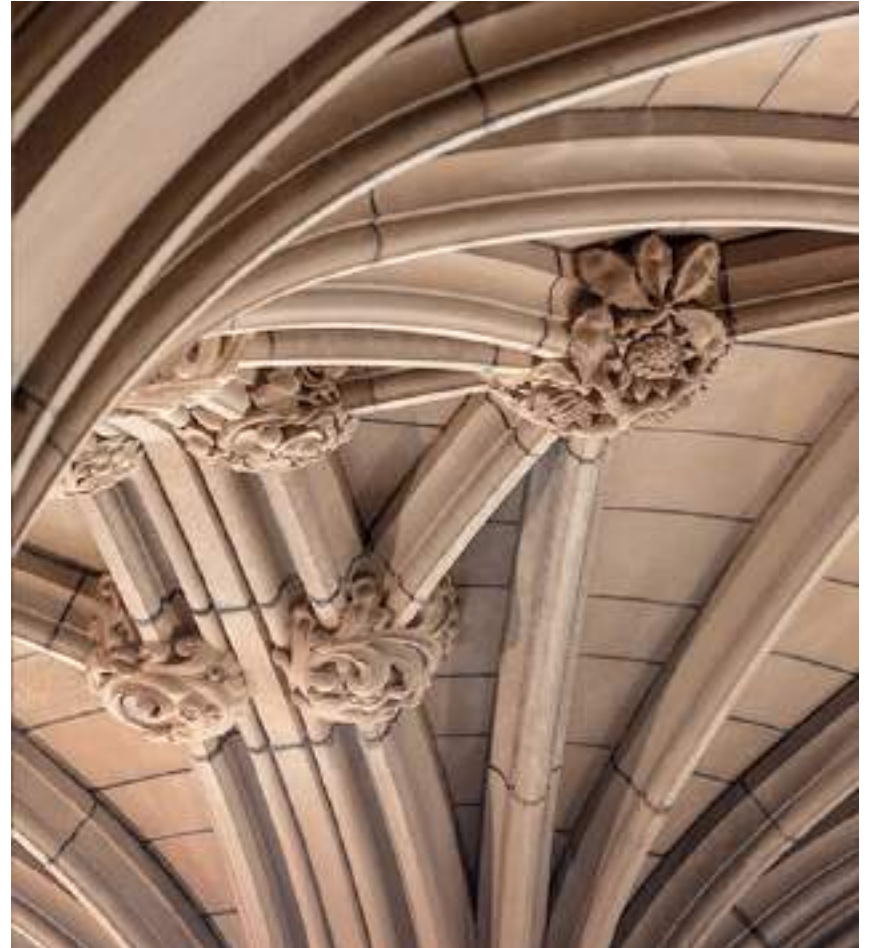


Agenda

- GoF Design Patterns
 - Abstract Factory
 - State
 - Command
 - Mediator
 - Observer
 - Strategy

Abstract Factory

Object Creational

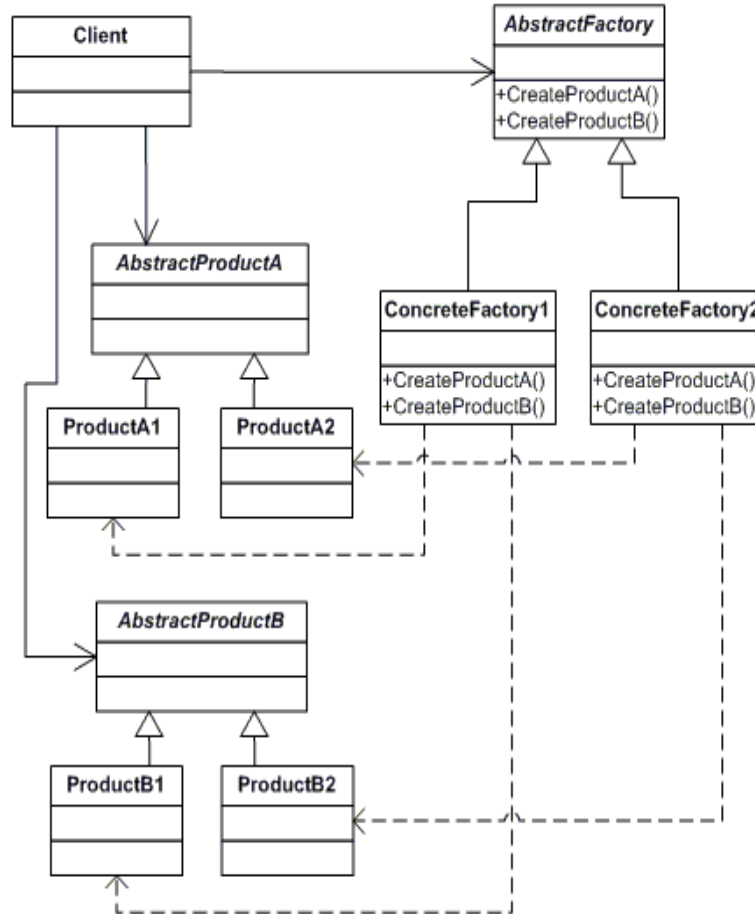


Abstract Factory Pattern

- Intent
 - Provide an interface for creating **families of related or dependent objects** without specifying their concrete classes
- Also known as
 - Kit
- Applicability
 - A system should be independent of how its products are created, composed and represented
 - A system should be configured with one of multiple families of products
 - Family of related product objects is designed to be used together and you need to enforce this constraint
 - You want to provide a class library of products, and you want to reveal just their interfaces, not their implementation

Abstract Factory

– Structure

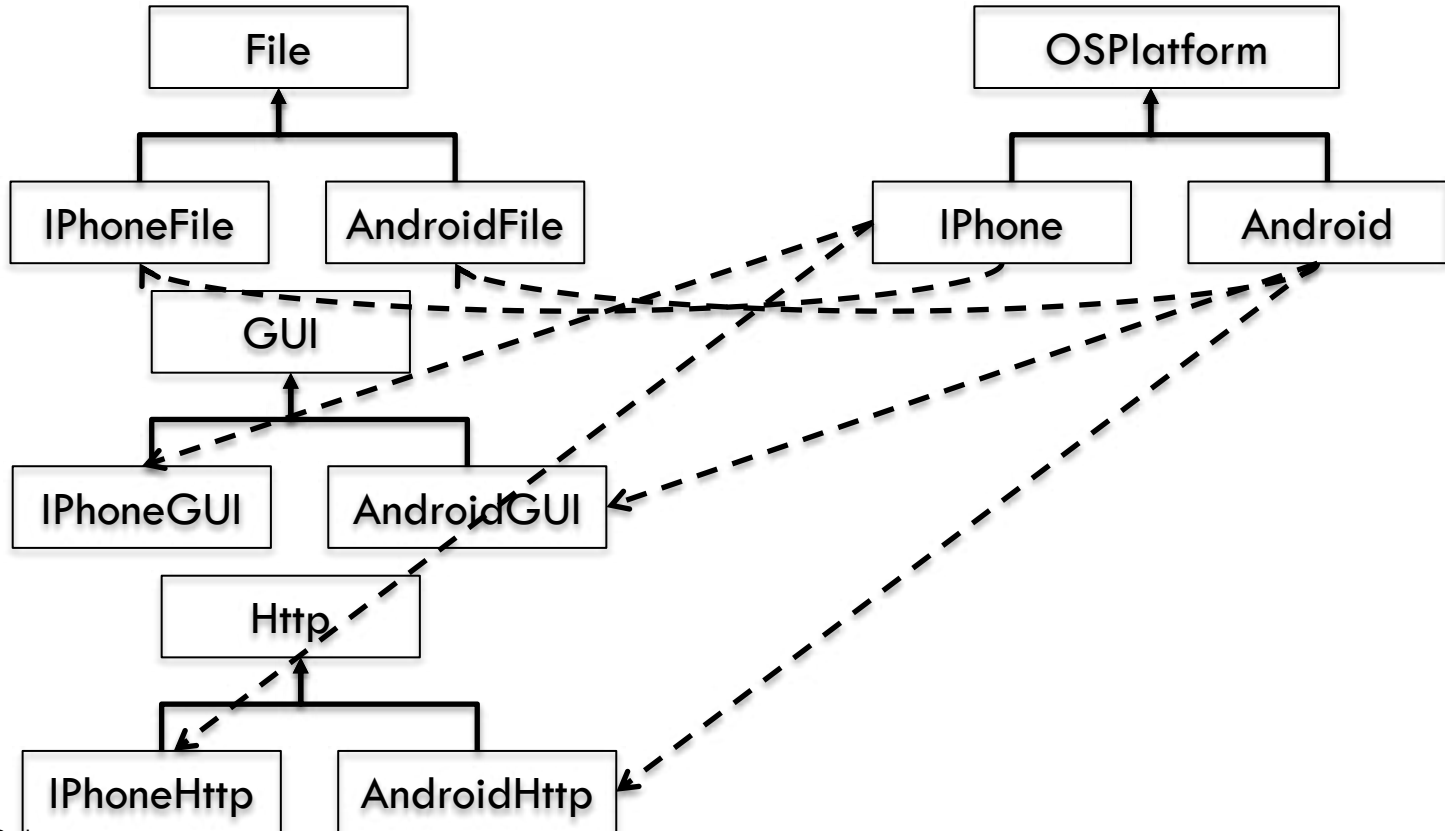


Abstract Factory Pattern – Participants

- **AbstractFactory**
 - Declares an interface for operations that create abstract product objects
- **ConcreteFactory**
 - Implements the operations to create concrete product objects
- **AbstractProduct**
 - declares an interface for a type of product object
- **ConcreteProduct**
 - defines a product object to be created by the corresponding concrete factory
 - Implements the AbstractProduct interface
- **Client**
 - uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Example

- Let's assume a platform independent app
- The app requires as products
 - Files
 - GUI
 - Http Access
- Factories depending on the OS
 - Create objects via abstract factory
 - Concrete factory represents a platform



```
from abc import ABC, abstractmethod

# Abstract Factory
class AbstractOSFactory(ABC):
    @abstractmethod
    def create_file(self):
        pass

    @abstractmethod
    def create_gui(self):
        pass

    @abstractmethod
    def create_http(self):
        pass

# Concrete Factories
class iPhoneOSFactory(AbstractOSFactory):
    def create_file(self):
        return iPhoneFile()

    def create_gui(self):
        return iPhoneGui()

    def create_http(self):
        return iPhoneHttp()

class AndroidOSFactory(AbstractOSFactory):
    def create_file(self):
        return AndroidFile()

    def create_gui(self):
        return AndroidGui()

    def create_http(self):
        return AndroidHttp()
```

```
# Abstract Products
class AbstractFile(ABC):
    @abstractmethod
    def read(self):
        pass

class AbstractGui(ABC):
    @abstractmethod
    def render(self):
        pass

class AbstractHttp(ABC):
    @abstractmethod
    def request(self):
        pass

# Concrete Products
class iPhoneFile(AbstractFile):
    def read(self):
        return "Reading file on iPhone"

class iPhoneGui(AbstractGui):
    def render(self):
        return "Rendering GUI on iPhone"

class iPhoneHttp(AbstractHttp):
    def request(self):
        return "Making HTTP request on iPhone"

class AndroidFile(AbstractFile):
    def read(self):
        return "Reading file on Android"

class AndroidGui(AbstractGui):
    def render(self):
        return "Rendering GUI on Android"

class AndroidHttp(AbstractHttp):
    def request(self):
        return "Making HTTP request on Android"
```

```
# Client
def client_code(factory: AbstractOSFactory):
    file = factory.create_file()
    gui = factory.create_gui()
    http = factory.create_http()

    print(file.read())
    print(gui.render())
    print(http.request())

if __name__ == "__main__":
    print("iPhone OS:")
    client_code(iPhoneOSFactory())

    print("\nAndroid OS:")
    client_code(AndroidOSFactory())
```

State Design Pattern

Object Behavioural

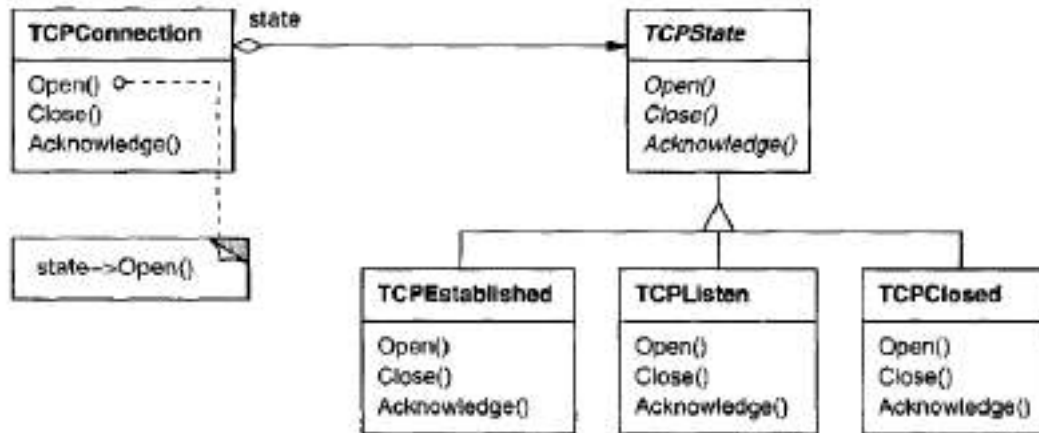


State Design Pattern

- Object behavioral
 - Object relationships which can be changed dynamically at run-time
 - How a group of objects can collaborate to perform a task that no single object can do alone
- Intent
 - Allow an object to change its behavior when its internal state changes
- Think finite-state-machine

Motivating Scenario

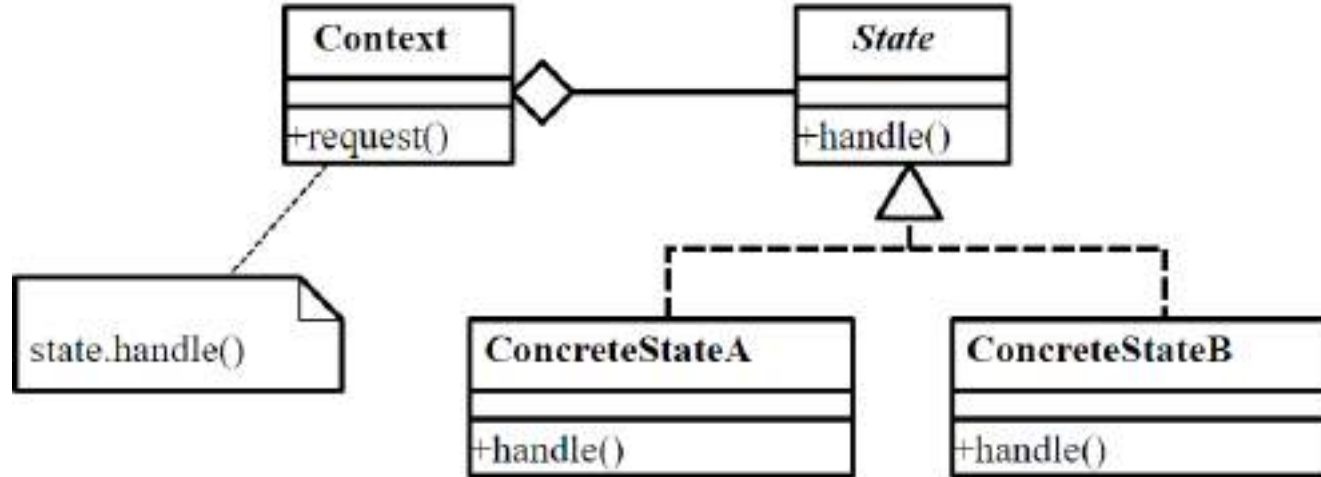
- TCP network connection
- States: Established, Listening and Closed
- TCP connection responds based on its current state



State Pattern – When to Use

- The state of an object drives its behavior and changes its behavior dynamically at run-time
- Operations have large, multipart conditional statements that depend on the object's state
 - Repeated conditional structure

State Pattern – Structure



By State_Design_Pattern_UML_Class_Diagram.png: JoaoTrindade (talk)Original uploader was JoaoTrindade at en.wikipediaderivative work: Ertwroc (talk) - State_Design_Pattern_UML_Class_Diagram.png, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=10234908>

State Pattern – Participants

- Context (TCPConnection)
 - Defines the interface of interest to clients
 - Maintains an instance of a ConcreteState subclass that defines the current state
- State (TCPState)
 - Defines an interface for encapsulating the behavior associated with a certain state of the Context
- ConcreteState subclasses (TCPEstablished, TCPListen, TCPClosed)
 - Each subclass implements a behavior associated with a state of the Context

State Pattern – Collaborations

- Context delegates state-specific requests to the current ConcreteState object
- A context may pass itself as an argument to the State object handling the request, so the State object access the context if necessary
- Context is the primary interface for clients
 - Clients can configure a context with State objects, so its clients don't have to deal with the State objects directly
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances

State Pattern – Consequences

- Localizes state-specific behavior for different states
 - Using data values and context operations make code maintenance difficult
 - State distribution across different sub-classes useful when there are many states
 - Better code structure for state-specific code (better than monolithic)
- It makes state transition explicit
 - State transitions as variable assignments
 - State objects can protect the context from inconsistent state
- State objects can be shared
 - When the state they represent is encoded entirely in their type

State Pattern – Implementation (1)

- Defining the state transitions
 - Let the state subclasses specify their successor state to make the transition (decentralized)
 - Achieves flexibility – easy to modify and extend the logic
 - Introduces implementation dependencies between subclasses
- Table-based state transitions
 - Look-up table that maps every possible input to a succeeding state
 - Easy to modify (transition data not the program code) but:
 - Less efficient than a functional call
 - Harder to understand the logic (transition criteria is less explicit)
 - Difficult to add actions to accompany the state transitions

State Pattern – Implementation (2)

- When to create and destroy state objects?
 - Pre-create them and never destroy them
 - Useful for frequent state changes (save costs of re-creating states)
 - Context must keep reference to all states
 - Only when they are needed and destroyed them thereafter
 - States are not known at run-time and context change states frequently

State Pattern – Implementation (3)

- Using dynamic inheritance
 - Changing the object's class at run-time is not possible in most OO languages
 - Delegation-based languages allow changing the delegation target at run-time

```
# State Interface
class TCPState(ABC):
```

```
    @abstractmethod
    def open(self):
        pass
```

```
    @abstractmethod
    def close(self):
        pass
```

```
# Concrete States
```

```
class TCPOpen(TCPState):
```

```
    def open(self):
        print("Connection is already open.")
        return self
```

```
    def close(self):
        print("Closing the connection.")
        return TCPClosed()
```

```
class TCPClosed(TCPState):
```

```
    def open(self):
        print("Opening the connection.")
        return TCPOpen()
```

```
    def close(self):
        print("Connection is already closed.")
        return self
```

```
# Context
```

```
class TCPConnection:
```

```
    def __init__(self):
        self.state = TCPClosed()
```

```
    def open(self):
        self.state = self.state.open()
```

```
    def close(self):
        self.state = self.state.close()
```

```
# Client code
```

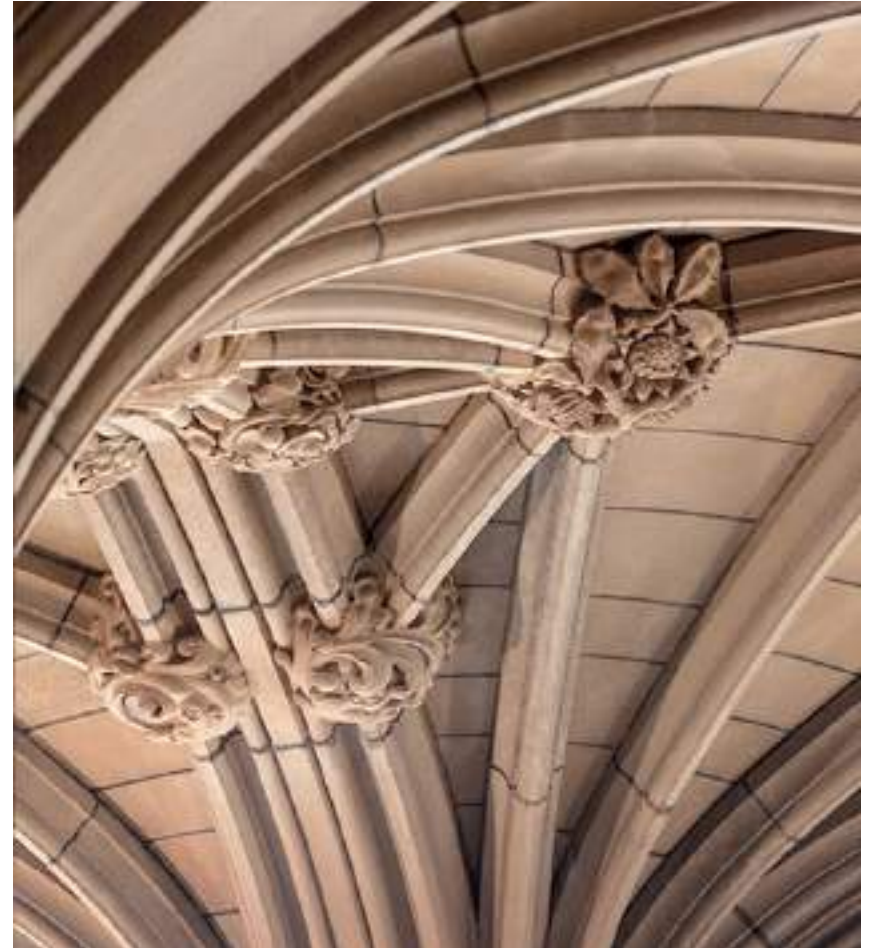
```
connection = TCPConnection()
connection.open()
connection.open()
connection.close()
connection.close()
```

State Pattern – Related Patterns

- Flyweight
 - Explains when and how state objects can be shared
- Singleton
 - State object is often implemented as Singleton

Command Design Pattern

Object Behavioural



Command Pattern

– Intent

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

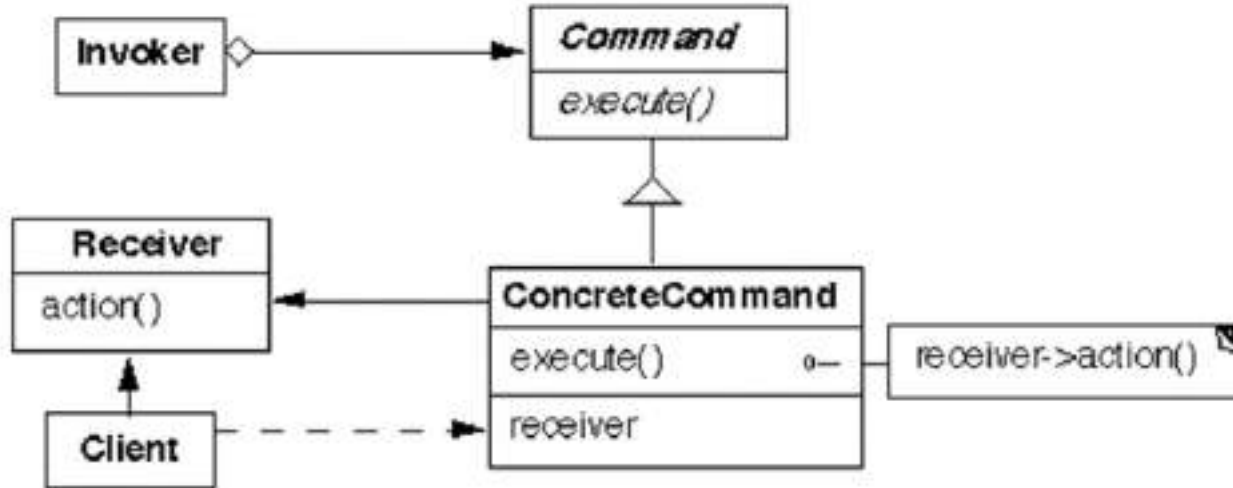
– Applicability

- To parameterize objects by an action to perform – like callback functions
- To specify, queue and execute requests at different times
 - A command object can have a lifetime independent of the original request
- To support undo
 - The Command's Execute operation can store state for reversing its effects in the command itself

Command Pattern – Applicability

- To support logging changes so it can be applied in case of a system crash
 - Load and Store operations in the Command interface to keep a persistent log of change
- To structure a system around high-level operations built on primitive operations
 - E.g., transaction systems maintain set of changes to data
 - Commands have a common interface that invoke all transactions the same way
 - Also, can extend the system with new transactions

Command Pattern – Structure



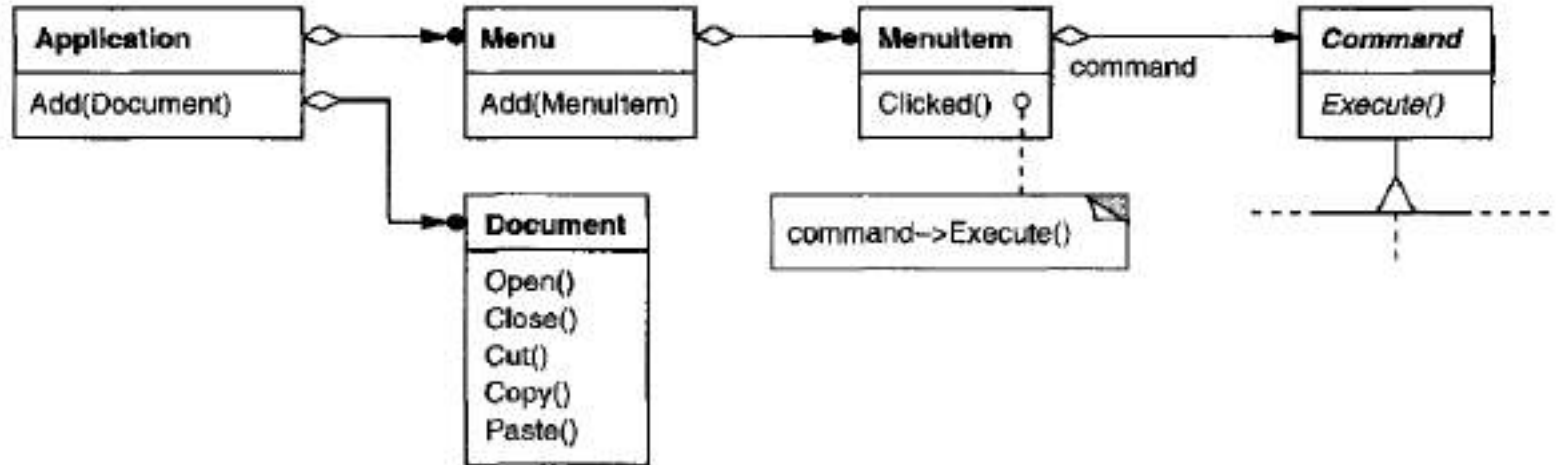
Command – Structure (Participants)

- **Command**
 - Declares an interface for executing an operation
- **ConcreteCommand** (PasteCommand, OpenCommand)
 - Defines a binding between a Receiver object and an action
 - Implements Execute by invoking the corresponding operation(s) on Receiver
- **Client (Application)**
 - Creates a ConcreteCommand object and sets its receiver
- **Invoker (MenuItem)**
 - Asks the command to carry out the request
- **Receiver**
 - Knows how to perform the operations associated with carrying out a request

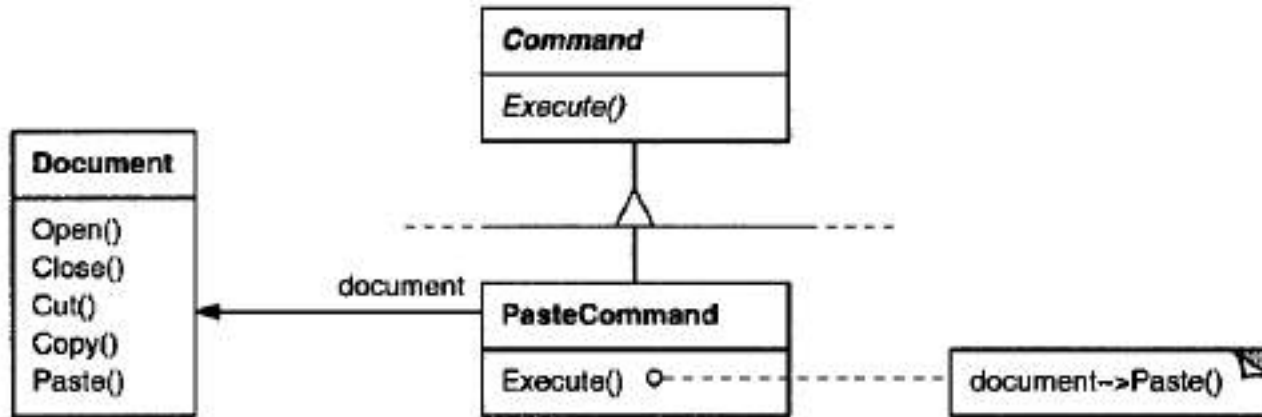
Command – Toolkits User Interface Example

- Consider a user interface toolkits that include objects like buttons and menus that carry out a request in response to user input
- The toolkit cannot implement the request in the button or menu objects; applications that use the toolkit know what should be done on which object
- Requests will be issued to objects without knowing anything about the operation being requested or the receiver of the request

Command – Example

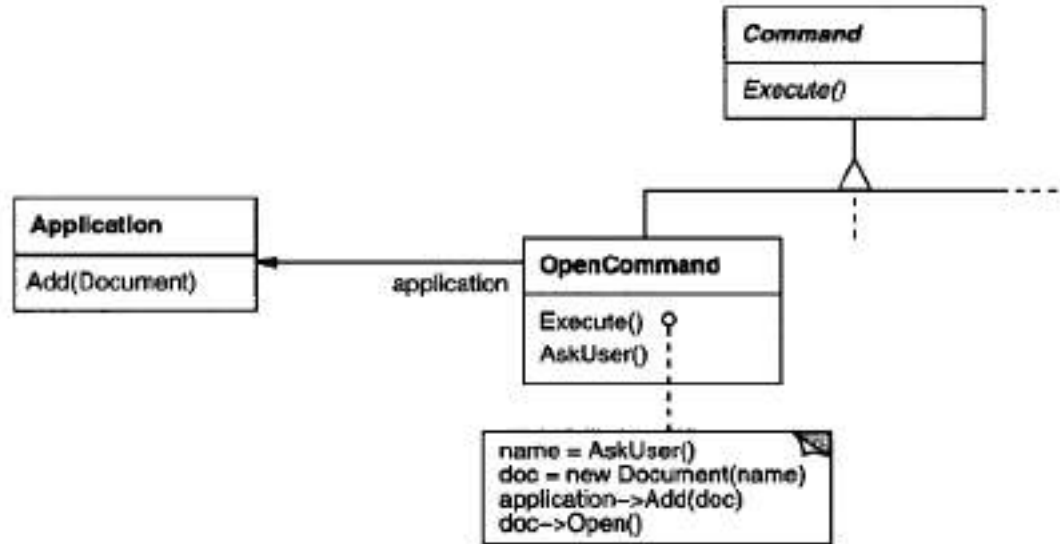


Command – Toolkit (Paste Command)



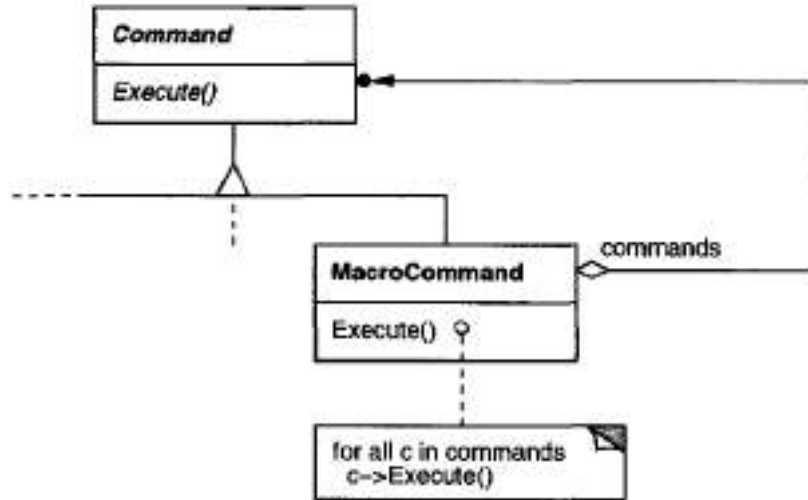
PasteCommand allows pasting text from the clipboard into a document

Command – Toolkit (Open Command)



OpenCommand's Execute operation

Command – Toolkit (Sequence of Commands)



- MenuItem needs to execute a sequence of commands
 - E.g., MenuItem for centering a page at normal size constructed from `CenterDocCommand` and `NormalSizeCommand`
- **MacroCommand** class allow menuItem to execute sequence of commands

```
import tkinter as tk
from abc import ABC, abstractmethod
```

```
# Command Interface
class Command(ABC):
    @abstractmethod
    def execute(self):
        pass
```

```
# Concrete Commands
class HelloCommand(Command):
    def __init__(self, app):
        self.app = app

    def execute(self):
        self.app.display_message("Hello!")

class GoodbyeCommand(Command):
    def __init__(self, app):
        self.app = app

    def execute(self):
        self.app.display_message("Goodbye!")
```

```
# Invoker (GUI Buttons)
class GUIButton(tk.Button):
    def __init__(self, master, command, *args, **kwargs):
        super().__init__(master, command=command.execute, *args, **kwargs)
```

```
# Receiver (Application)
class MyApp:
    def __init__(self, root):
        self.label = tk.Label(root, text="")
        self.label.pack(pady=10)

        hello_command = HelloCommand(self)
        hello_button = GUIButton(root, hello_command, text="Say Hello")
        hello_button.pack(pady=5)

        goodbye_command = GoodbyeCommand(self)
        goodbye_button = GUIButton(root, goodbye_command, text="Say Goodbye")
        goodbye_button.pack(pady=5)

    def display_message(self, message):
        self.label.config(text=message)
```

```
# Client
if __name__ == "__main__":
    root = tk.Tk()
    root.title("Command Pattern with GUI")
    app = MyApp(root)
    root.mainloop()
```

Mediator Design Pattern

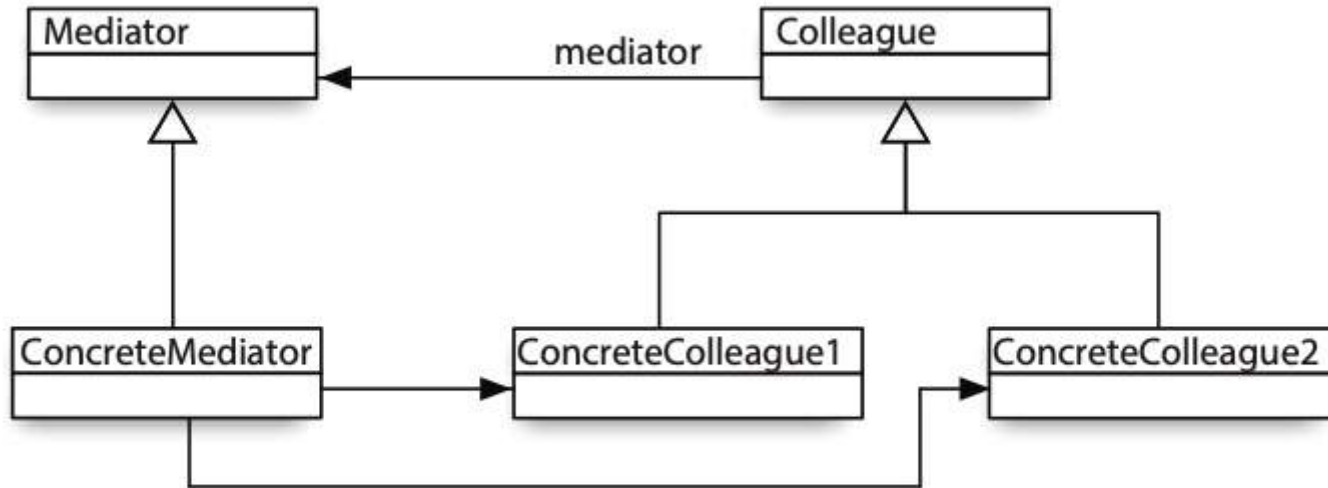
Object Behavioural



Mediator

- **Purpose/Intent**
 - *Create an object that manages communications for a set of objects, promoting loose coupling and maintaining encapsulation*
- **Motivation**
 - Allowing objects which need to communicate with one another to be decoupled by introducing an intermediary class which handles all the communication between objects.
- **Applicability**
 - Any time you have a set of objects that can communicate in complex ways, or when reusing an object is difficult because it has many connections, or group behaviour (over a set of classes) needs to be easily customisable
- **Benefits**
 - Limits need to subclassing communicating objects (Colleagues)
 - simplifies communication protocols via clear abstraction of how the objects are communicating
- **Limitations**
 - The Mediator itself can become very complex
- **See also: Façade.**

Structure



Participants

- Mediator
 - defines an interface for communicating with Colleague objects
- ConcreteMediator
 - implements cooperative behaviour by coordinating Colleague objects
 - maintain a list of its Colleagues
- Colleague
 - defines an interface for Colleagues
- ConcreteColleague
 - implements the Colleague interface
 - knows its Mediator object.
 - communicates with another Colleague through its Mediator object.

```

from abc import ABC, abstractmethod

# Mediator Interface
class Mediator(ABC):

    @abstractmethod
    def notify(self, sender, event):
        pass

# Concrete Mediator
class ConcreteMediator(Mediator):

    def __init__(self, component1, component2):
        self.component1 = component1
        self.component1.mediator = self
        self.component2 = component2
        self.component2.mediator = self

    def notify(self, sender, event):
        if sender == self.component1 and event == "A":
            print("Mediator reacts on event A and triggers component2:")
            self.component2.do_something()
        elif sender == self.component2 and event == "B":
            print("Mediator reacts on event B and triggers component1:")
            self.component1.do_another_thing()

```

```

class BaseComponent(ABC):
    def __init__(self, mediator: Mediator = None):
        self.mediator = mediator

    def notify_mediator(self, event):
        if self.mediator is not None:
            self.mediator.notify(self, event)

class Component1(BaseComponent):
    def do_something(self):
        print("Component1 does something.")
        self.notify_mediator("A")

    def do_another_thing(self):
        print("Component1 does another thing.")

class Component2(BaseComponent):
    def do_something(self):
        print("Component2 does something.")
        self.notify_mediator("B")

    def do_another_thing(self):
        print("Component2 does another thing.")

# Client code
component1 = Component1()
component2 = Component2()
mediator = ConcreteMediator(component1, component2)

component1.do_something() # Output:
# Component1 does something.
# Mediator reacts on event A and triggers component2:
# Component2 does something.

component2.do_something() # Output:
# Component2 does something.
# Mediator reacts on event B and triggers component1:
# Component1 does another thing.

```

Observer

- Defines a one-to-many dependency between objects, so that when one object (the subject or publisher) changes its state, all its dependents (observers or subscribers) are notified and updated automatically.
- This pattern is useful when you need to maintain consistency between related objects without making them tightly coupled.

```
# Observer Interface
class Observer(ABC):

    @abstractmethod
    def update(self, subject):
        pass

# Concrete Observers
class ConcreteObserverA(Observer):

    def update(self, subject):
        if subject.state < 3:
            print("ConcreteObserverA: Reacted to the event")

class ConcreteObserverB(Observer):

    def update(self, subject):
        if subject.state == 0 or subject.state >= 2:
            print("ConcreteObserverB: Reacted to the event")
```

```
# Subject
class Subject:

    def __init__(self):
        self.observers = []
        self._state = None

    @property
    def state(self):
        return self._state

    @state.setter
    def state(self, value):
        self._state = value
        self._notify()

    def attach(self, observer):
        self.observers.append(observer)

    def detach(self, observer):
        self.observers.remove(observer)

    def _notify(self):
        for observer in self.observers:
            observer.update(self)

# Client code
subject = Subject()

observer_a = ConcreteObserverA()
subject.attach(observer_a)

observer_b = ConcreteObserverB()
subject.attach(observer_b)

subject.state = 0 # Output:
# ConcreteObserverA: Reacted to the event
# ConcreteObserverB: Reacted to the event

subject.state = 2 # Output:
# ConcreteObserverA: Reacted to the event
# ConcreteObserverB: Reacted to the event

subject.state = 4 # Output:
# ConcreteObserverB: Reacted to the event
```

Strategy

- Allows you to define a family of algorithms, encapsulate each one, and make them interchangeable.
- Enables an algorithm's behavior to be selected at runtime.
- Involves a strategy interface with a method that the concrete strategy classes implement, and a context class that uses the strategy interface to delegate a specific behavior to the currently selected strategy.

```

from abc import ABC, abstractmethod

# Strategy Interface
class SortingStrategy(ABC):

    @abstractmethod
    def sort(self, data):
        pass

# Concrete Strategies
class BubbleSortStrategy(SortingStrategy):

    def sort(self, data):
        print("Using Bubble Sort")
        n = len(data)
        for i in range(n):
            for j in range(0, n - i - 1):
                if data[j] > data[j + 1]:
                    data[j], data[j + 1] = data[j + 1], data[j]
        return data

class QuickSortStrategy(SortingStrategy):

    def sort(self, data):
        print("Using Quick Sort")
        return self._quick_sort(data)

    def _quick_sort(self, data):
        if len(data) <= 1:
            return data
        pivot = data[len(data) // 2]
        left = [x for x in data if x < pivot]
        middle = [x for x in data if x == pivot]
        right = [x for x in data if x > pivot]
        return self._quick_sort(left) + middle + self._quick_sort(right)

```

```

# Context
class SortingContext:

    def __init__(self, strategy: SortingStrategy):
        self.strategy = strategy

    def set_strategy(self, strategy: SortingStrategy):
        self.strategy = strategy

    def execute_sort(self, data):
        return self.strategy.sort(data)

# Client code
data = [5, 2, 3, 1, 4]

sorting_context = SortingContext(BubbleSortStrategy())
sorted_data = sorting_context.execute_sort(data.copy())
print(sorted_data) # Output:
# Using Bubble Sort
# [1, 2, 3, 4, 5]

sorting_context.set_strategy(QuickSortStrategy())
sorted_data = sorting_context.execute_sort(data.copy())
print(sorted_data) # Output:
# Using Quick Sort
# [1, 2, 3, 4, 5]

```

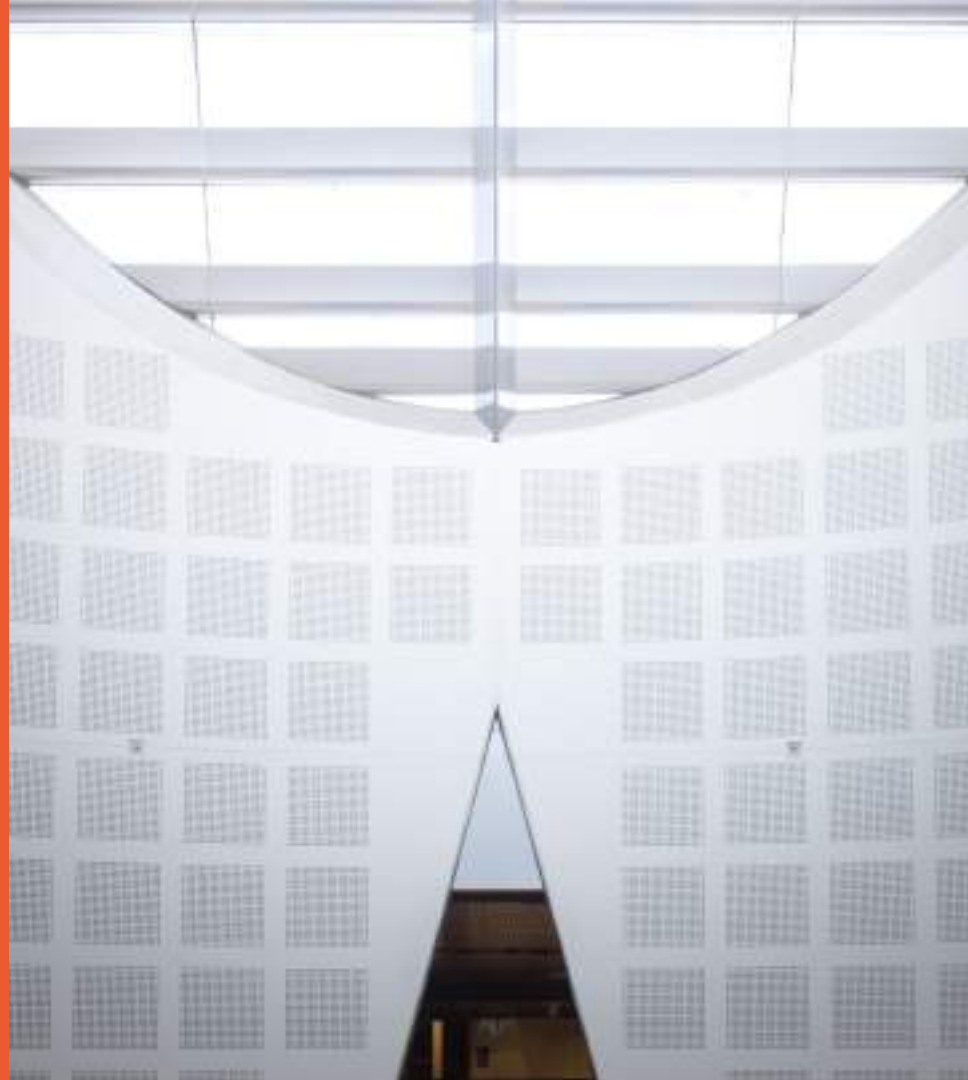
Software Construction and Design 2

SOFT3202 / COMP9202

Patterns For Interactive
Applications

Dr Rahul Gopinath

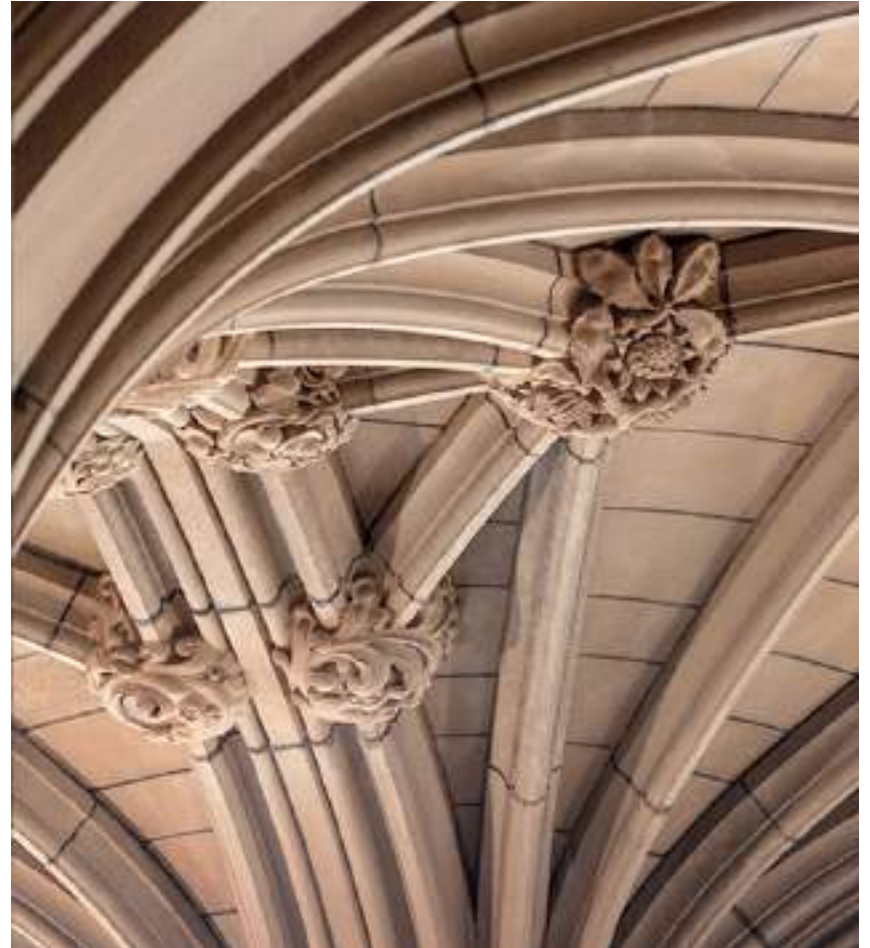
School of Computer Science



Agenda

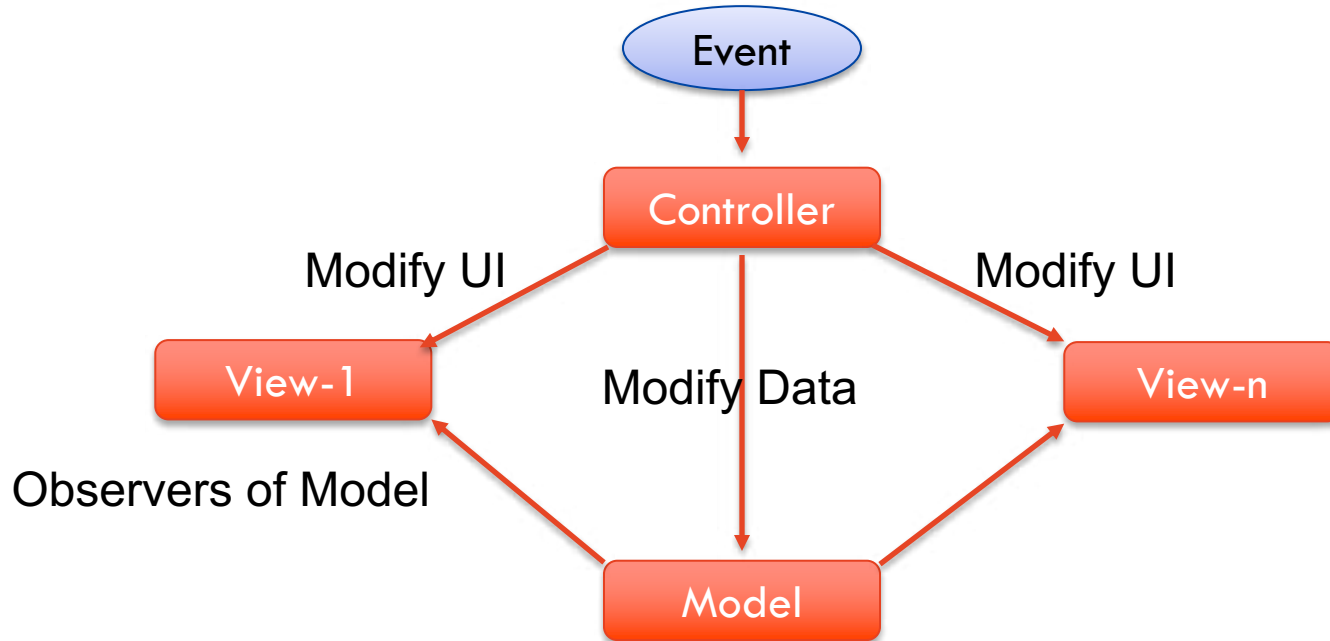
- Design Patterns for Interactive Applications

Model View Controller



Model View Controller (MVC)

“Splits user interface interaction into three distinct roles”



MVC – How it Works

- *Model*
 - An object that represents 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 UI

MVC – View and Model

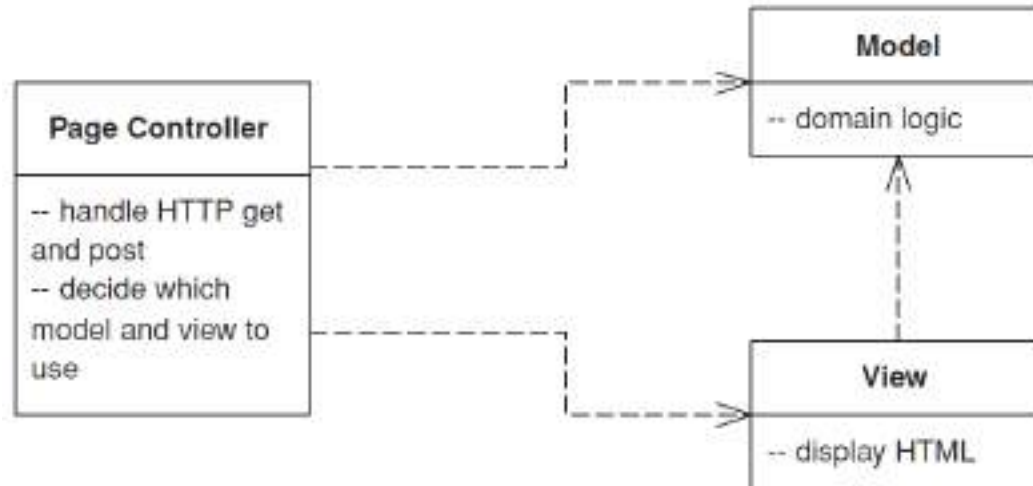
- Dependencies
 - The presentation depends on the model but not vice-versa
 - Maintain the presentation without changing the model
- Several presentations of a model on multiple windows/UIs.
- If the user makes changes to the model through one of the presentations, this should be reflected in the other presentations
- Use the Observer pattern (GoF); presentation is observer of the model

MVC –View and Controller

- Separation/dependency is less important
- Controller 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

MVC – Page Controller

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



(You are familiar with Spring Controller)

Page Controller – How It works

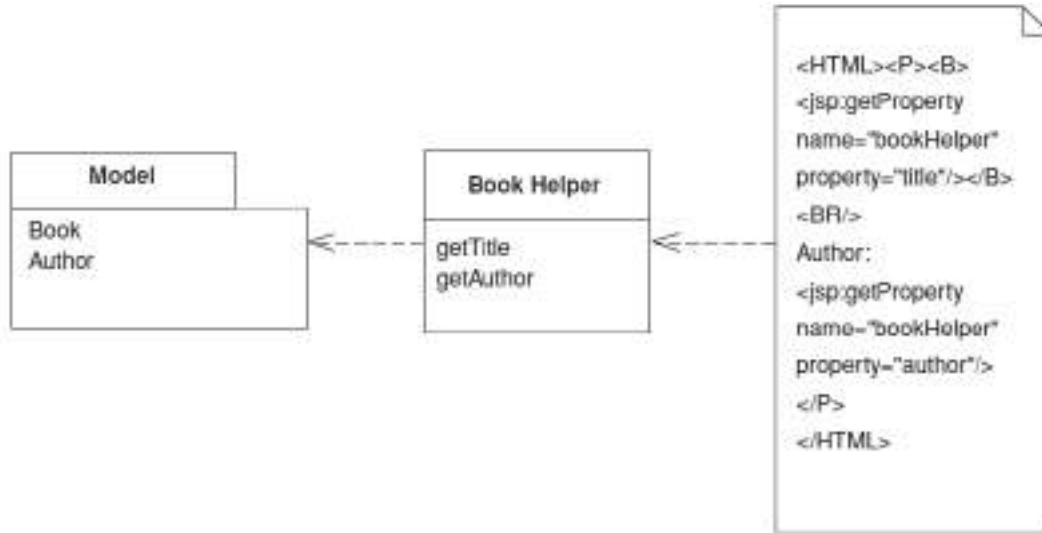
- One module on the Web server act as the controller for each page on the Web site (ideally)
- With dynamic content different pages might be sent, controllers link to each action (e.g., click a link or button)

Page Controller – 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)

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

(You are familiar with Spring Thymeleaf)

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)

Model-View-Presenter (MVP)

- Separates the application into
 - Model
 - View,
 - Presenter
- Presenter acts as the middleman between the View and Model, handling user input and updating the View and Model accordingly.
- Commonly used in desktop and mobile applications.

Model-View-Adapter (MVA)

- Separates the application into
 - Model
 - View,
 - Adapter (like a Mediator)
- Model and view only talks through adapter

Model-View-ViewModel (MVVM):

- Separates the application into three components
 - Model
 - View
 - ViewModel
- ViewModel is an abstraction of the View
 - exposes properties and commands for data binding
- Popular for developing WPF and UWP applications on the Windows platform, as well as mobile applications using Xamarin.

Flux:

- Emphasizes unidirectional data flow
 - Dispatcher
 - Stores
 - Views (React components)
- Actions are dispatched, and the Stores update the state based on the dispatched action
- Views listen for changes in the Stores and re-render as needed.
- Primarily used in web applications built with React.

Redux:

- State management library for JavaScript applications, often used in combination with React.
- The core idea is to maintain the application's state in a single, immutable data store.
- Actions are dispatched to modify the state, and Reducers process those actions and produce a new state.
- Components can subscribe to state changes and re-render as needed.

Supervising-Controller

- Separates the application into
 - View,
 - Controller
- Limited model

Passive View

- Separates the application into
 - View,
 - Controller
- Limited model and View

Presentation Abstraction Control (PAC)

- Separates the concerns of an application into a hierarchy of cooperating components, each of which are comprised of a Presentation, Abstraction, and Control.
- The PAC pattern seeks to decompose an application into a hierarchy of abstractions, and to achieve a consistent framework for constructing user interfaces at any level of abstraction within the application.

Data Communication/Context Interaction (DCI)

- Separates the concerns of an application into a hierarchy of cooperating components, each of which are comprised of a Presentation, Abstraction, and Control.
- The PAC pattern seeks to decompose an application into a hierarchy of abstractions, and to achieve a consistent framework for constructing user interfaces at any level of abstraction within the application.

Component Based

- Applications are built using reusable and self-contained components, which encapsulate both the logic and presentation.
- Popular in modern web development, particularly with frameworks like React, Angular, and Vue.js.