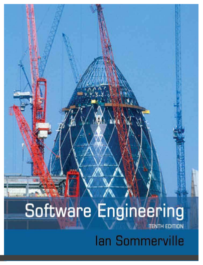
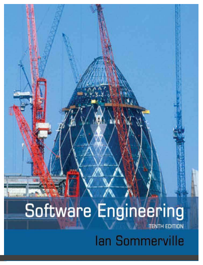


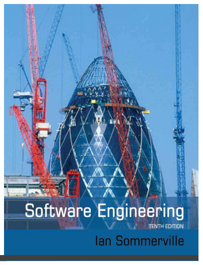
Design Patterns

Gangs of Four (GOF)



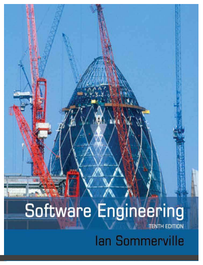
- In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled Design Patterns - Elements of Reusable Object-Oriented Software.
- These authors are collectively known as Gang of Four (GOF)
- According to these authors design patterns are primarily based on the following principles of object orientated design
  - Program to an interface not an implementation
  - Favor object composition over inheritance

Design Patterns



- Design Pattern
  - A description of a problem and its solution that you can apply to many similar programming situations

- Patterns:
  - facilitate reuse of good, tried-and-tested solutions
  - capture the structure and interaction between components
  - Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved



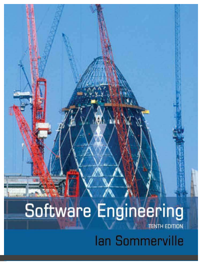
Why is this important?

- Using proven, effective design patterns can make you a better software *designer* & *coder*

- You will recognize commonly used patterns in others' code
  - Java API
  - Project team members
  - Ex-employees

In addition, different technologies have their own patterns:
Servlet patterns, GUI patterns, etc …

- And you'll learn when to apply them to your own code
  - experience reuse (as opposed to code reuse)
  - we want you thinking at the pattern level

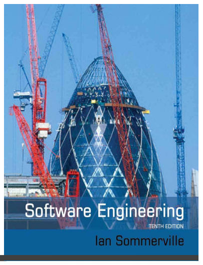


Common Design Patterns

Creational	Structural	Behavioral
• Factory	• Decorator	• Strategy
• Singleton	• Adapter	• Template
• Builder	• Facade	• Observer
• Prototype	• Flyweight	• Command
	• Component architecture	• Iterator
		• State

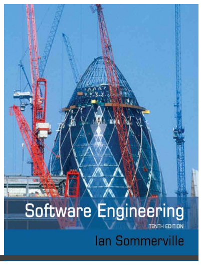
Textbook: Head First Design Patterns

Creational Design Pattern



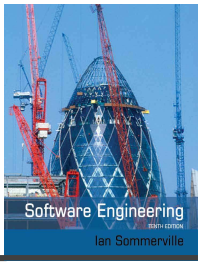
- Creational design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator.
- This gives program more flexibility in deciding which objects need to be created for a given use case.

The Factory Pattern



- Factories make stuff
- Factory classes make objects
- Shouldn't constructors do that?
  - factory classes employ constructors
- What's the point?
  - prevent misuse/improper construction
  - hides construction
  - provide API convenience
  - one stop shop for getting an object of a family type

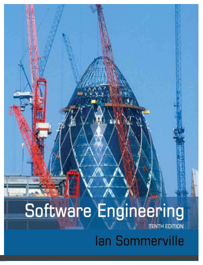
What objects do factories make?



- Typically objects of the same family
  - common ancestor
  - same apparent type
  - different actual type

- Factory Pattern in the Java API:
  - BorderFactory.createXXXBorder methods
    - return apparent type of Border
    - return actual types of BevelBorder, EtchedBorder, etc …
  - lots of factory classes in security packages

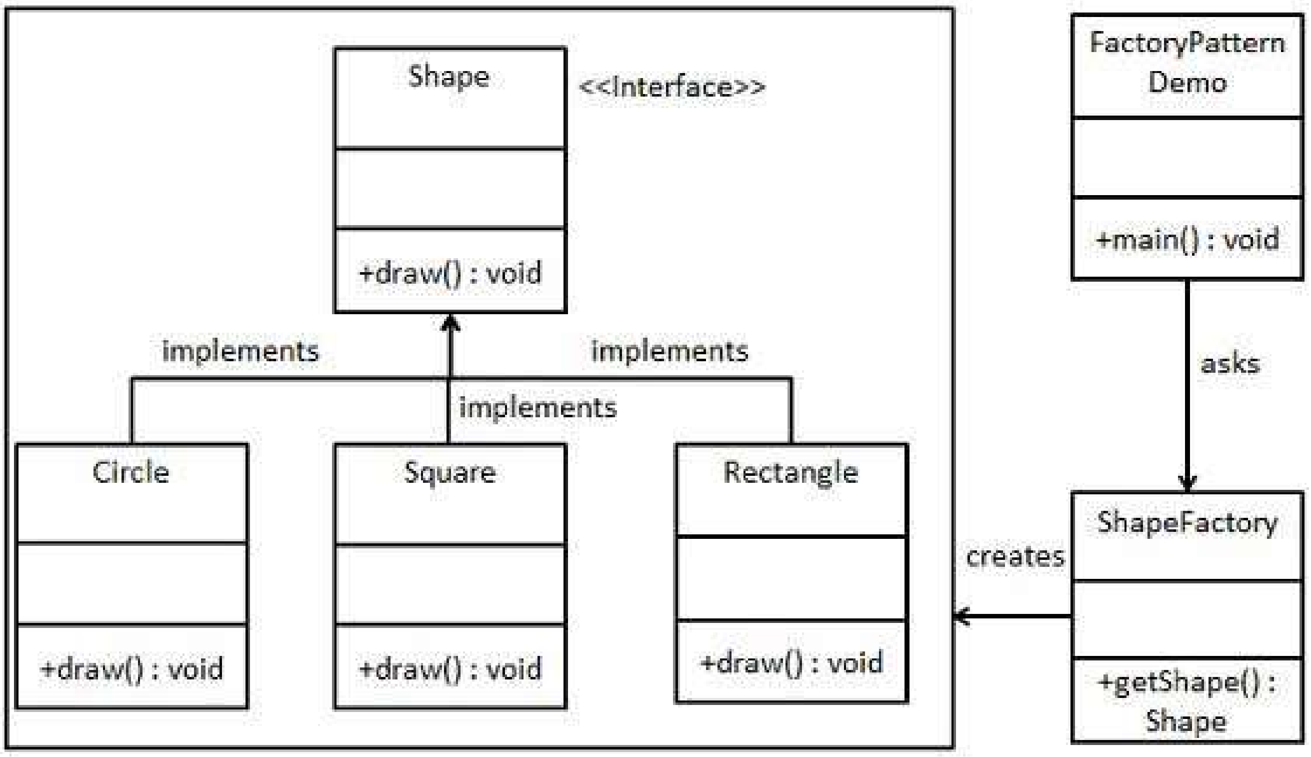
Factory Pattern Bonus



- The programmer using the Factory class never needs to know about the actual class/type
  - simplifies use for programmer
  - fewer classes to learn

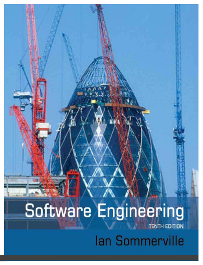
- Ex: Using BorderFactory, one only needs to know Border & BorderFactory
  - not TitledBorder, BeveledBorder, EtchedBorder, etc.

Factory Pattern Example



– https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

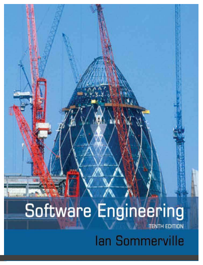
The Singleton Pattern



- Define a type where only one object of that type may be constructed:
  - make the constructor private.
  - singleton object favorable to fully static class, why?
    - can be used as a method argument
    - class can be extended

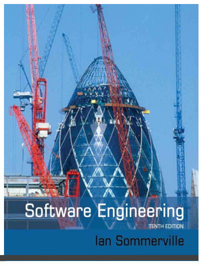
- What makes a good singleton candidate?
  - central app organizer class
  - something everybody needs

Example: The PropertiesManager Singleton



```
public class PropertiesManager
{
   private static PropertiesManager singleton
                                                          = null;

   private PropertiesManager() {}

   public static PropertiesManager
                       getPropertiesManager()
   {
       if (singleton == null)
       {   singleton = new PropertiesManager();
       }
       return singleton;
   }
…
}
```

What's so great about a singleton?

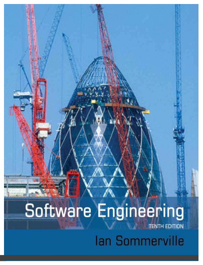


- Other classes may now easily USE the **`PropertiesManager`**

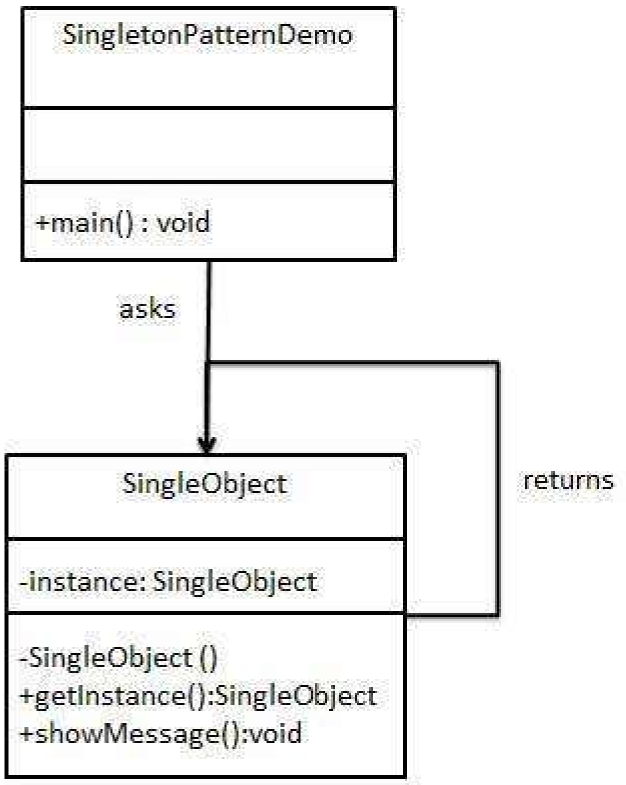
```
PropertiesManager singleton =
      PropertiesManager.getPropertiesManager();
Singleton.dowhaterver()
```

- Don’t have to worry about passing objects around
- Don’t have to worry about object consistency
- **Note:** the singleton is of course only good for classes that will never need more than one instance in an application

- https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm

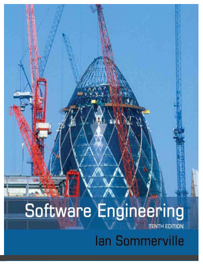


Singleton Pattern Example

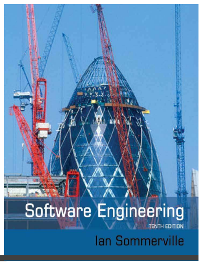


- https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm

The Builder Pattern



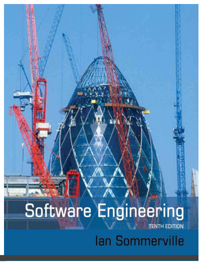
- Use the Builder Pattern to:
  - encapsulate the construction of a product
  - allow it to be constructed in steps
- Good for complex object construction
  - objects that require lots of custom initialized pieces
- **Scenario**:
  - build a JavaFX component
  - put it in its container
  - register it by id to be retrieved later
  - add a style class
  - etc.



The Builder Pattern

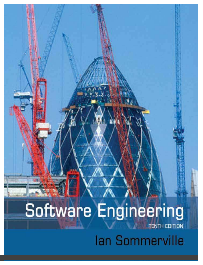
```
public class AppNodesBuilder {
   public CheckBox buildCheckBox(
   public ColorPicker buildColorPicker(
   public ComboBox buildComboBox(
   public HBox buildHBox(
   public Label buildLabel(
   public Slider buildSlider(
   public VBox buildVBox(
   public Button buildIconButton(
   public Button buildTextButton(
   public ToggleButton buildIconToggleButton(
   public ToggleButton buildTextToggleButton(
   public TextField buildTextField(
   public TableView buildTableView(
   public TableColumn buildTableColumn(
   …
}
```

Using a builder



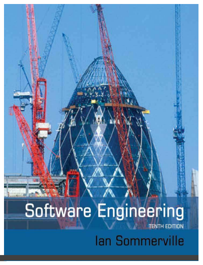
```
// INIT CONTROLS
HBox nameOwnerPane = tdlBuilder.buildHBox(…
HBox namePane = tdlBuilder.buildHBox(…
Label nameLabel = tdlBuilder.buildLabel(…
TextField nameTextField = tdlBuilder.buildTextField(…
HBox ownerPane = tdlBuilder.buildHBox(…
Label ownerLabel = tdlBuilder.buildLabel(…
TextField ownerTextField = tdlBuilder.buildTextField(…
```

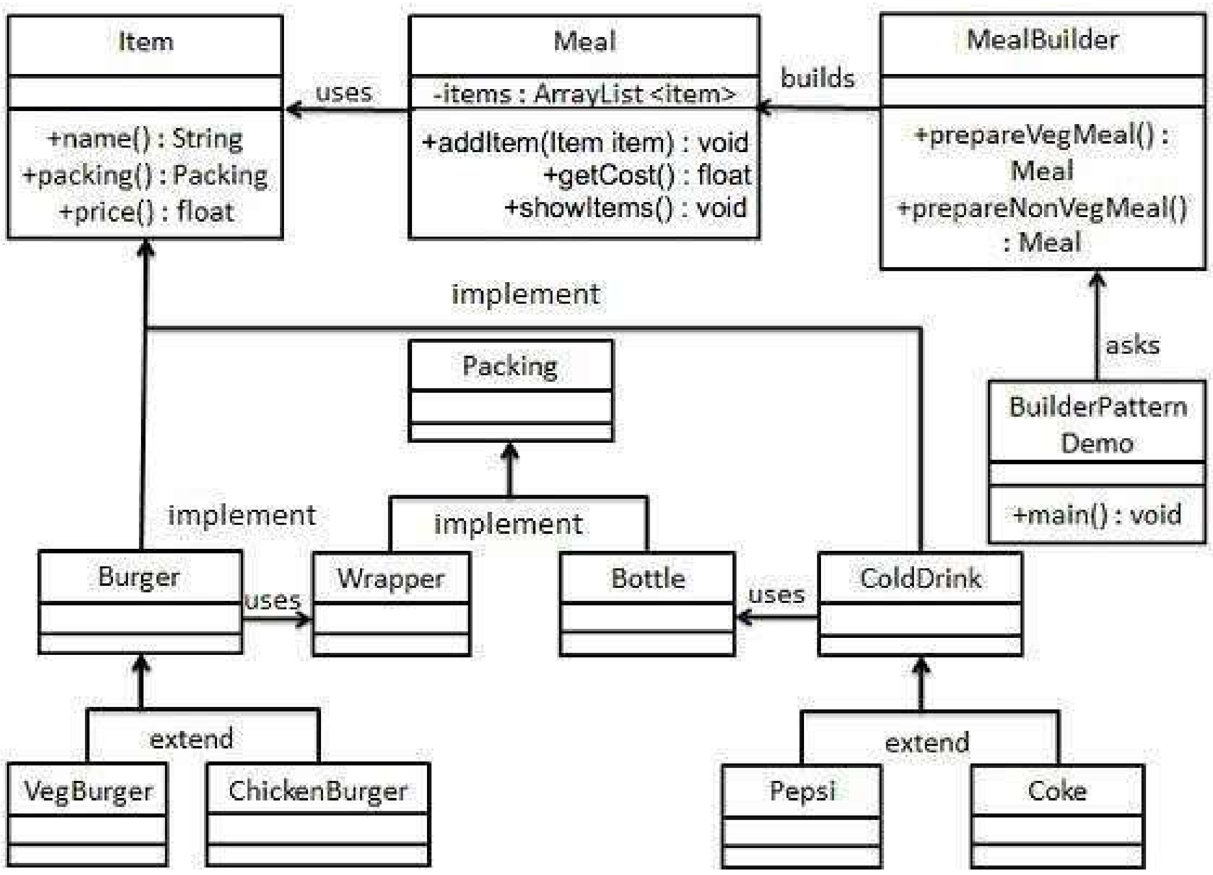
Builder Benefits



- Encapsulates the way a complex object is constructed.
- Allows objects to be constructed in a multistep and varying process (as opposed to one step factories).
- Hides the internal representation of the product from the client.
- Product implementations can be swapped in and out because the client only sees an abstract interface.

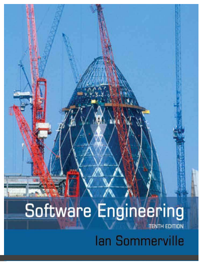
Builder Pattern Example





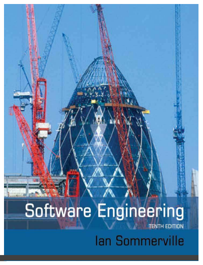
- https://www.tutorialspoint.com/design_pattern/builder_pattern.htm

The Prototype Pattern



- Use the Prototype Pattern when creating an instance of a given class is either expensive or complicated.
- This pattern involves implementing a prototype interface which tells to create a clone of the current object.
- This pattern is used when creation of object directly is costly.
- For example, an object is to be created after a costly database operation.
- We can cache the object, returns its clone on next request and update the database as and when needed thus reducing database calls.

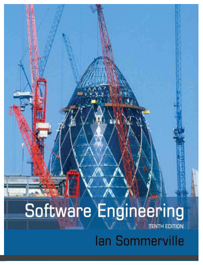
So what does the Prototype Pattern do?



- Allows you to make new instances by copying existing instances
  - in Java this typically means using the clone() method, or de-serialization when you need deep copies

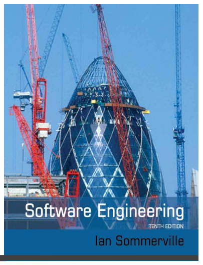
- A key aspect of this pattern is that the client code can make new instances without knowing which specific class is being instantiated

Prototype Benefits



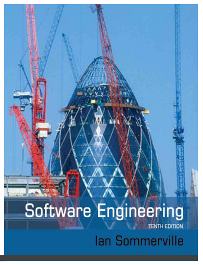
- Hides the complexities of making new instances from the client.
- Provides the option for the client to generate objects whose type is not known.
- In some circumstances, copying an object can be more efficient than creating a new object.

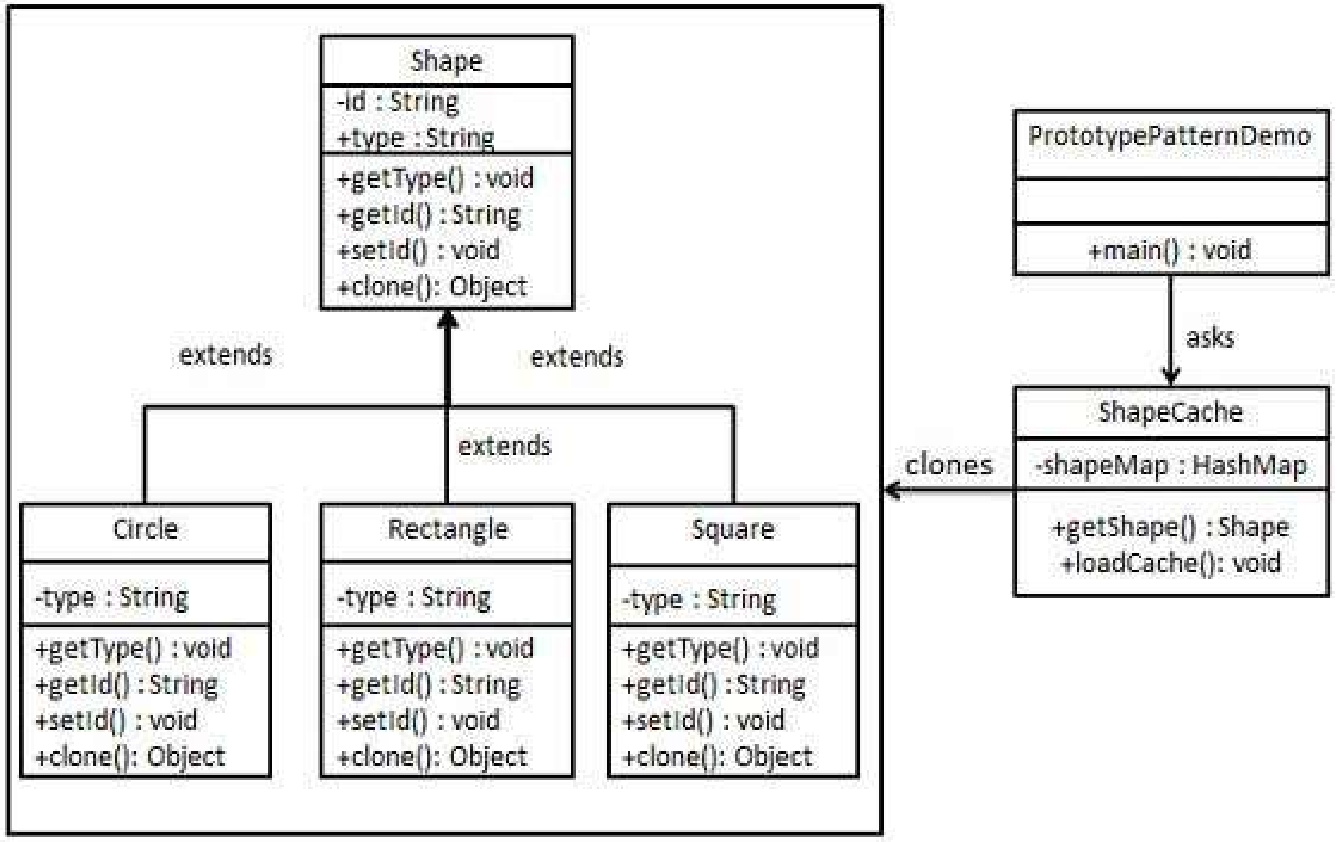
Prototype Uses and Drawbacks



- Prototype should be considered when a system must create new objects of many types in a complex class hierarchy.
- A drawback to using the Prototype is that making a copy of an object can sometimes be complicated.

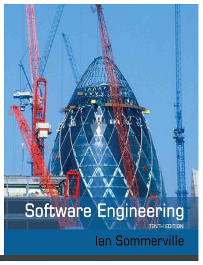
A Prototype Pattern Example



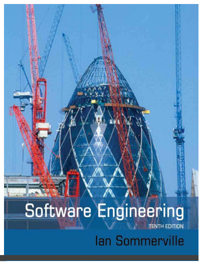


https://www.tutorialspoint.com/design_pattern/prototype_pattern.htm

Structural Design Patterns



- These design patterns concern class and object composition.
- Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.



The Decorator Pattern

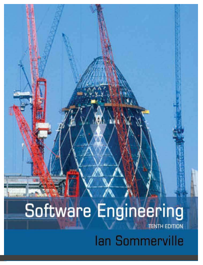
- Attaches additional responsibilities to an object ***dynamically***.
  - i.e. *decorating* an object

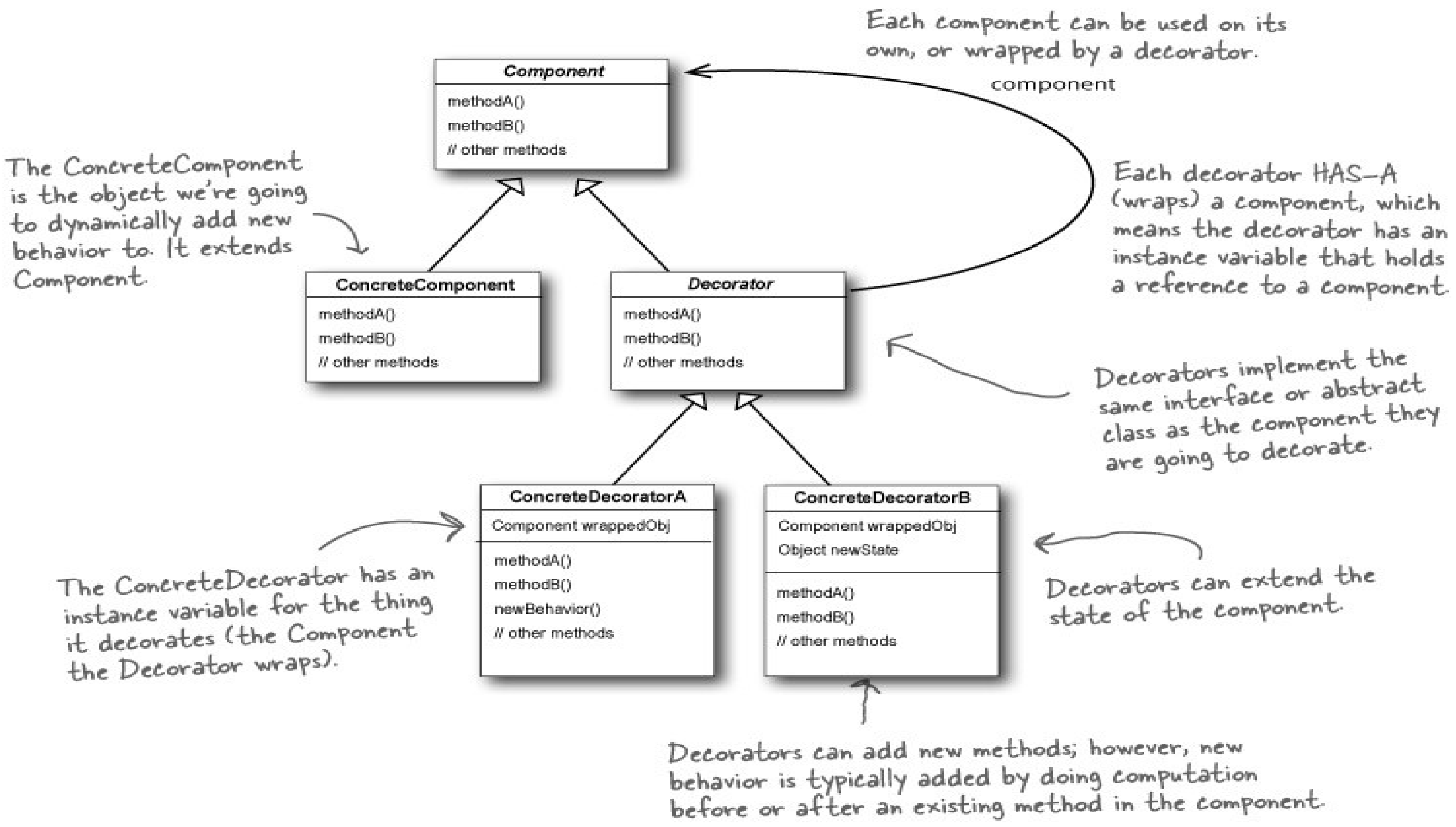
- Decorators provide a flexible alternative to sub-classing for extending functionality

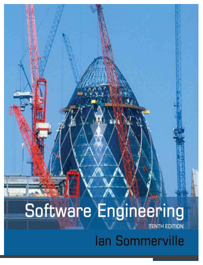
- How?
  - By ***wrapping*** an object

- Works on the principle that classes should be open to extension but closed to modification
- Allow classes to be easily extended to incorporate new behavior without modifying existing code

Decorators Override Functionality

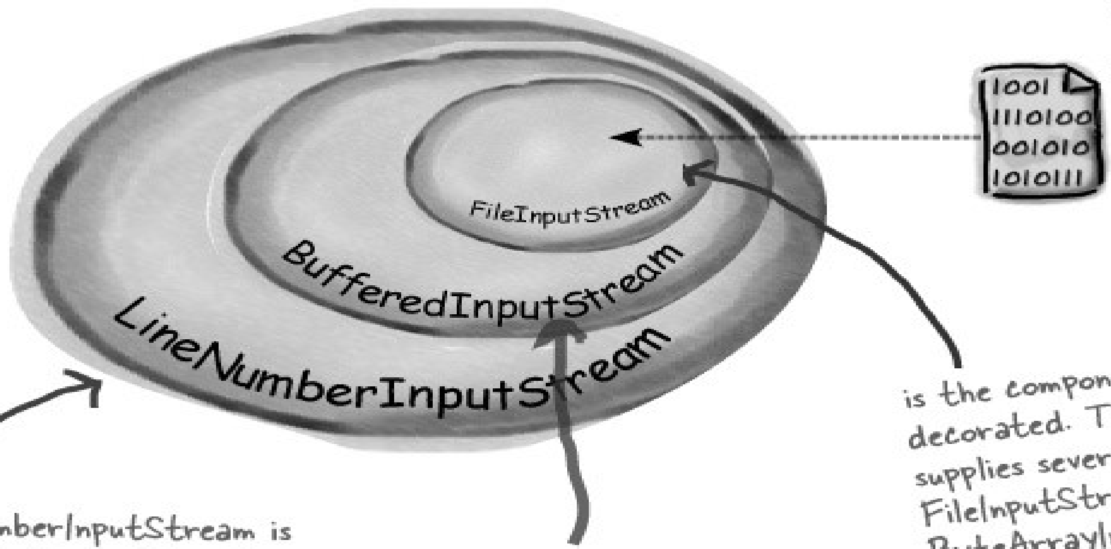



Each component can be used on its own, or wrapped by a decorator.
component
Component
methodA()
methodB()
// other methods
The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.
ConcreteComponent
methodA()
methodB()
// other methods
Decorator
methodA()
methodB()
// other methods
Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.
Decorators implement the same interface or abstract class as the component they are going to decorate.
ConcreteDecoratorA
Component wrappedObj
methodA()
methodB()
newBehavior()
// other methods
ConcreteDecoratorB
Component wrappedObj
Object newState
methodA()
methodB()
// other methods
The ConcreteDecorator has an instance variable for the thing it decorates (the Component the Decorator wraps).
Decorators can extend the state of the component.
Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.



Java’s IO Library

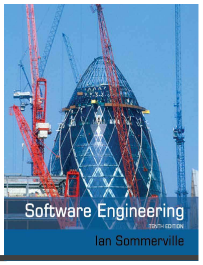
A text file for reading.



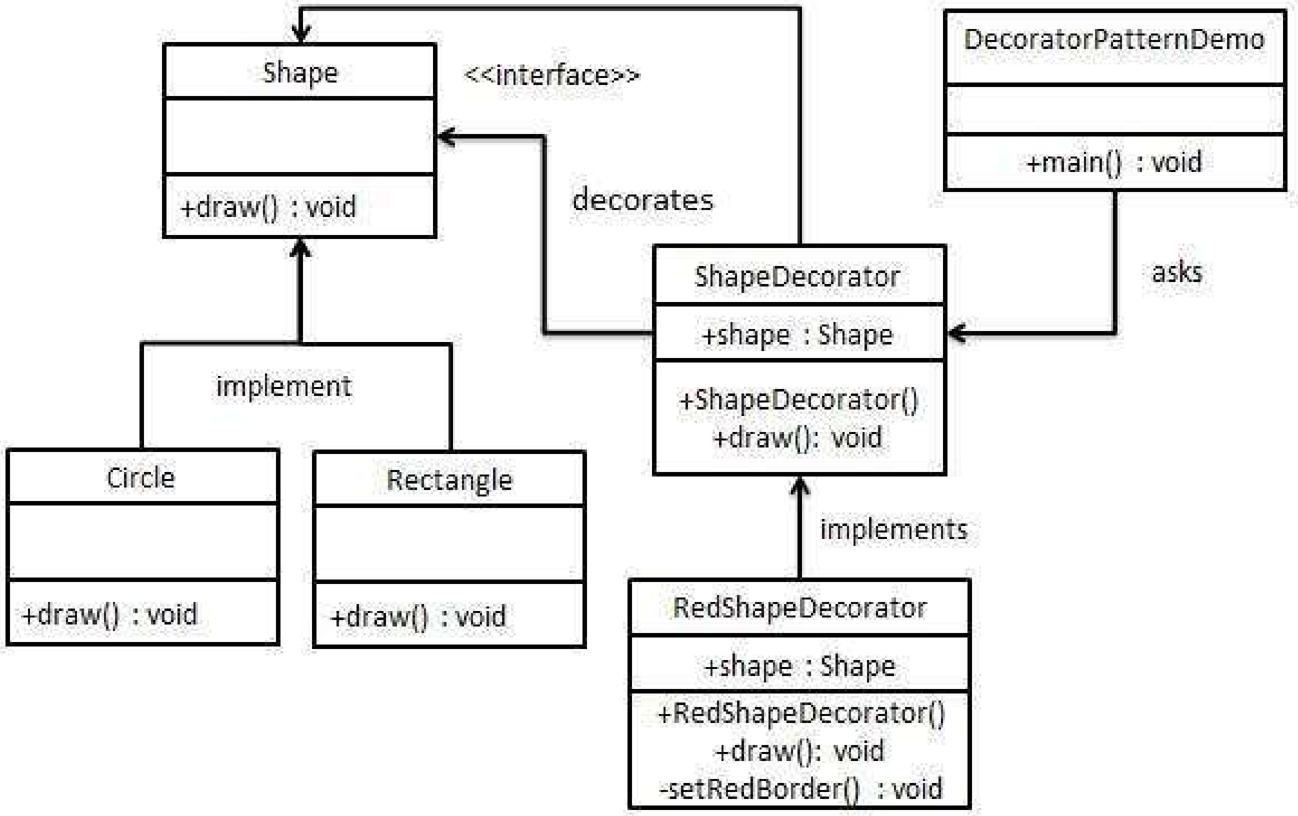
FileInputStream is the component that’s being decorated. The Java I/O library supplies several components, including FileInputStream, StringBufferInputStream, ByteArrayInputStream and a few others. All of these give us a base component from which to read bytes.

LineNumberInputStream is also a concrete decorator. It adds the ability to count the line numbers as it reads data.

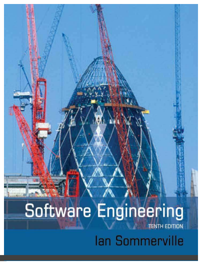
BufferedInputStream is a concrete decorator. BufferedInputStream adds behavior in two ways: it buffers input to improve performance, and also augments the interface with a new method readLine() for reading character-based input, a line at a time.



Decorator Pattern Example

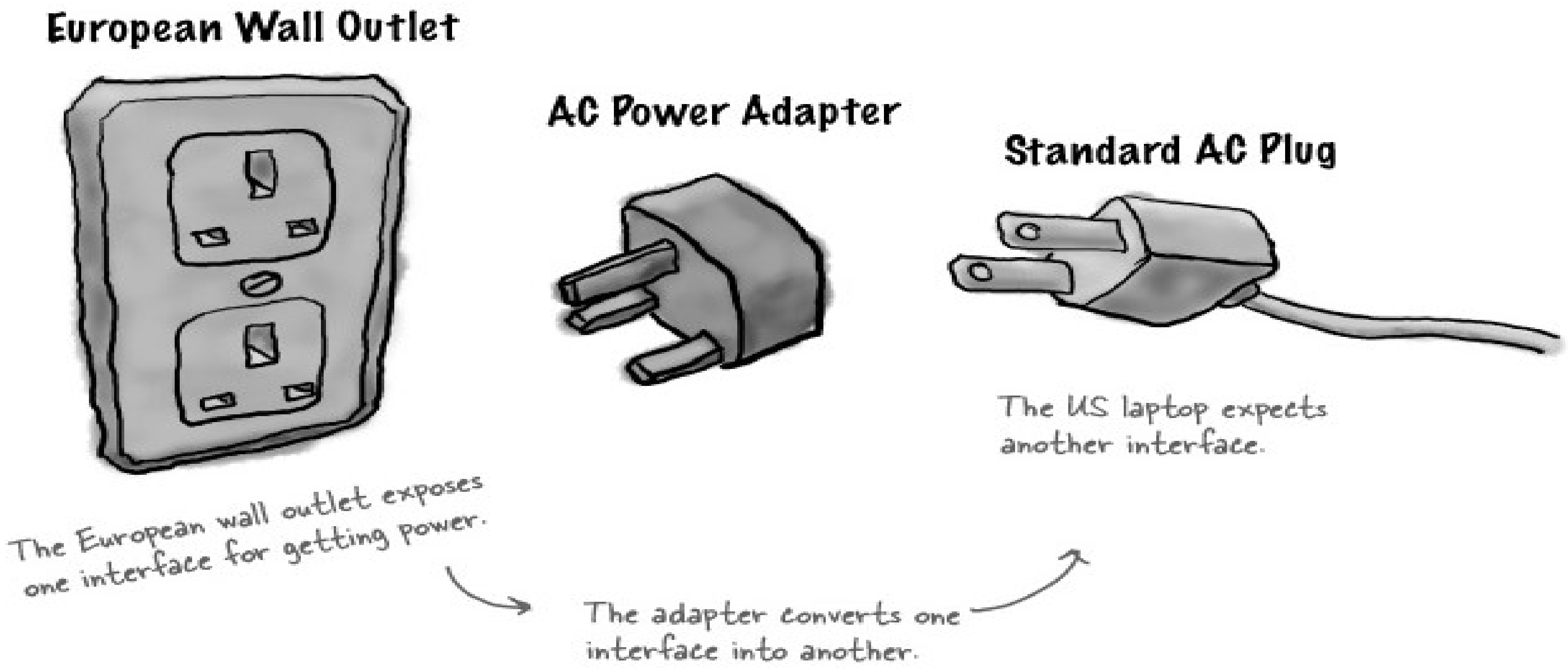


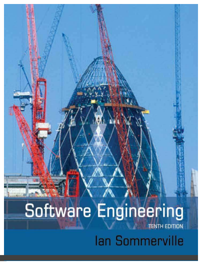
https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm



Ever been to Europe?

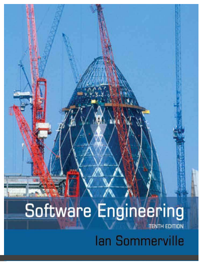
- This is an abstraction of the Adapter Pattern


European Wall Outlet
AC Power Adapter
Standard AC Plug
The US laptop expects another interface.
The European wall outlet exposes one interface for getting power.
The adapter converts one interface into another.



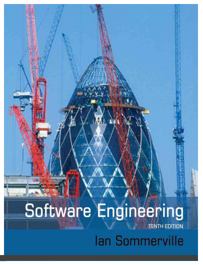
The Adapter Pattern

- Converts the interface of a class into another interface a client expects
- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
- Interfaces?
  - Do you know what a driver is?

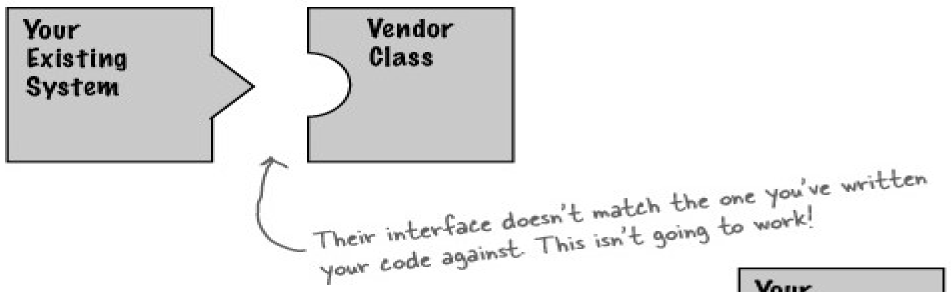


Adapter Scenario

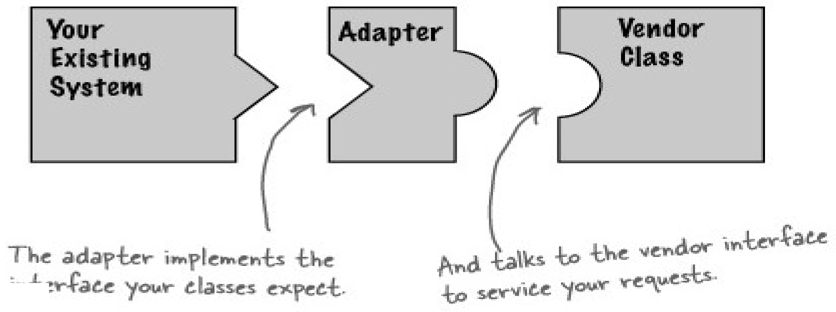
- You have an existing system
- You need to work a vendor library into the system
- The new vendor interface is different from the last vendor
- You really don't want to change your existing system
- Solution?
  - Make a class that adapts the new vendor interface into what the system uses



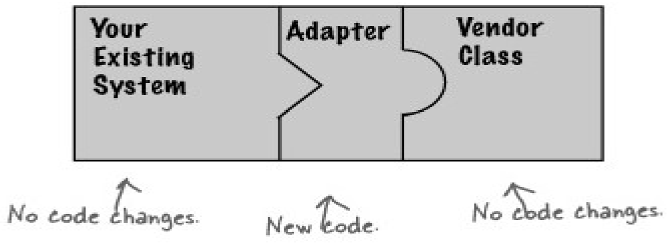
Adapter Visualized



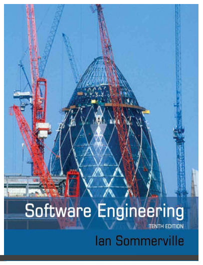
Your Existing System
Vendor Class
Their interface doesn't match the one you've written your code against. This isn't going to work!



Your Existing System
Adapter
Vendor Class
The adapter implements the interface your classes expect.
And talks to the vendor interface to service your requests.



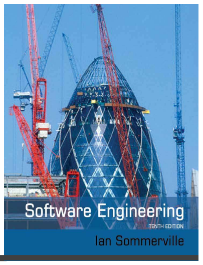
How do we do it?



- Ex: Driver
  - Existing system uses a driver via an interface
  - New hardware uses a different interface
  - Adapter can adapt differences

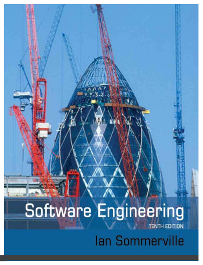
- Existing system HAS-A OldInterface
- Adapter implements OldInterface and HAS-A NewInterface
- Existing system calls OldInterface methods on adapter, adapter forwards them to NewInterface implementations

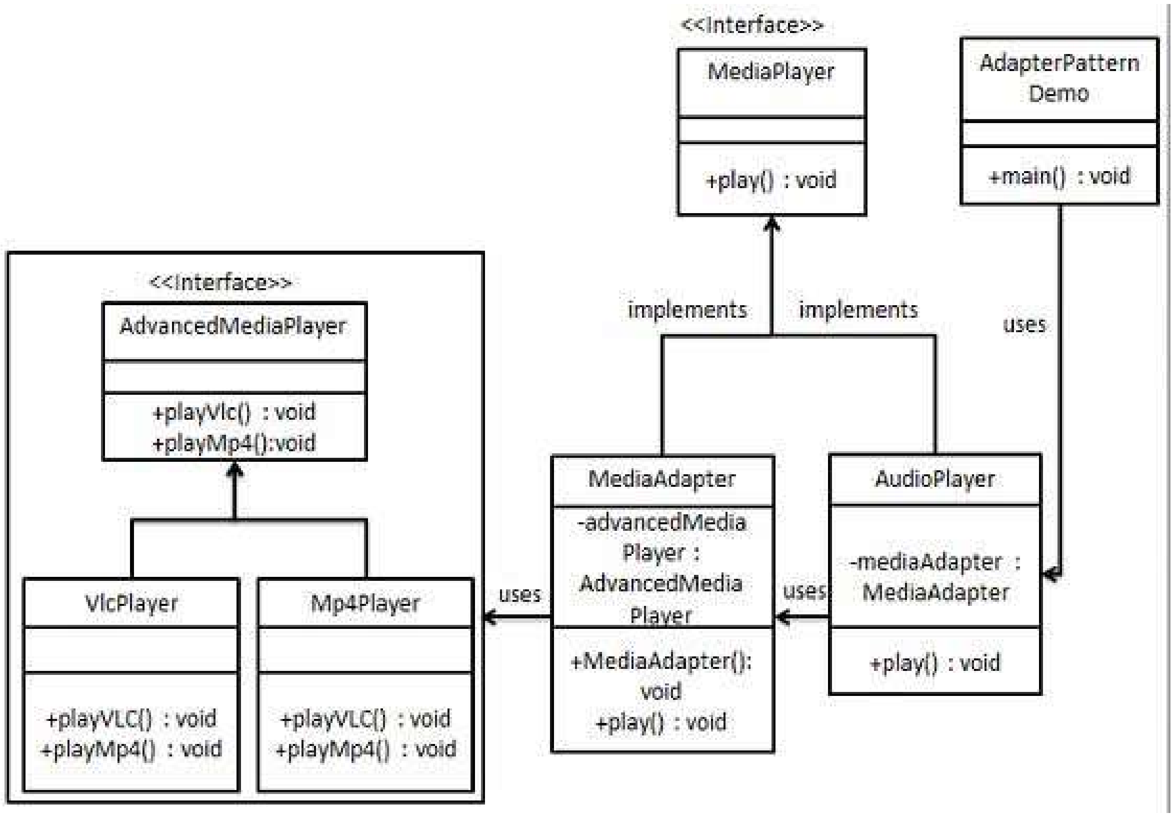
What’s good about this?



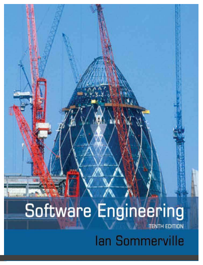
- Decouple the client from the implemented interface
- If we expect the interface to change over time, the adapter encapsulates that change so that the client doesn’t have to be modified each time it needs to operate against a different interface.

Adapter Pattern Example



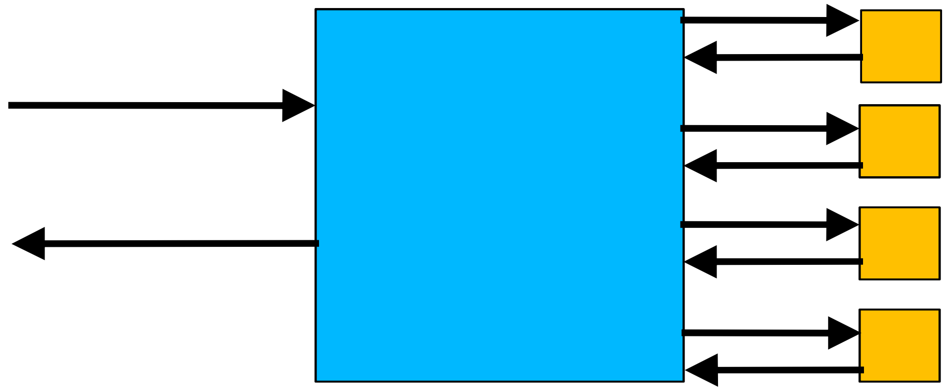


https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm

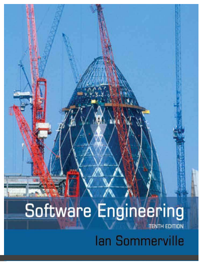


The Facade Pattern

- Provides a unified interface to a set of interfaces in a subsystem.
- The facade defines a higher-level interface that makes the subsystem easier to use
- Employs the principle of least knowledge
- A facade is a class or a group of classes hiding internal implementation/services from the user.
- The factory pattern is used when you want to hide the details on constructing instances.



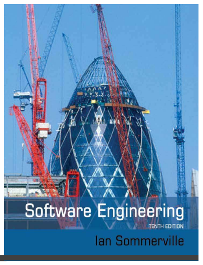
Scenario: We need a dialog



- Making a dialog can be a pain
  - setting up controls
  - providing layout
  - many common simple dialogs needed
  - applications like common presentation settings

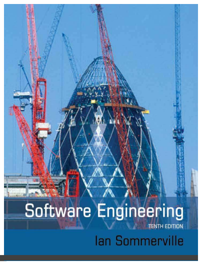
- Solution?
  - `AppDialogsFacade`

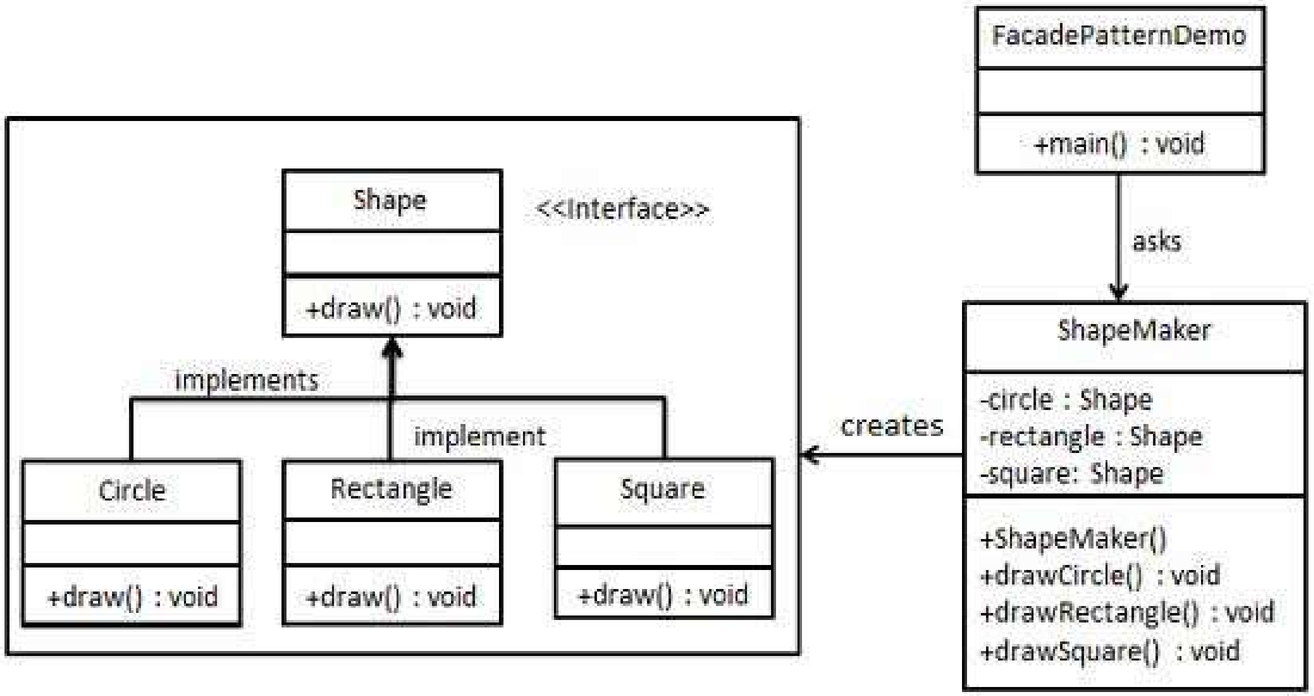
AppDialogsFacade



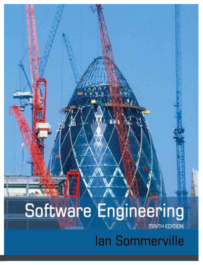
```
public class AppDialogsFacade {
    public static void showAboutDialog(
    public static void showExportDialog(
    public static void showHelpDialog(
    public static void showLanguageDialog(
    public static void showMessageDialog(
    public static File showOpenDialog(
    public static File showSaveDialog(
    public static void showStackTraceDialog(
    public static String showTextInputDialog(
    public static String showWelcomeDialog(
    public static ButtonType showYesNoCancelDialog(
}
```

Tutorial





https://www.tutorialspoint.com/design_pattern/facade_pattern.htm



Which is which?

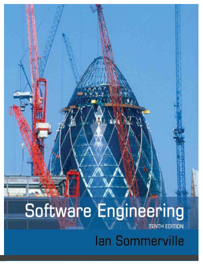
- Converts one interface to another
- Makes an interface simpler
- Doesn’t alter the interface, but adds responsibility

A) Decorator

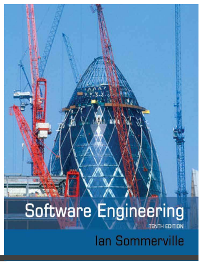
B) Adapter

C) Facade

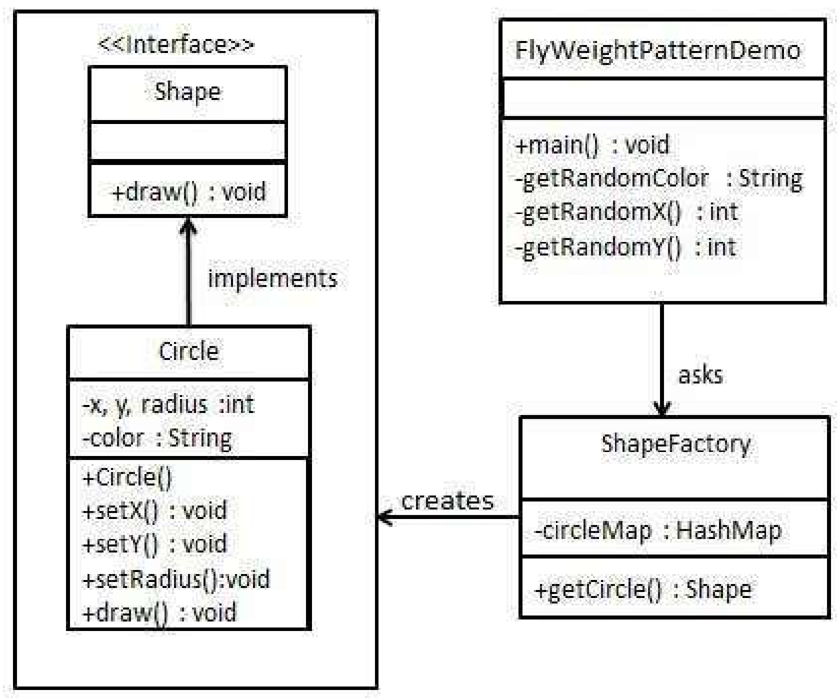
The Flyweight Pattern



- A “neat hack”
- Allows one object to be used to represent many identical instances
  - Flyweights must be immutable.
  - Flyweights depend on an associated table
    - maps identical instances to the single object that represents all of them
- Used in processing many large documents
  - search engines
  - a document as an array of immutable Strings
  - repeated Words would share objects
  - just one object for “the” referenced all over the place
    - use static Hashtable to store mappings
- The flyweight pattern is used to minimize the amount of memory used when you need to create a large number of similar objects. It accomplishes this by sharing instances.
- The name derives from the weight classification in boxing, but refers to the little amount of memory. That is, memory = weight.

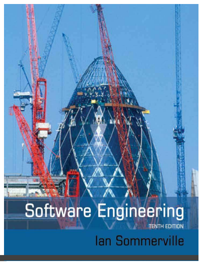


Flyweight Pattern Example



https://www.tutorialspoint.com/design_pattern/flyweight_pattern.htm

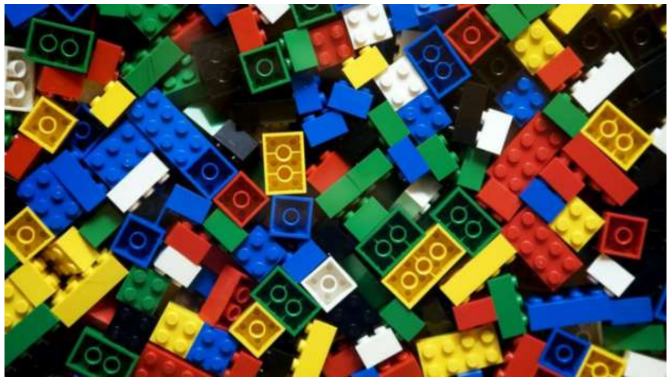
A Component Architecture



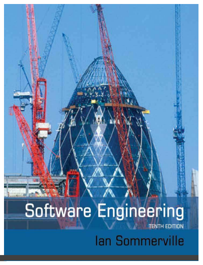
- System uses a set of pluggable ***components***

- **Each component:**
  - can be plugged in
  - can be updated
  - can be replaced

  independently of the other components



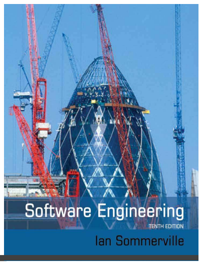
`AppTemplate` uses Components



- With Default behavior:
  - `AppFileModule`
  - `AppFoolproofModule`
  - `AppGUIModule`
  - `AppLanguageModule`
  - `AppRecentWorkModule`

- With Custom behavior:
  - `AppClipboardComponent`
  - `AppDataComponent`
  - `AppFileComponent`
  - `AppWorkspaceComponent`

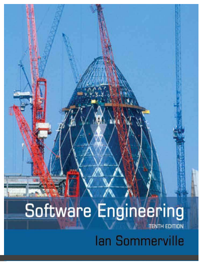
Behavioral Design Pattern



- These design patterns are specifically concerned with communication between objects.
- Increase flexibility in carrying out communication between objects.

- https://www.youtube.com/watch?v=kiTDR0YoIqA

The Strategy Pattern



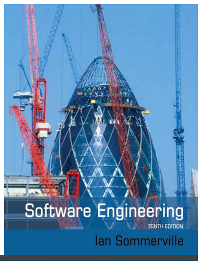
- How does it work?
  - defines a family of algorithms, encapsulates each one, and makes them interchangeable
  - lets the algorithm vary independently from the clients that use them

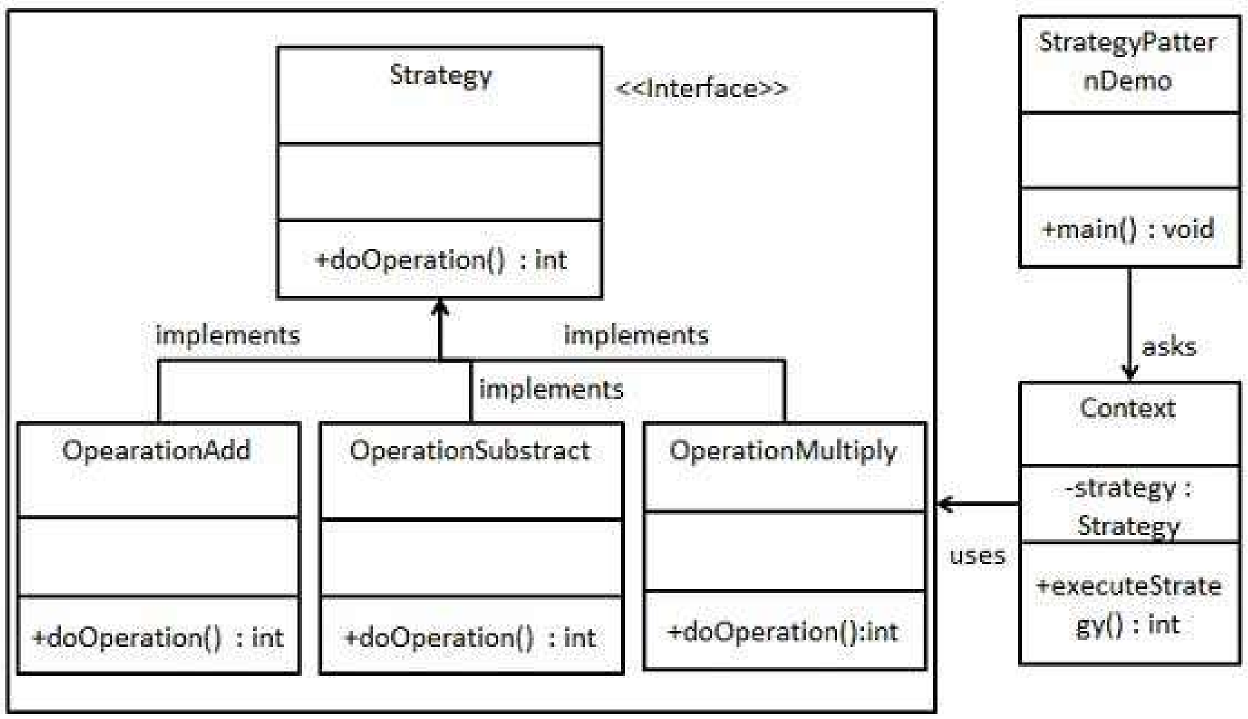
- A class behavior or its algorithm can be changed at run time.

- In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object.
- The strategy object changes the executing algorithm of the context object.

- Classes can be composed (HAS-A) of the interface type
  - the interface type is the apparent type
  - the actual type can be determined at run-time

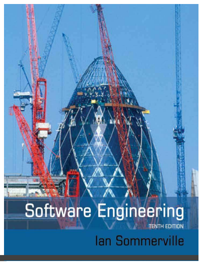
Strategy Pattern Example



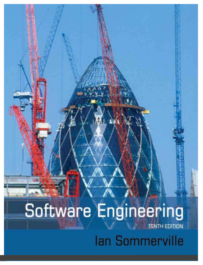


https://www.youtube.com/watch?v=QKeL46JoDU4
https://www.tutorialspoint.com/design_pattern/strategy_pattern.htm

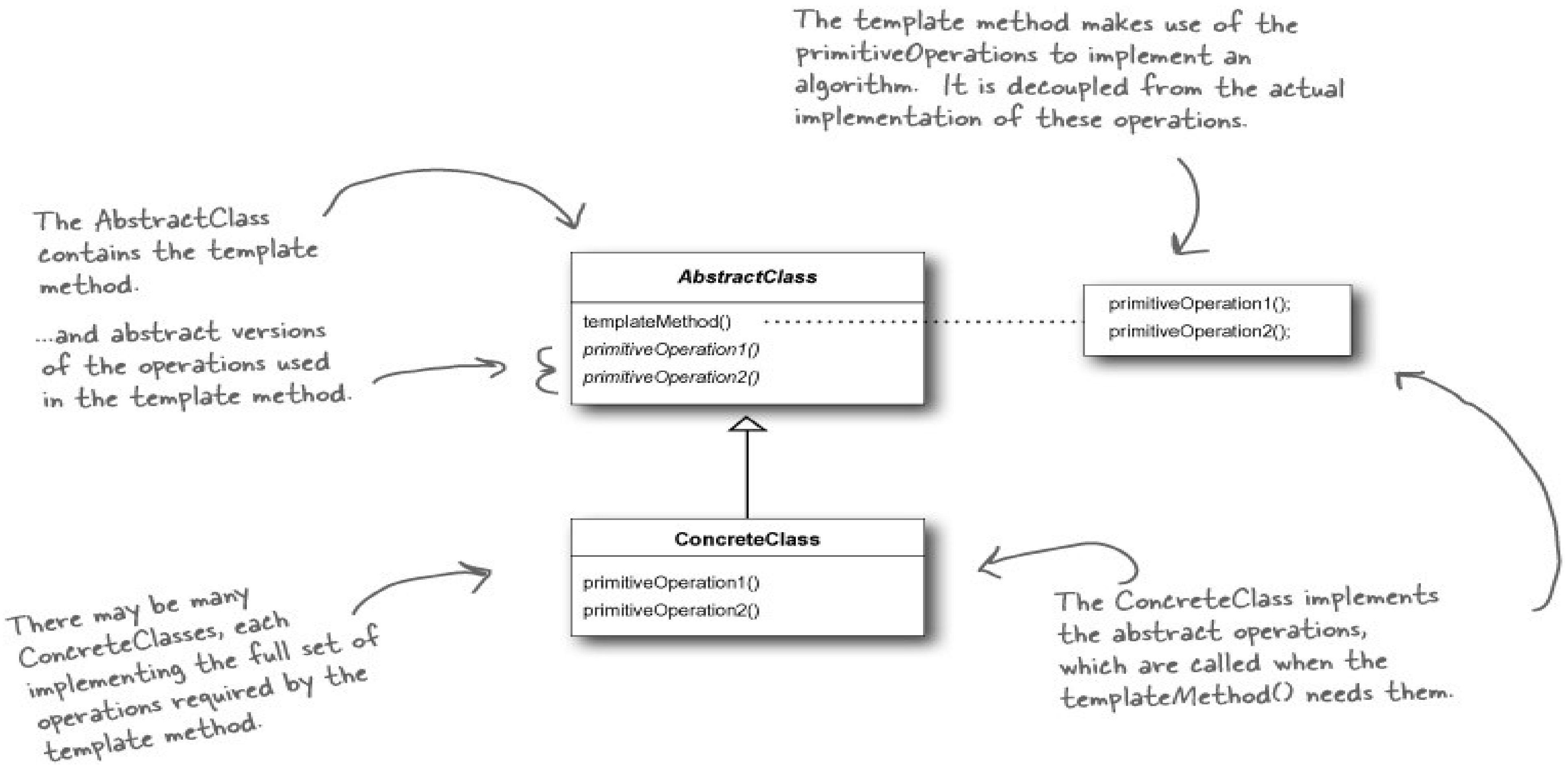
Template Method Pattern

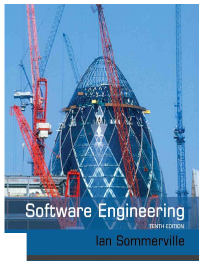


- Defines the skeleton of an algorithm in a method, deferring some steps to subclasses.
- Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- A template for an ***algorithm***
- https://www.youtube.com/watch?v=bPVDEkl1z0o

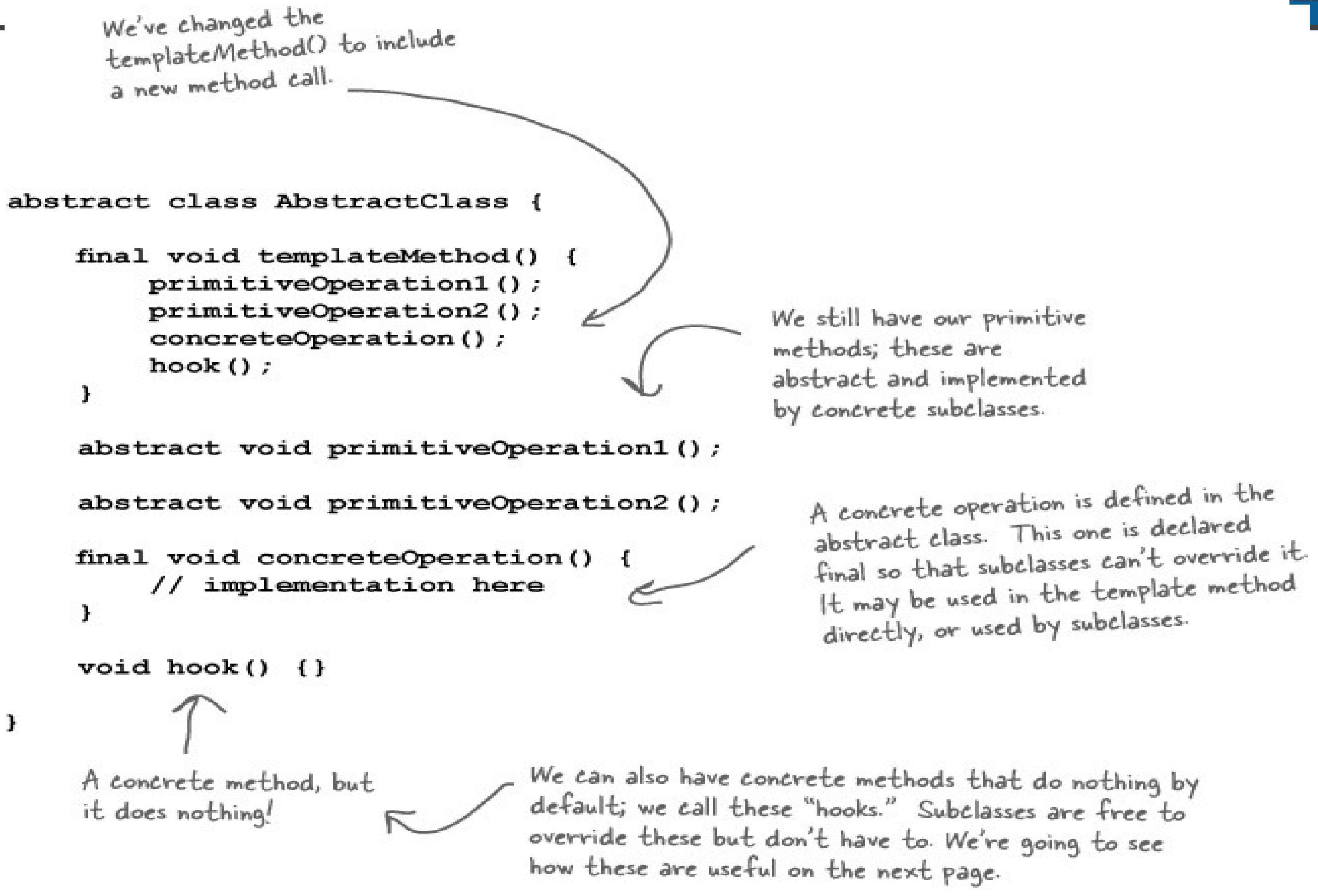


Template Method Pattern


The template method makes use of the primitiveOperations to implement an algorithm. It is decoupled from the actual implementation of these operations.
The AbstractClass contains the template method.
...and abstract versions of the operations used in the template method.
AbstractClass
templateMethod()
primitiveOperation1()
primitiveOperation2()
primitiveOperation1();
primitiveOperation2();
ConcreteClass
primitiveOperation1()
primitiveOperation2()
There may be many ConcreteClasses, each implementing the full set of operations required by the template method.
The ConcreteClass implements the abstract operations, which are called when the templateMethod() needs them.



Template Method Pattern



We've changed the templateMethod() to include a new method call.

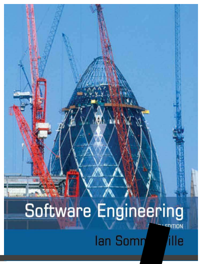
```
abstract class AbstractClass {

    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
        hook();
    }

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    final void concreteOperation() {
        // implementation here
    }

    void hook() {}

}
```

We still have our primitive methods; these are abstract and implemented by concrete subclasses.

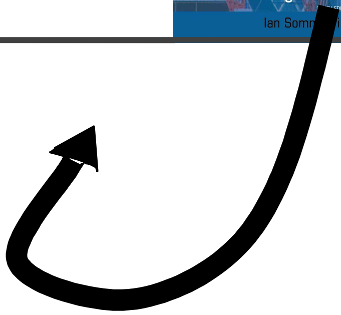
A concrete operation is defined in the abstract class. This one is declared final so that subclasses can't override it. It may be used in the template method directly, or used by subclasses.

A concrete method, but it does nothing!

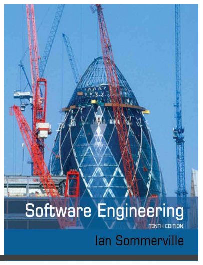
We can also have concrete methods that do nothing by default; we call these "hooks." Subclasses are free to override these but don't have to. We're going to see how these are useful on the next page.



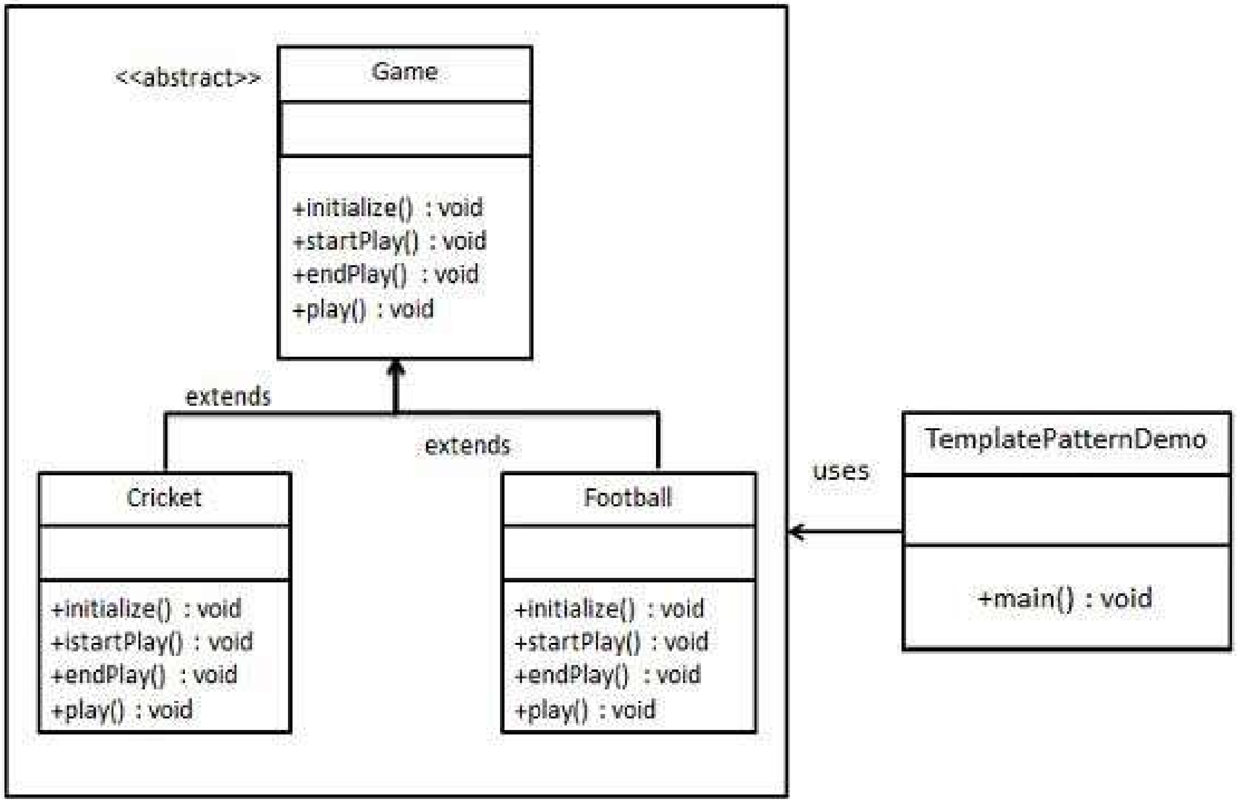
What’s a hook?



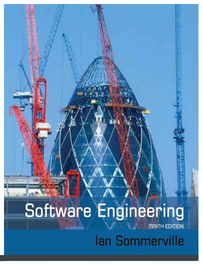
- A type of method
- Declared in the abstract class
  - only given an empty or default implementation
- Gives subclasses the ability to “hook into” the algorithm at various points, if they wish
  - a subclass is also free to ignore the hook.



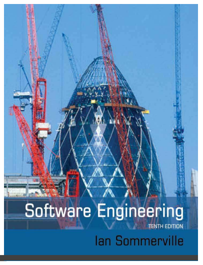
Template Design Pattern Example



The Observer Pattern

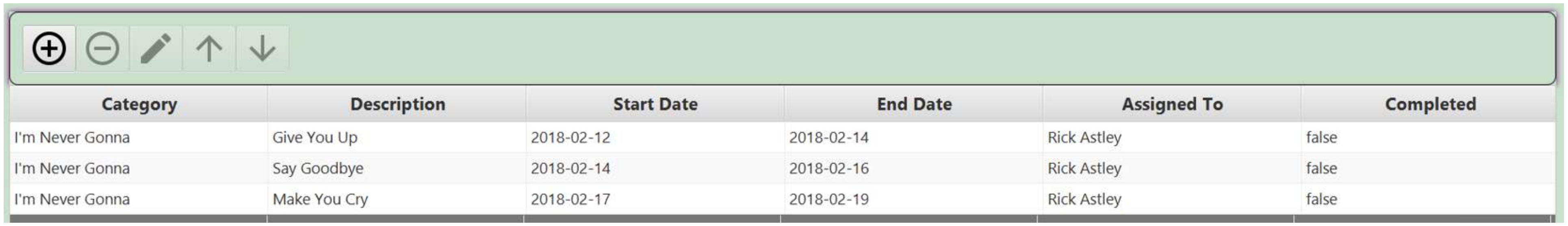


- Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.
- Hmm, where have we seen this?
  - in our GUI
- State Manager class maintains application's state
  - call methods to change app's state
  - app's state change forces update of GUI



TableView

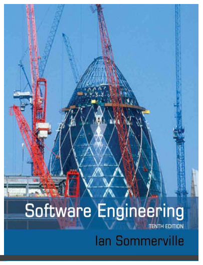
- Used to display spreadsheets and tables



Category	Description	Start Date	End Date	Assigned To	Completed
I'm Never Gonna	Give You Up	2018-02-12	2018-02-14	Rick Astley	false
I'm Never Gonna	Say Goodbye	2018-02-14	2018-02-16	Rick Astley	false
I'm Never Gonna	Make You Cry	2018-02-17	2018-02-19	Rick Astley	false

- How is the table data stored?
  - in an ***`ObservableList`***
  - we call this the table's data ***model***

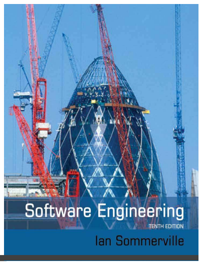
Editing the table



- To edit the table, you must go through the model:

```
TableView table = new TableView(…
ObservableList<DataPrototype> model = table.getItems();

//  Add Data
model.add(…

// Remove Data
model.remove(…

// Change Data
DataPrototype data = model.get(…
data.set(…

// UPDATING THE MODEL (ObservableList) AND/OR THE DATA (DataPrototype)
// WILL AUTOMATICALLY UPDATE THE VIEW (TableView) THANKS TO MVC!
```

Complex Controls have their own States

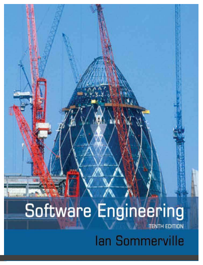


- Tables, trees, lists, combo boxes, etc.
  - data is managed separately from the view
  - when state changes, view is updated

- This is called MVC
  - Model
  - View
  - Controller

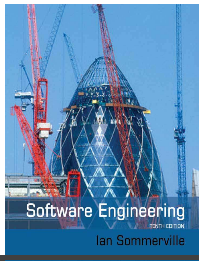
- MVC ***employs*** the Observer Pattern

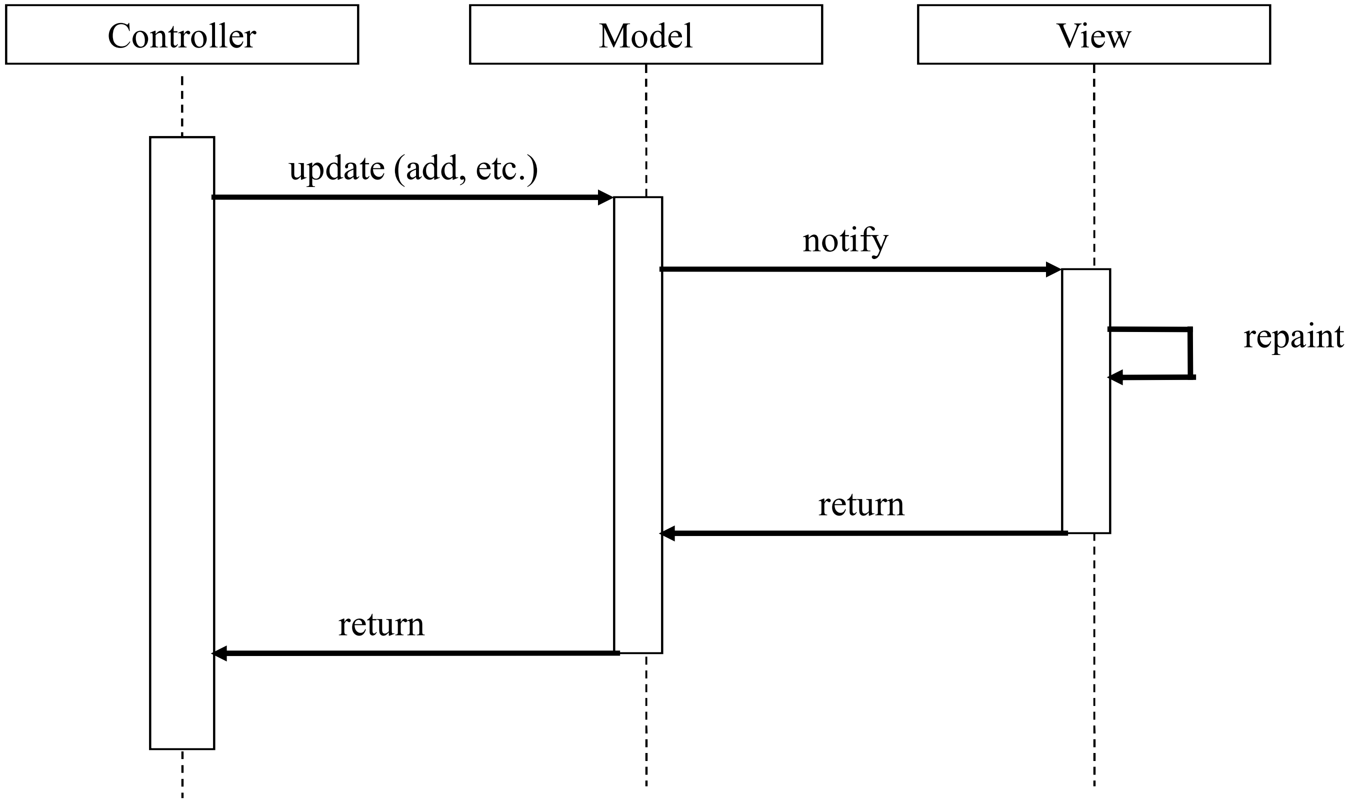
MVC *employs* the Observer Pattern

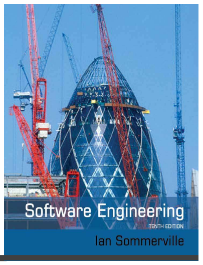


- **Model**
  - data structure, no visual representation
  - notifies views when something interesting happens
- **View**
  - visual representation
  - views attach themselves to model in order to be notified
- **Controller**
  - event handler
  - listeners that are attached to view in order to be notified of user interaction (or otherwise)

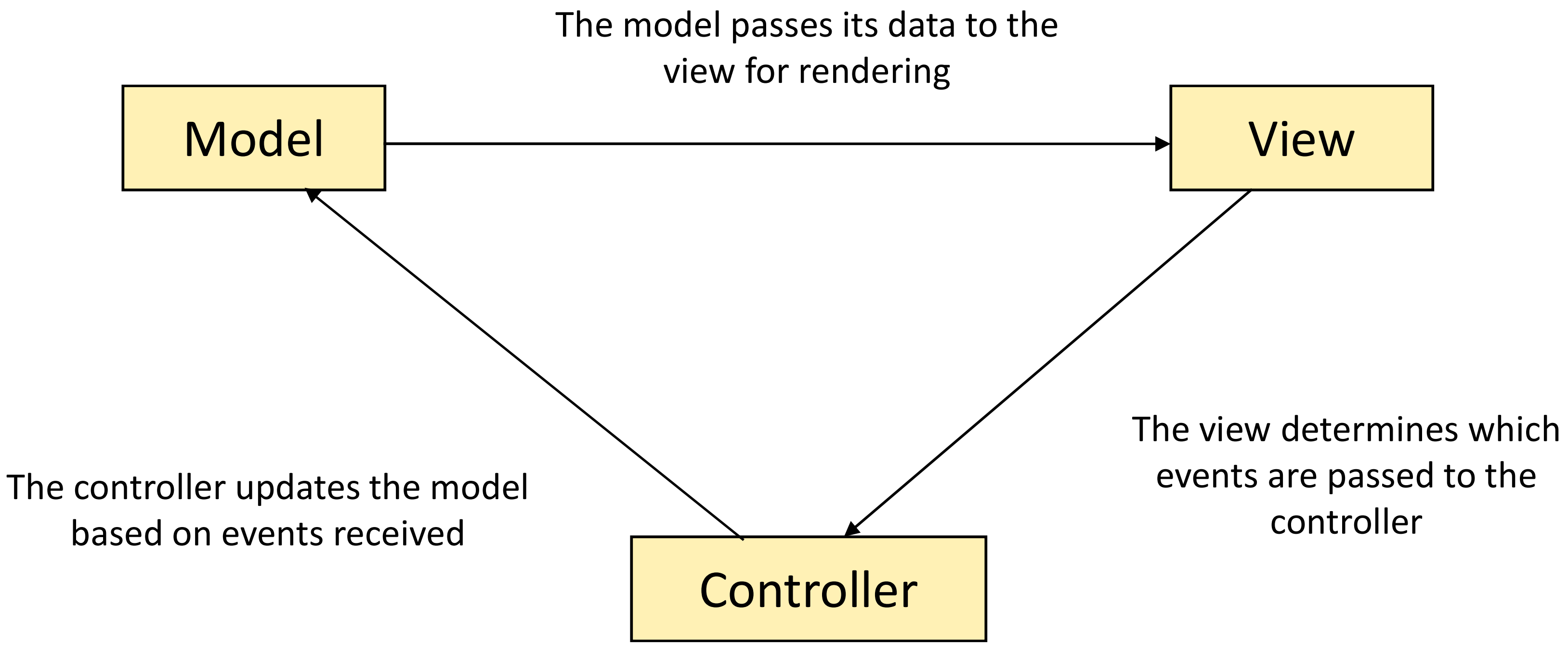
- **MVC Interaction**
  - controller updates model
  - model tells view that data has changed
  - view redrawn


Software Engineering
TENTH EDITION
Ian Sommerville

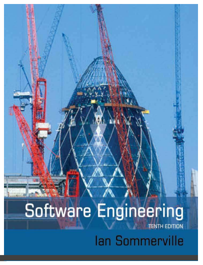

Controller
Model
View
update (add, etc.)
notify
repaint
return
return

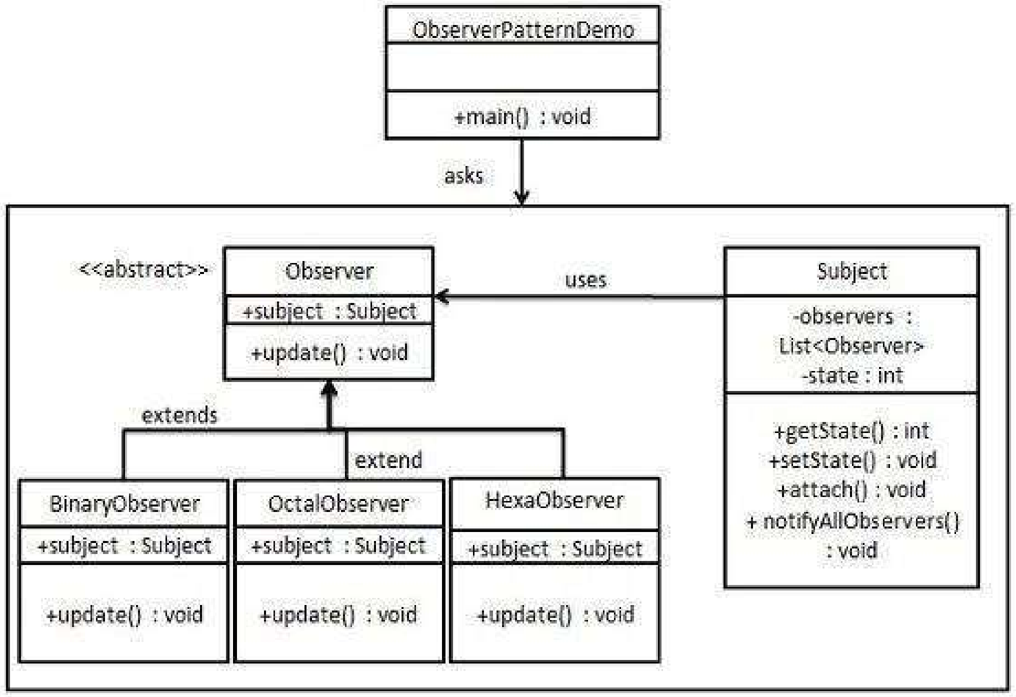


MVC Architecture

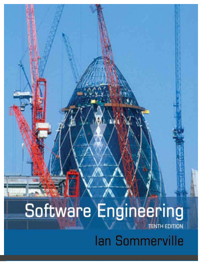

The model passes its data to the view for rendering
Model
View
The controller updates the model based on events received
The view determines which events are passed to the controller
Controller

Observer Design Pattern Example





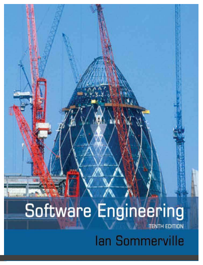
https://www.tutorialspoint.com/design_pattern/observer_pattern.htm



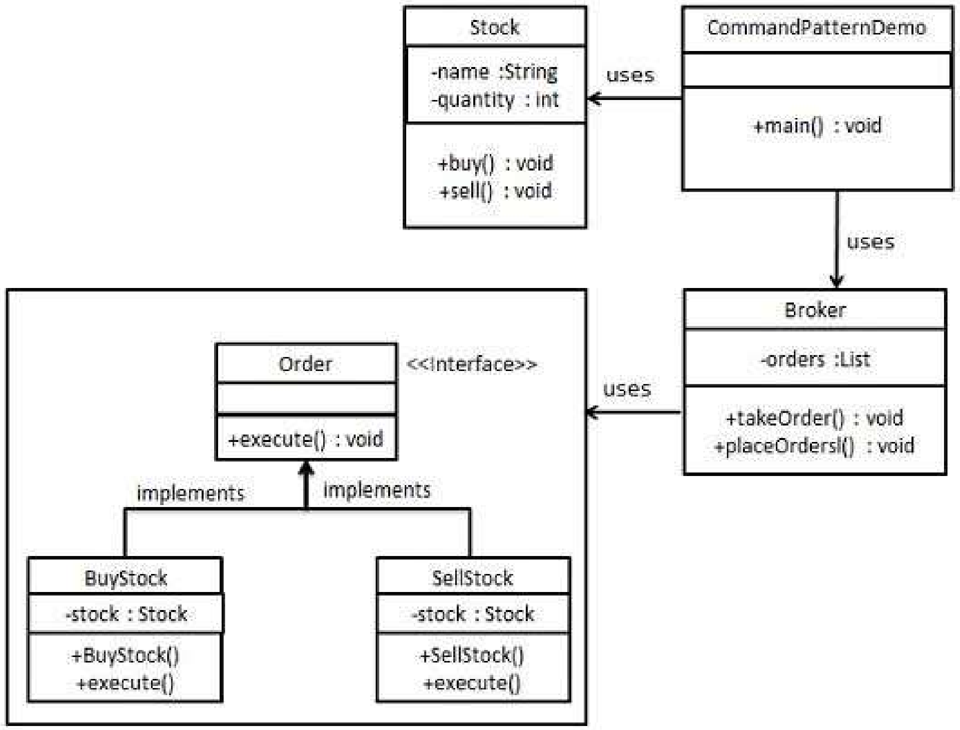
Command Abstraction

- For many GUIs, a single function may be triggered by many means (e.g., keystroke, menu, button, etc…)
  - we want to link all similar events to the same listener
- The information concerning the command can be abstracted to a **separate command object**
- Common Approach:
  - specify a String for each command
    - have listener respond to each command differently
  - ensure commands are handled in a uniform way
  - commands can be specified inside a text file
  - The Command Pattern

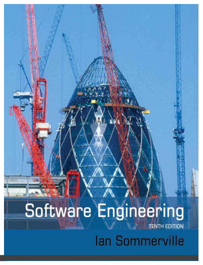
https://www.youtube.com/watch?v=iNKvqMiPtmY



Command Design Pattern Example



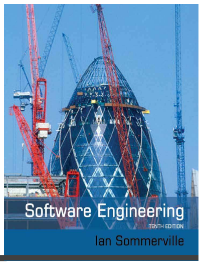
https://www.tutorialspoint.com/design_pattern/command_pattern.htm



Iteration

- What’s the problem?
  - you have to perform some operation on a sequence of elements in a given data structure
- Solution:
  - Iterator Pattern a.k.a. Iteration Abstraction
    - iterate over a group of objects without revealing details of how the items are obtained

https://www.youtube.com/watch?v=Pganyj1dVVU



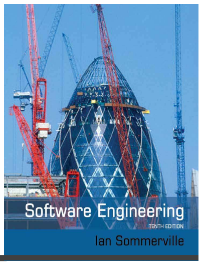
Iterator

- An **`Iterator`** produces proper elements for processing
- Defining an Iterator may be complex
- Using an Iterator must be simple
  - they're all used in the same way
- E.g. **`update()`** all elements of **`List list`**:

```
Iterator it;
for (it=list.listIterator(); it.hasNext(); )
   it.next().update();
```

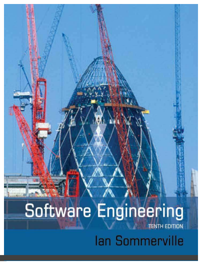
- Makes iteration through elements of a set "higher level"
- Separates the *production* of elements for iteration from the *operation* at each step in the iteration.

Iterator (cont'd)



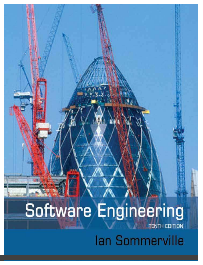
- Iterator is a design pattern that is encountered very often.
  - Problem: Mechanism to operate on every element of a set.
  - Context: The set is represented in some data structure (list, array, hashtable, etc.)
  - Solution: Provide a way to iterate through every element.
- Common Classes using Iterators in Java API
  - StringTokenizer
  - Vector, ArrayList, etc …
  - Even I/O streams work like Iterators

Iterator (in Java)



```
public interface Iterator {
   // Returns true if there are more
   // elements to iterate over; false
   // otherwise
   public boolean hasNext();

   // If there are more elements to
   // iterate over, returns the next one.
   // Modifies the state "this" to record
   // that it has returned the element.
   // If no elements remain, throw
  // NoSuchElementException.
   public Object next()
           throws NoSuchElementException;

   public void remove();
}
```

`Iterator` vs. `Enumeration`

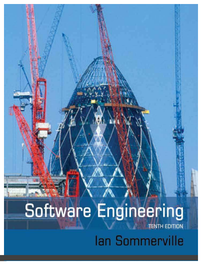


- Java provides another interface **`Enumeration`** for iterating over a collection.
- **`Iterator`** is
  - newer (since JDK 1.2)
  - has shorter method names
  - has a **`remove`**() method to remove elements from a collection during iteration
-

Iterator	Enumeration
`hasNext()`	`hasMoreElements()`
`next()`	`nextElement()`
`remove()`	-

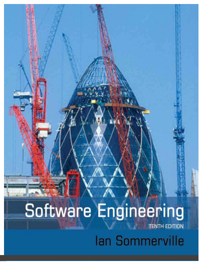
- **`Iterator`** is recommended for new implementations.

Example Loop controlled by `next()`



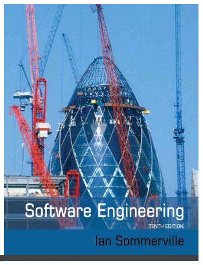
```
private Payroll payroll = new Payroll();
...
public void decreasePayroll() {
   Iterator it = payroll.getIterator();
  while (it.hasNext()) {
       Employee e = (Employee)it.next();
       double salary = e.getSalary();
       e.setSalary(salary*.9);
   }
   for (Employee emp : payroll) {

   }
}
```

Implementing an `Iterator`



```
public class Payroll {
   private Employee[] employees;
   private int num_employees;
   ...
   // An iterator to loop through all Employees
   public Iterator getIterator() {
       return new EmplGen();
   }
   ...
   private class EmplGen implements Iterator {
   // see next slide
   ...
   }
}
```

Implementing an Iterator



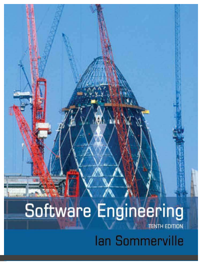
```
private class EmplGen implements Iterator {
   private int n = 0;

   public boolean hasNext() {
       return n < num_employees;
   }

   public Object next() throws NoSuchElementException {
       Object obj;
       if (n < num_employees) {
           obj = employees[n];
           n++;
           return obj;
       }
       else throw new NoSuchElementException
       ("No More Employees");
   }
}
```

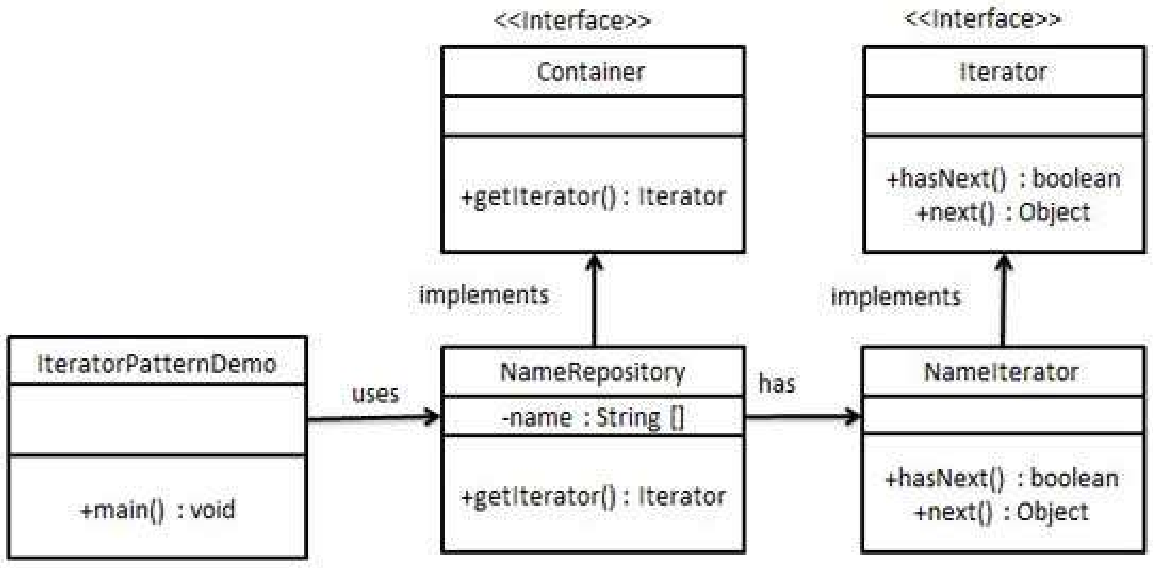
state of iteration captured by index n

returns true if there is an element left to iterate over

returns the next element in the iteration sequence

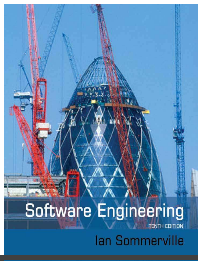
Iterator Design Pattern Example



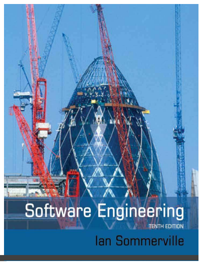


https://www.tutorialspoint.com/design_pattern/iterator_pattern.htm

State Pattern



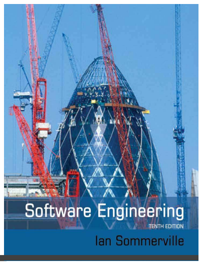
- Dynamically change the representation of an object.
  - also called Data Abstraction
- Users of the object are unaware of the change.
- Example:
  - Implement a set as a **`Vector`** if the number of elements is small
  - Implement a set as a **`Hashtable`** if the number of elements is large
- State pattern is used only on mutable objects.



Example: Set

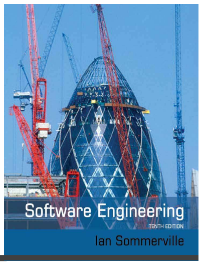
```
public class Set {
   private Object elements;

   public boolean isIn(Object member){
       if (elements instanceof Vector)
           // search using Vector methods
       else
           // search using Hashtable methods
   }

   public void add(Object member){
       if (elements instanceof Vector)
           // add using Vector methods
       else
           // add using Hashtable methods
   }
}
```

Using the state pattern: `SetRep`

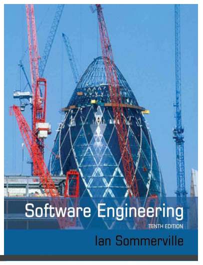


```
public interface SetRep {
   public void add(object member);
   public boolean isIn(Object member);
   public int size();
   public void remove();
}

public class SmallSet implements SetRep {
   private Vector set;
   public void add(Object element) { ... }
   public void remove() { ... }
   ...
}

public class LargeSet implements SetRep {
   private Hashtable set;
   public void add(Object element) { ... }
   public void remove() { ... }
   ...
}
```

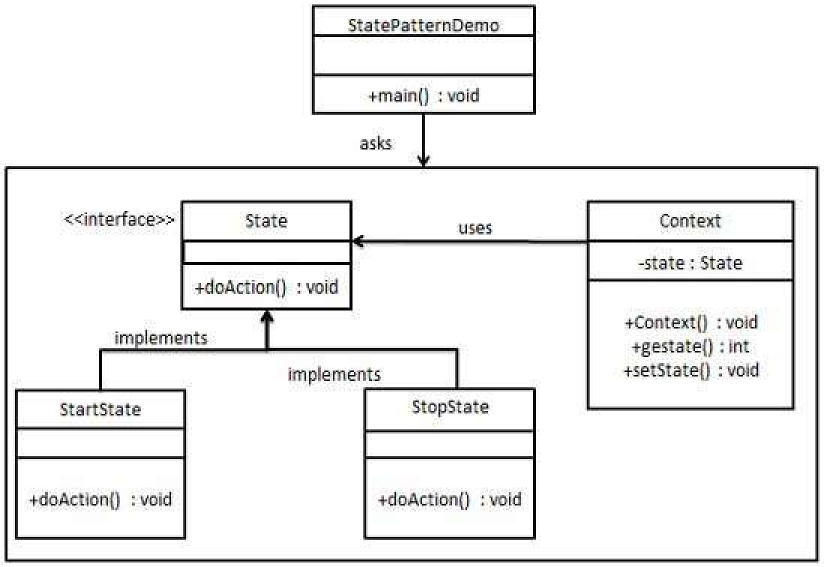
Using the state pattern: a new Set



```
public class Set {
  private SetRep rep;
  private int threshold;

  public void add(Object element) {
     if (rep.size() == threshold)
        rep = new LargeSet(rep.elements());
     rep.add(element);
  }

  public void remove(Object element) {
     rep.remove(elem);
     if (rep.size == threshold)
        rep = new SmallSet(rep.elements());
  }
}
```

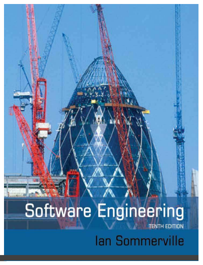


State Pattern Example



https://www.tutorialspoint.com/design_pattern/state_pattern.htm

There are others too



- Chain of Responsibility
- Composite
- Interpreter
- Mediator
- Memento
- Proxy
- Visitor