

Lecture 19

ECE 1145: Software Construction and Evolution

Composite (CH 26)

Observer (CH 28)

Model-View Controller (CH 29)

Template Method (CH 31)

Announcements

- Iteration 7: Blackbox Testing and Pattern Hunting due Nov. 14
 - Bonus: EtaCiv due Dec. 12
- Relevant Exercises: 28.2 29.1
- Midterm survey on Canvas

Questions for Today

How do we combine patterns to build flexible systems?

What is a framework?

You have been building one!

→ HotCiv

MiniDraw: Another example, a 2D GUI framework that you will integrate with HotCiv

More Patterns!

- Composite
- Observer
- Model-View-Controller (MVC)
- Template Method

Composite Pattern: The Problem

Suppose that in a graphics editor we have several graphical objects that need to be moved, resized, etc. as a group.

- Group may be part of a larger group
- Hierarchical, “part-whole” structure

Composite Pattern: The Problem

Folder class

- addFile
- addFolder
- removeFile
- etc.

File class

- delete
- size
- etc.

Composite Pattern: The Problem

Folder class

- addFile
- addFolder
- removeFile
- etc.

Operations are likely to be similar for folders and files

File class

- delete
- size
- etc.

Composite Pattern: The Problem

Folder class

- addFile
- addFolder
- removeFile
- etc.

Operations are likely to be similar for folders and files

```
private static void displaySize(Object item) {  
    if (item instanceof File) {  
        File file = (File) item;  
        System.out.println( "File size is "+file.size() );  
    } else if (item instanceof Folder) {  
        Folder folder = (Folder) item;  
        System.out.println( "Folder size is "+folder.size() );  
    }  
}
```

File class

- delete
- size
- etc.

Composite Pattern: The Problem

Folder class

- addFile
- addFolder
- removeFile
- etc.

File class

- delete
- size
- etc.

Operations are likely to be similar for folders and files

```
private static void displaySize(Object item) {  
    if (item instanceof File) {  
        File file = (File) item;  
        System.out.println( "File size is "+file.size() );  
    } else if (item instanceof Folder) {  
        Folder folder = (Folder) item;  
        System.out.println( "Folder size is "+folder.size() );  
    }  
}
```

Stability and changeability?

Composite Pattern: The Solution

1. Program to an interface

→ Define a common interface for the **part** and the **whole**

Composite Pattern: The Solution

1. Program to an interface

→ Define a common interface for the **part** and the **whole**

→ File, Folder

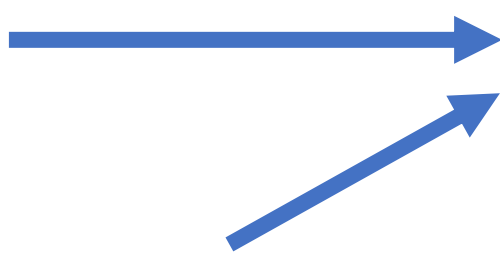
Folder class

- addFile
- addFolder
- removeFile
- etc.

File class

- delete
- size
- etc.

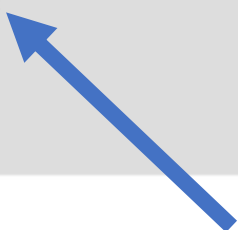
Component interface



```
/** Define the Component interface  
 * (partial for a folder hierarchy) */  
interface Component {  
    public void addComponent(Component child);  
    public int size();  
}
```

Composite Pattern: The Solution

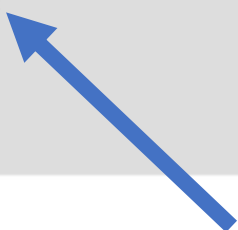
```
/** Define a (partial) folder abstraction */  
class Folder implements Component {  
    private List<Component> components = new ArrayList<Component>();  
    public void addComponent(Component child) {  
        components.add(child);  
    }  
    public int size() {  
        int size = 0;  
        for ( Component c: components ) {  
            size += c.size();  
        }  
        return size;  
    }  
}
```



2. Compose required behavior (recursively)

Composite Pattern: The Solution

```
/** Define a (partial) folder abstraction */  
class Folder implements Component {  
    private List<Component> components = new ArrayList<Component>();  
    public void addComponent(Component child) {  
        components.add(child);  
    }  
    public int size() {  
        int size = 0;  
        for (Component c: components) {  
            size += c.size();  
        }  
        return size;  
    }  
}
```



2. Compose required behavior (recursively)

Composite: Compose objects into tree structures to represent part-whole hierarchies, such that clients treat individual objects and compositions of objects uniformly

Composite Pattern

Problem: Handling part-whole hierarchies of objects

Solution: Define a common interface for composite and atomic components alike; define composites as a set of composites/components

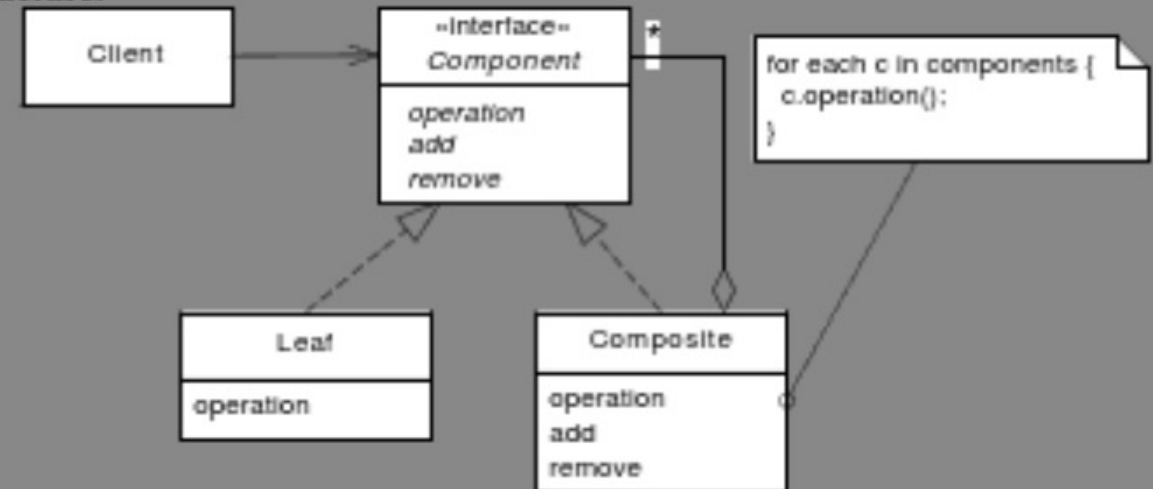
[26.1] Design Pattern: Composite

Intent Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Problem Handling of tree data structures.

Solution Define a common interface for composite and atomic components alike. Define composites in terms of a set of children, each either a composite or atomic component. Define composite behavior in terms of aggregating or composing behavior of each child.

Structure:



Roles **Component** defines a common interface. **Composite** defines a component by means of aggregating other components. **Leaf** defines a primitive, atomic, component i.e. one that has no substructure.

Cost - Benefit It defines a *hierarchy of primitive and composite objects*. It makes the *client interface uniform* as it does not need to know if it is a simple or composite component. It is *easy to add new kinds of components* as they will automatically work with the existing components. A liability is that the *design can become overly general* as it is difficult to constrain the types of leaves a composite may contain. The *interfaces may method bloat* with methods that are irrelevant; for instance an `add` method in a leaf.

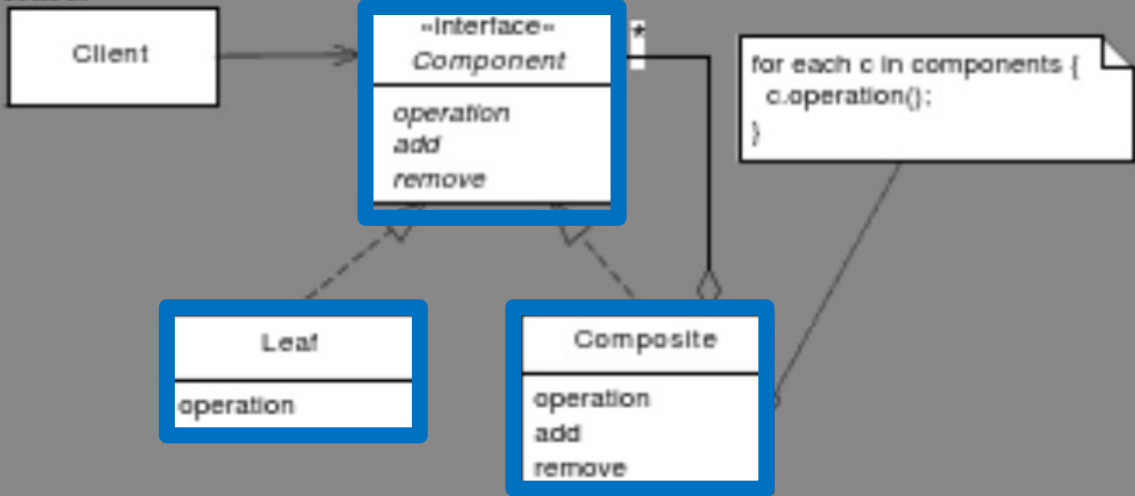
Composite Pattern

Component role: defines the common interface for the part and whole objects

Leaf role: defines primitive part/component

Composite role: aggregates components

[26.1] Design Pattern: Composite

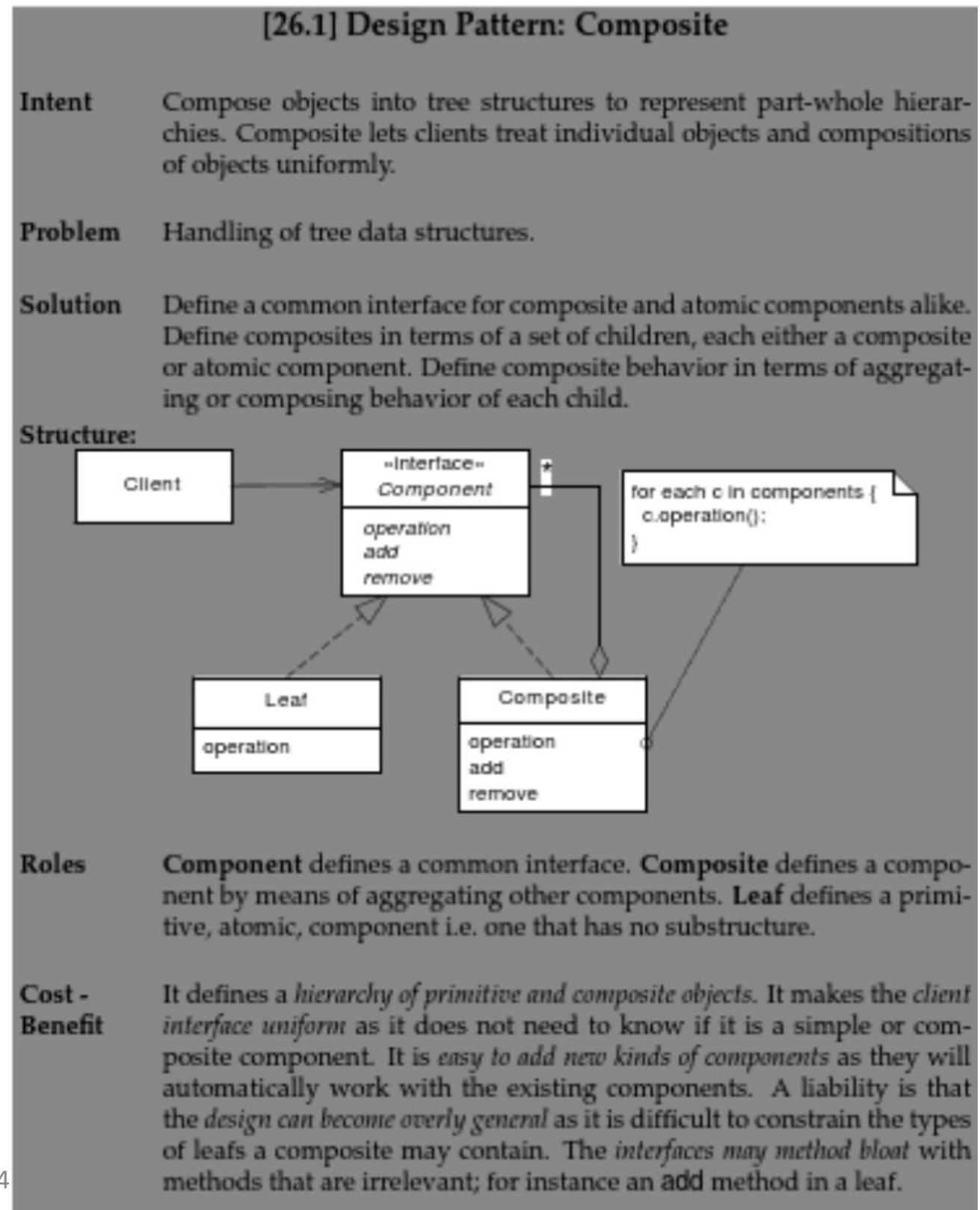
Intent	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Problem	Handling of tree data structures.
Solution	Define a common interface for composite and atomic components alike. Define composites in terms of a set of children, each either a composite or atomic component. Define composite behavior in terms of aggregating or composing behavior of each child.
Structure:	
Roles	Component defines a common interface. Composite defines a component by means of aggregating other components. Leaf defines a primitive, atomic, component i.e. one that has no substructure.
Cost - Benefit	It defines a <i>hierarchy of primitive and composite objects</i> . It makes the <i>client interface uniform</i> as it does not need to know if it is a simple or composite component. It is <i>easy to add new kinds of components</i> as they will automatically work with the existing components. A liability is that the <i>design can become overly general</i> as it is difficult to constrain the types of leaves a composite may contain. The <i>interfaces may method bloat</i> with methods that are irrelevant; for instance an add method in a leaf.

Composite Pattern



Common interface for leaf and composite

- Enables an abstract class defining operations that are identical for both



Composite Pattern



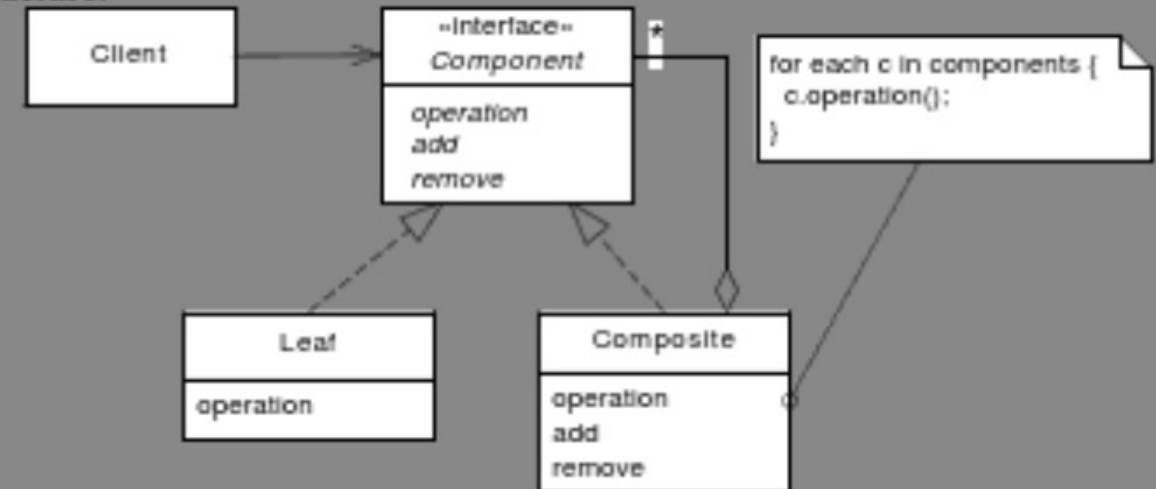
Add and remove methods are meaningless for a leaf object

- Only define them in Composite?

[26.1] Design Pattern: Composite

Intent	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Problem	Handling of tree data structures.
Solution	Define a common interface for composite and atomic components alike. Define composites in terms of a set of children, each either a composite or atomic component. Define composite behavior in terms of aggregating or composing behavior of each child.

Structure:



Roles	Component defines a common interface. Composite defines a component by means of aggregating other components. Leaf defines a primitive, atomic, component i.e. one that has no substructure.
-------	---

Cost - Benefit	It defines a <i>hierarchy of primitive and composite objects</i> . It makes the <i>client interface uniform</i> as it does not need to know if it is a simple or composite component. It is <i>easy to add new kinds of components</i> as they will automatically work with the existing components. A liability is that the <i>design can become overly general</i> as it is difficult to constrain the types of leaves a composite may contain. The <i>interfaces may method bloat</i> with methods that are irrelevant; for instance an <code>add</code> method in a leaf.
----------------	---

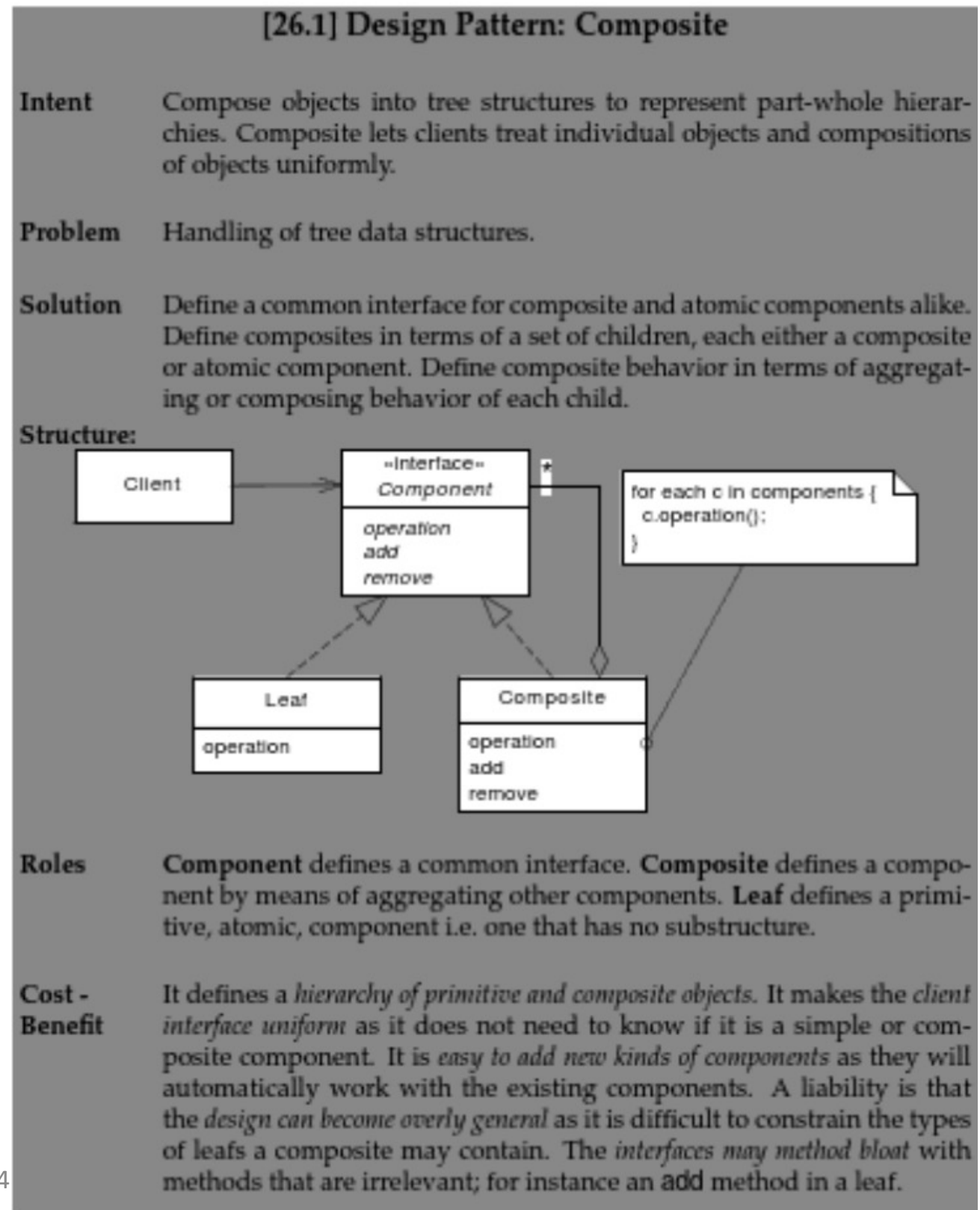
Composite Pattern



Add and remove methods are meaningless for a leaf object

- Only define them in Composite?
- Would reintroduce the initial switch/cast problem

Composite trades **lower cohesion of the leaf** to get a **uniform interface for all objects** of the part-whole structure



Observer: The Problem

Consider a spreadsheet where we want to simultaneously display the same data with a bar chart and a pie chart in their own windows

→ The windows should be synchronized and updated (redrawn) according to changes in the data

Observer: The Problem

Consider a spreadsheet where we want to simultaneously display the same data with a bar chart and a pie chart in their own windows

→ The windows should be synchronized and updated (redrawn) according to changes in the data

Common for displays: they only behave correctly if they are constantly notified of any state changes in the information for display

Observer: The Solution

Recall: Roles and Responsibilities

Roles:

- Object being monitored (data cell in the spreadsheet)
- Dependent object (bar chart window, pie chart window, any other displays)

Observer: The Solution

Recall: Roles and Responsibilities

Roles:

- Object being monitored (data cell in the spreadsheet)

→ Subject

- Dependent object (bar chart window, pie chart window, any other displays)

→ Observer

Observer: The Solution

3. Consider what should be variable

When the subject's state changes, we know the observers have to do some processing, but we don't know what kind of processing.

Observer: The Solution

3. Consider what should be variable

When the subject's state changes, we know the observers have to do some processing, but we don't know what kind of processing.

- Variability is the processing that takes place in response to a change
- Common behavior is notifying the observers that the subject state has changed

Observer: The Solution

1. Program to an interface

Observer interface for the processing responsibility

Observer: The Solution

1. Program to an interface

Observer interface for the processing responsibility

```
/** Observer role in the Observer pattern  
*/  
  
public interface Observer {  
    /** Perform processing appropriate for the changed state.  
    * Subject invokes this method every time its state changes.  
    */  
    public void update();  
}
```

Observer: The Solution

1. Program to an interface

Observer interface for the processing responsibility

```
/** Observer role in the Observer pattern  
*/  
  
public interface Observer {  
    /** Perform processing appropriate for the changed state.  
    * Subject invokes this method every time its state changes.  
    */  
    public void update();  
}
```

Pie chart and bar chart implement Observer, and their redrawing behavior goes in the update method

Observer: The Solution

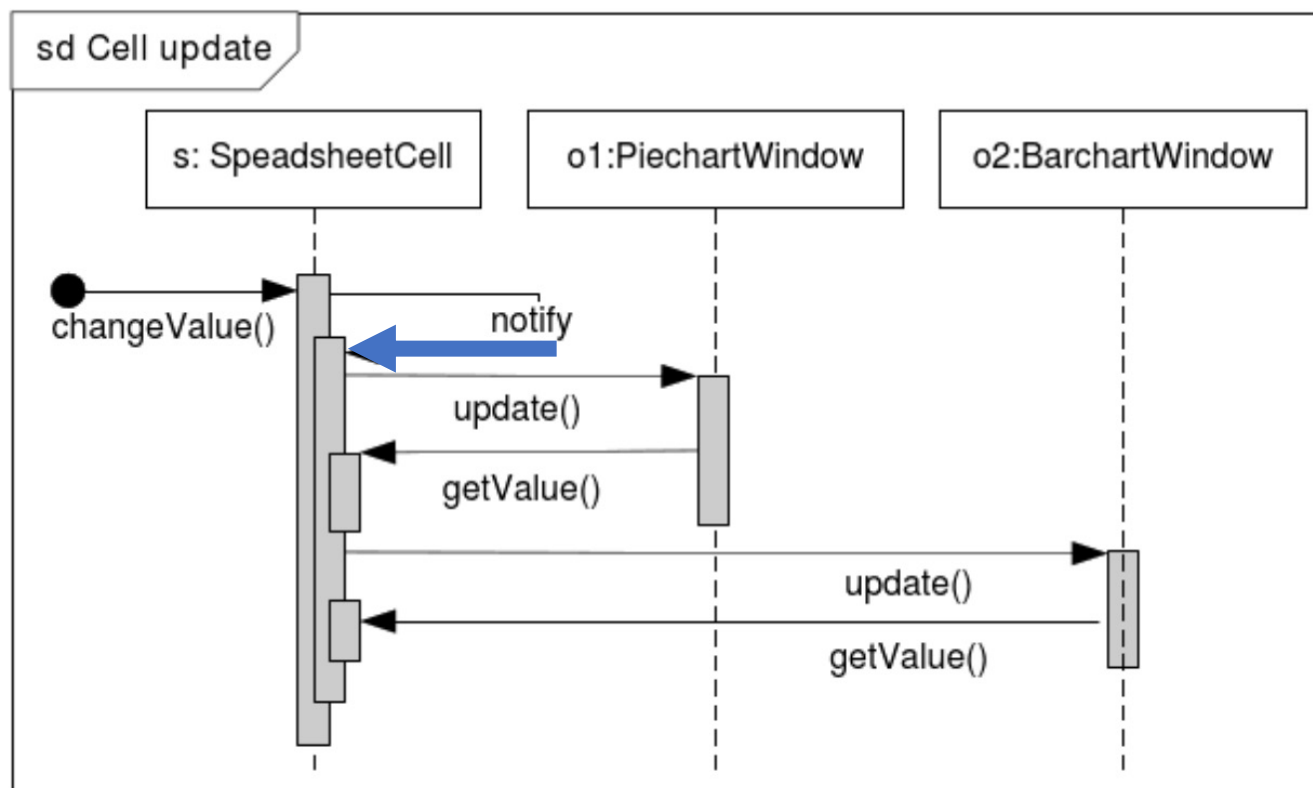
2. Favor object composition

Subject maintains a set of observers and is responsible for invoking the update method of all its observers

Observer: The Solution

2. Favor object composition

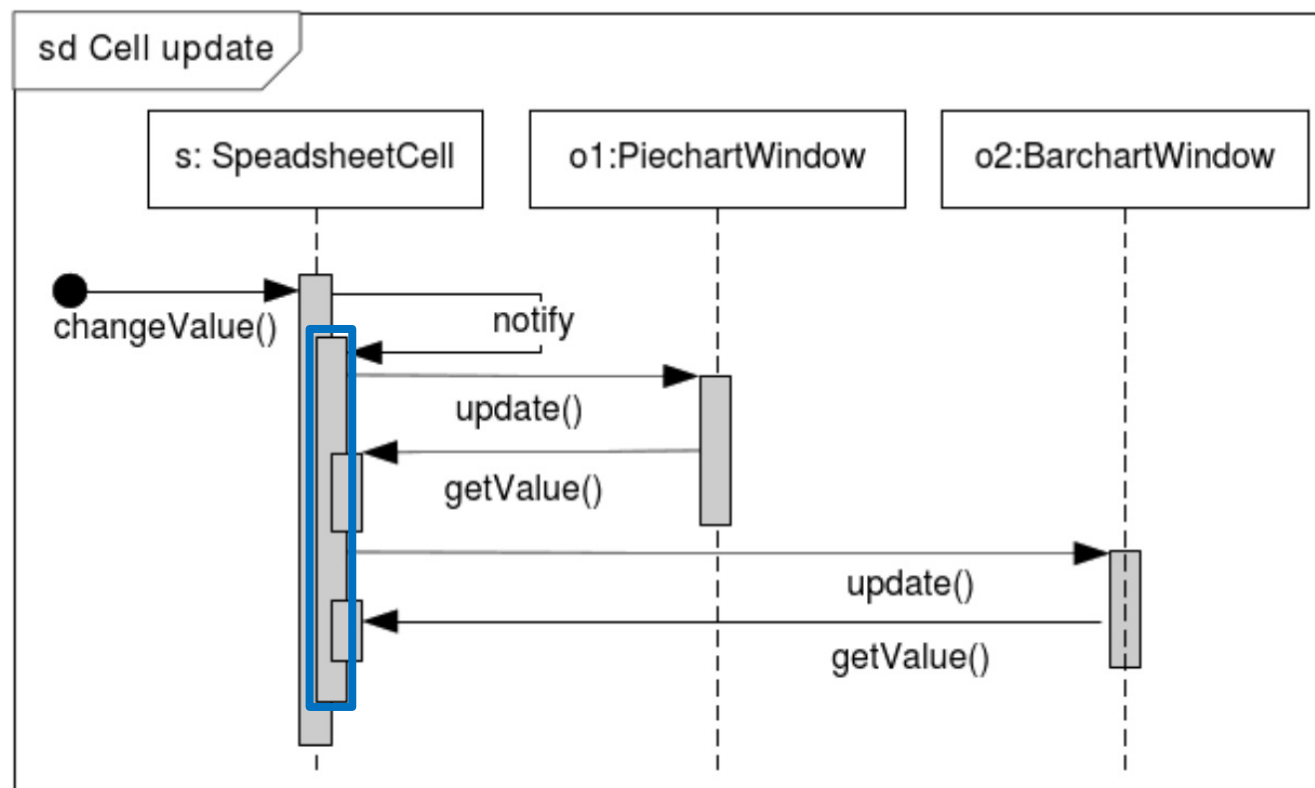
Subject maintains a set of observers and is responsible for invoking the update method of all its observers



Observer: The Solution

2. Favor object composition

Subject maintains a set of observers and is responsible for invoking the update method of all its observers



Can add and remove
observers from the
subject's set

Observer: The Solution

2. Favor object composition

Subject maintains a set of observers and is responsible for invoking the update method of all its observers

```
/** Subject role in the Observer pattern */  
  
public interface Subject {  
    /** add an observer to the set of observers receiving notifications  
     * from this subject.  
     * @param newObserver the observer to add to this subject's set */  
    public void addObserver(Observer newObserver);  
    /** Remove the observer from the set of observers receiving  
     * notifications from this subject.  
     * @param observer the observer to remove from the set.*/  
    public void removeObserver(Observer observer);  
    /** notify all observers in case this subject has changed state. */  
    public void notifyObservers();  
}
```


Observer: The Solution

```
class PieChartWindow implements Observer {
    SpreadsheetCell myCell;
    public PieChartWindow(SpreadsheetCell c){
        myCell = c;
    }
    public void update() {
        System.out.println( "Pie chart notified: value: "+
                             myCell.getValue() );
    }
}
class BarChartWindow implements Observer {
    SpreadsheetCell myCell;
    public BarChartWindow(SpreadsheetCell c){
        myCell = c;
    }
}
```

Observer: The Solution

```
class PieChartWindow implements Observer {
    SpreadsheetCell myCell;
    public PieChartWindow(SpreadsheetCell c){
        myCell = c;
    }
    public void update() {
        System.out.println( "Pie chart notified: value: "+
                             myCell.getValue() );
    }
}
class BarChartWindow implements Observer {
    SpreadsheetCell myCell;
    public BarChartWindow(SpreadsheetCell c){
        myCell = c;
    }
}
```

```
public class DemoObserver {
    public static void main(String[] args) {
        SpreadsheetCell a1 = new SpreadsheetCell();
        Observer
            observer1 = new PieChartWindow(a1),
            observer2 = new BarChartWindow(a1);
        a1.addObserver(observer1);
        a1.addObserver(observer2);

        a1.changeValue(32);
        a1.changeValue(42);

        a1.removeObserver(observer1);
        a1.changeValue(12);
    }
}
```

Observer: The Solution

```
class PieChartWindow implements Observer {
    SpreadsheetCell myCell;
    public PieChartWindow(SpreadsheetCell c){
        myCell = c;
    }
    public void update() {
        System.out.println( "Pie chart notified: value: "+
                             myCell.getValue() );
    }
}
class BarChartWindow implements Observer {
    SpreadsheetCell myCell;
    public BarChartWindow(SpreadsheetCell c){
        myCell = c;
    }
}
```

Register 

```
public class DemoObserver {
    public static void main(String[] args) {
        SpreadsheetCell a1 = new SpreadsheetCell();
        Observer
            observer1 = new PieChartWindow(a1),
            observer2 = new BarChartWindow(a1);
        a1.addObserver(observer1);
        a1.addObserver(observer2);

        a1.changeValue(32);
        a1.changeValue(42);

        a1.removeObserver(observer1);
        a1.changeValue(12);
    }
}
```

Observer: The Solution

```
class PieChartWindow implements Observer {
    SpreadsheetCell myCell;
    public PieChartWindow(SpreadsheetCell c){
        myCell = c;
    }
    public void update() {
        System.out.println( "Pie chart notified: value: "+
                             myCell.getValue() );
    }
}
class BarChartWindow implements Observer {
    SpreadsheetCell myCell;
    public BarChartWindow(SpreadsheetCell c){
        myCell = c;
    }
}
```

Register →

Notify →

```
public class DemoObserver {
    public static void main(String[] args) {
        SpreadsheetCell a1 = new SpreadsheetCell();
        Observer
            observer1 = new PieChartWindow(a1),
            observer2 = new BarChartWindow(a1);
        a1.addObserver(observer1);
        a1.addObserver(observer2);

        a1.changeValue(32);
        a1.changeValue(42);

        a1.removeObserver(observer1);
        a1.changeValue(12);
    }
}
```

Observer: The Solution

```
class PieChartWindow implements Observer {
    SpreadsheetCell myCell;
    public PieChartWindow(SpreadsheetCell c){
        myCell = c;
    }
    public void update() {
        System.out.println( "Pie chart notified: value: "+
                             myCell.getValue() );
    }
}
class BarChartWindow implements Observer {
    SpreadsheetCell myCell;
    public BarChartWindow(SpreadsheetCell c){
        myCell = c;
    }
}
```

Observer: Define a dependency among objects so that when one object changes state, all dependents are notified and updated

```
public class DemoObserver {
    public static void main(String[] args) {
        SpreadsheetCell a1 = new SpreadsheetCell();
        Observer
            observer1 = new PieChartWindow(a1),
            observer2 = new BarChartWindow(a1);
        a1.addObserver(observer1);
        a1.addObserver(observer2);

        a1.changeValue(32);
        a1.changeValue(42);

        a1.removeObserver(observer1);
        a1.changeValue(12);
    }
}
```

Observer Pattern

Problem: A set of objects (Observers) needs to be notified if an object (Subject) changes state

Solution: All observers implement an interface containing an update() method; Subject maintains a list of observers and invokes update()

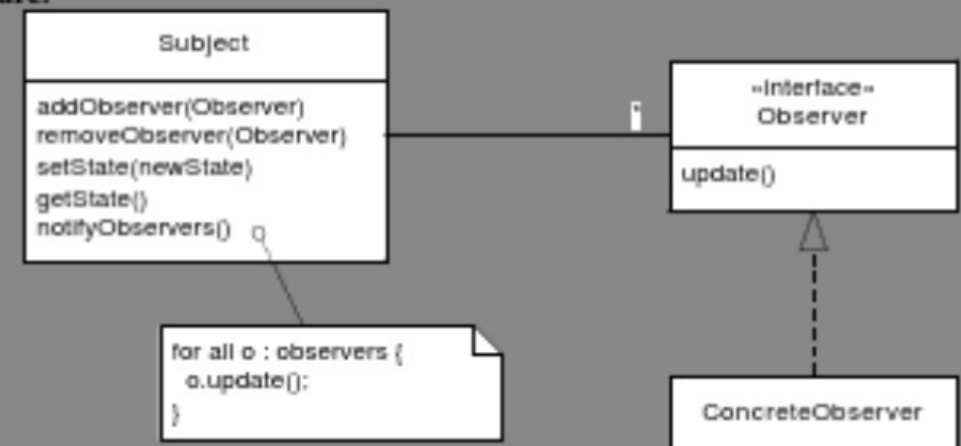
[28.1] Design Pattern: Observer

Intent Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Problem A set of objects needs to be notified in case a common object changes state to ensure system wide consensus and consistency. You want to ensure this consistency in a loosely coupled way.

Solution All objects that must be notified (**Observers**) implements an interface containing an update method. The common object (**Subject**) maintains a list of all observers and when it changes state, it invokes the update method on each object in the list. Thereby all observing objects are notified of state changes.

Structure:



Roles **Observer** specifies the responsibility and interface for being able to be notified. **Subject** is responsible for holding state information, for maintaining a list of all observers, and for invoking the update method on all observers in its list. **ConcreteObserver** defines concrete behavior for how to react when the subject experiences a state change.

Cost - Benefit The benefits are: *Loose coupling between Subject and Observer* thus it is easy to add new observers to the system. *Support broadcast communication* as it is basically publishing information in a one-to-many relation. The liabilities are: *Unexpected/multiple updates*: as the coupling is low it is difficult for observers to infer the nature of subject state changes which may lead to the observers updating too often or spuriously.

Observer Pattern

Subject:

- Handles storage, access, and manipulation of state
- Maintains a set of observers, allows adding/removing observers
- Notifies every observer in the set of any state change by invoking each observers' update() method

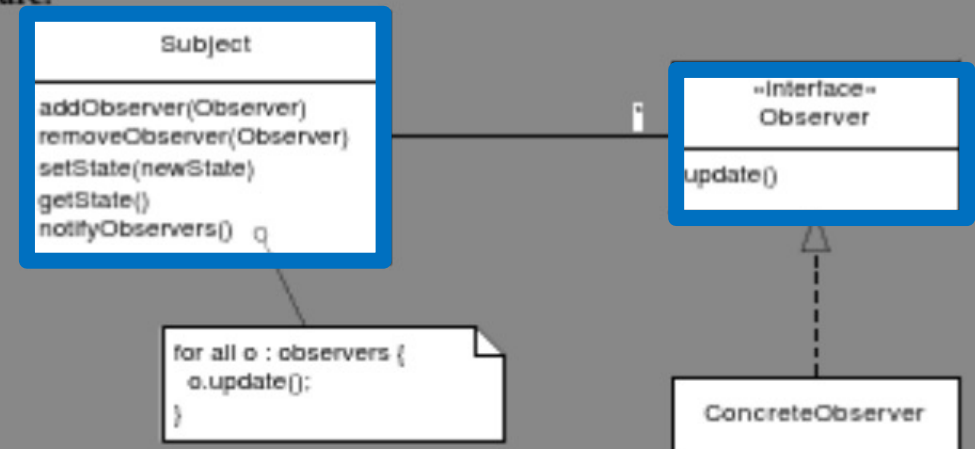
Observer:

- Registers itself with the subject
- Reacts and processes state changes when the update() method is invoked

[28.1] Design Pattern: Observer

Intent	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Problem	A set of objects needs to be notified in case a common object changes state to ensure system wide consensus and consistency. You want to ensure this consistency in a loosely coupled way.
Solution	All objects that must be notified (Observers) implements an interface containing an update method. The common object (Subject) maintains a list of all observers and when it changes state, it invokes the update method on each object in the list. Thereby all observing objects are notified of state changes.

Structure:



Roles	Observer specifies the responsibility and interface for being able to be notified. Subject is responsible for holding state information, for maintaining a list of all observers, and for invoking the update method on all observers in its list. ConcreteObserver defines concrete behavior for how to react when the subject experiences a state change.
-------	---

Cost - Benefit	The benefits are: <i>Loose coupling between Subject and Observer</i> thus it is easy to add new observers to the system. <i>Support broadcast communication</i> as it is basically publishing information in a one-to-many relation. The liabilities are: <i>Unexpected/multiple updates</i> : as the coupling is low it is difficult for observers to infer the nature of subject state changes which may lead to the observers updating too often or spuriously.
----------------	--

Observer Pattern

Recall: **Protocol** – a convention detailing the sequence of interactions expected by a set of roles

[28.1] Design Pattern: Observer

Intent	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Problem	A set of objects needs to be notified in case a common object changes state to ensure system wide consensus and consistency. You want to ensure this consistency in a loosely coupled way.
Solution	All objects that must be notified (Observers) implements an interface containing an <code>update</code> method. The common object (Subject) maintains a list of all observers and when it changes state, it invokes the <code>update</code> method on each object in the list. Thereby all observing objects are notified of state changes.

Structure:



Roles **Observer** specifies the responsibility and interface for being able to be notified. **Subject** is responsible for holding state information, for maintaining a list of all observers, and for invoking the `update` method on all observers in its list. **ConcreteObserver** defines concrete behavior for how to react when the subject experiences a state change.

Cost - Benefit The benefits are: *Loose coupling between Subject and Observer* thus it is easy to add new observers to the system. *Support broadcast communication* as it is basically publishing information in a one-to-many relation. The liabilities are: *Unexpected/multiple updates*: as the coupling is low it is difficult for observers to infer the nature of subject state changes which may lead to the observers updating too often or spuriously.

Observer Pattern

Recall: **Protocol** – a convention detailing the sequence of interactions expected by a set of roles

- **Registration:** observer must be registered with the subject so it will be notified
- **Notification:** when the subject's state is changed, the subject notifies all registered observers by invoking `update()`

[28.1] Design Pattern: Observer

Intent	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Problem	A set of objects needs to be notified in case a common object changes state to ensure system wide consensus and consistency. You want to ensure this consistency in a loosely coupled way.
Solution	All objects that must be notified (Observers) implements an interface containing an <code>update</code> method. The common object (Subject) maintains a list of all observers and when it changes state, it invokes the <code>update</code> method on each object in the list. Thereby all observing objects are notified of state changes.

Structure:



Roles **Observer** specifies the responsibility and interface for being able to be notified. **Subject** is responsible for holding state information, for maintaining a list of all observers, and for invoking the `update` method on all observers in its list. **ConcreteObserver** defines concrete behavior for how to react when the subject experiences a state change.

Cost - Benefit The benefits are: *Loose coupling between Subject and Observer* thus it is easy to add new observers to the system. *Support broadcast communication* as it is basically publishing information in a one-to-many relation. The liabilities are: *Unexpected/multiple updates*: as the coupling is low it is difficult for observers to infer the nature of subject state changes which may lead to the observers updating too often or spuriously.

Observer Pattern

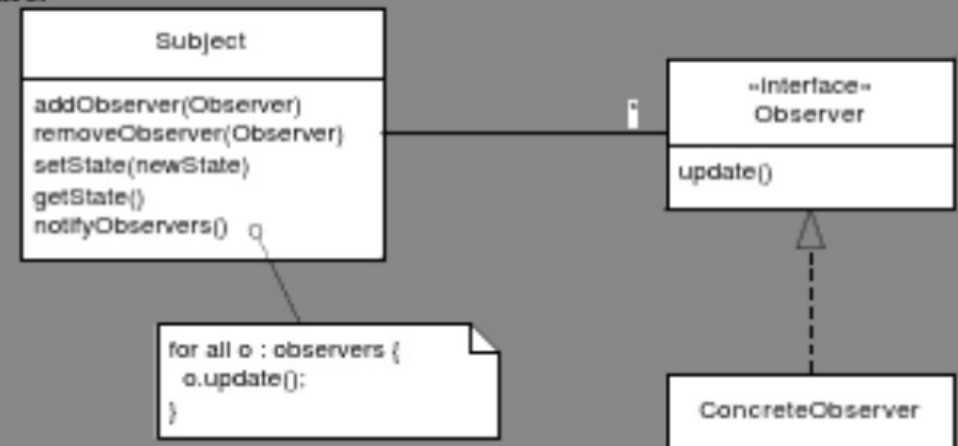
Recall: **Protocol** – a convention detailing the sequence of interactions expected by a set of roles

- **Registration:** observer must be registered with the subject so it will be notified
- **Notification:** when the subject's state is changed, the subject notifies all registered observers by invoking `update()`
- State change can come from any source, but must start the notification protocol
- Observers may ignore state changes

[28.1] Design Pattern: Observer

Intent	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Problem	A set of objects needs to be notified in case a common object changes state to ensure system wide consensus and consistency. You want to ensure this consistency in a loosely coupled way.
Solution	All objects that must be notified (Observers) implements an interface containing an <code>update</code> method. The common object (Subject) maintains a list of all observers and when it changes state, it invokes the <code>update</code> method on each object in the list. Thereby all observing objects are notified of state changes.

Structure:



Roles **Observer** specifies the responsibility and interface for being able to be notified. **Subject** is responsible for holding state information, for maintaining a list of all observers, and for invoking the `update` method on all observers in its list. **ConcreteObserver** defines concrete behavior for how to react when the subject experiences a state change.

Cost - Benefit The benefits are: *Loose coupling between Subject and Observer* thus it is easy to add new observers to the system. *Support broadcast communication* as it is basically publishing information in a one-to-many relation. The liabilities are: *Unexpected/multiple updates*: as the coupling is low it is difficult for observers to infer the nature of subject state changes which may lead to the observers updating too often or spuriously.

Observer Pattern

Pull variant: no information is provided on the type of state change

Push variant: different state changes invoke different update methods

Example: Mouse listener events, press vs. release

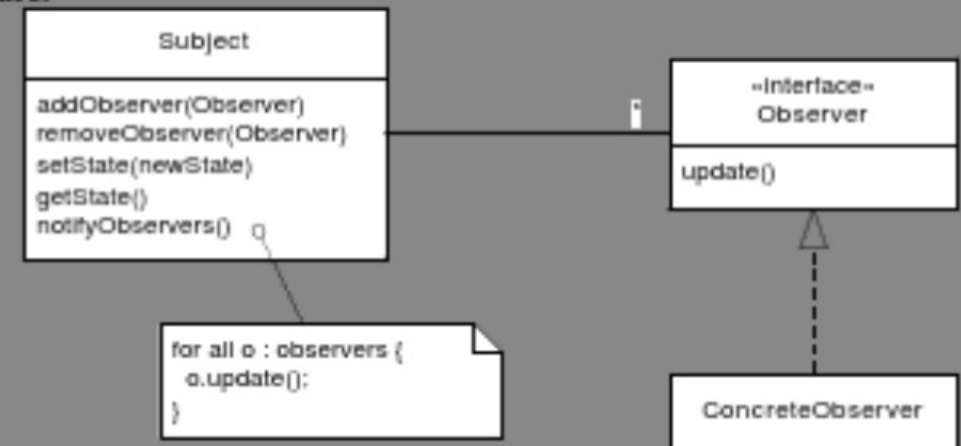
[28.1] Design Pattern: Observer

Intent Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Problem A set of objects needs to be notified in case a common object changes state to ensure system wide consensus and consistency. You want to ensure this consistency in a loosely coupled way.

Solution All objects that must be notified (**Observers**) implements an interface containing an `update` method. The common object (**Subject**) maintains a list of all observers and when it changes state, it invokes the `update` method on each object in the list. Thereby all observing objects are notified of state changes.

Structure:



Roles **Observer** specifies the responsibility and interface for being able to be notified. **Subject** is responsible for holding state information, for maintaining a list of all observers, and for invoking the `update` method on all observers in its list. **ConcreteObserver** defines concrete behavior for how to react when the subject experiences a state change.

Cost - Benefit The benefits are: *Loose coupling between Subject and Observer* thus it is easy to add new observers to the system. *Support broadcast communication* as it is basically publishing information in a one-to-many relation. The liabilities are: *Unexpected/multiple updates*: as the coupling is low it is difficult for observers to infer the nature of subject state changes which may lead to the observers updating too often or spuriously.

Observer Pattern

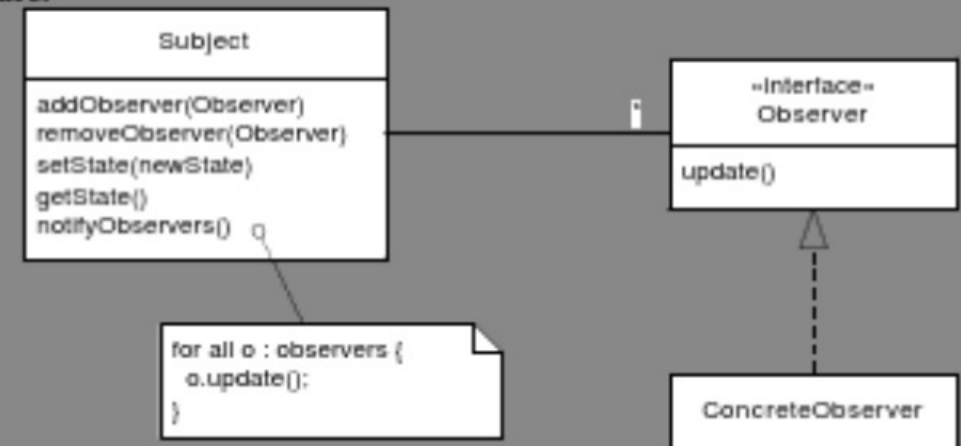


- Loose coupling between subject and observer
- Can have many subjects, many observers

[28.1] Design Pattern: Observer

Intent	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Problem	A set of objects needs to be notified in case a common object changes state to ensure system wide consensus and consistency. You want to ensure this consistency in a loosely coupled way.
Solution	All objects that must be notified (Observers) implements an interface containing an <code>update</code> method. The common object (Subject) maintains a list of all observers and when it changes state, it invokes the <code>update</code> method on each object in the list. Thereby all observing objects are notified of state changes.

Structure:



Roles	Observer specifies the responsibility and interface for being able to be notified. Subject is responsible for holding state information, for maintaining a list of all observers, and for invoking the <code>update</code> method on all observers in its list. ConcreteObserver defines concrete behavior for how to react when the subject experiences a state change.
--------------	---

Cost - Benefit	The benefits are: <i>Loose coupling between Subject and Observer</i> thus it is easy to add new observers to the system. <i>Support broadcast communication</i> as it is basically publishing information in a one-to-many relation. The liabilities are: <i>Unexpected/multiple updates</i> : as the coupling is low it is difficult for observers to infer the nature of subject state changes which may lead to the observers updating too often or spuriously.
-----------------------	--

Observer Pattern

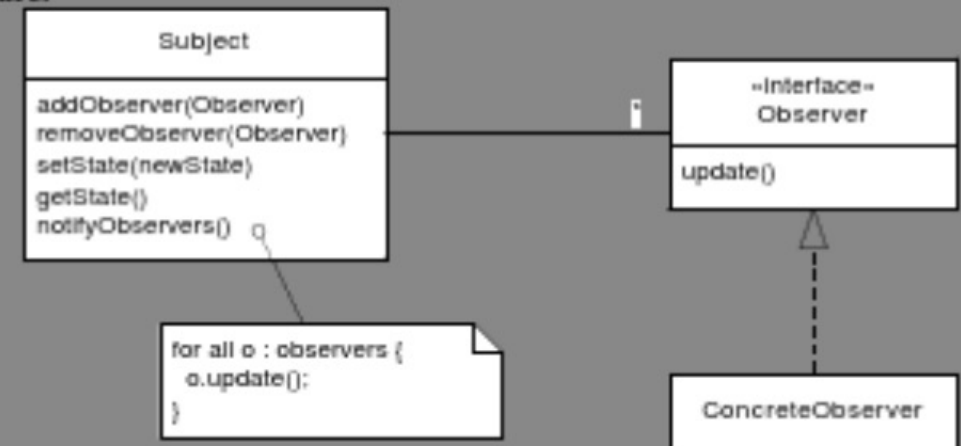


- Every state change will trigger the notification protocol
 “Flickering” when redrawing application graphics
- Risk of circular dependencies

[28.1] Design Pattern: Observer

Intent	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Problem	A set of objects needs to be notified in case a common object changes state to ensure system wide consensus and consistency. You want to ensure this consistency in a loosely coupled way.
Solution	All objects that must be notified (Observers) implements an interface containing an <code>update</code> method. The common object (Subject) maintains a list of all observers and when it changes state, it invokes the <code>update</code> method on each object in the list. Thereby all observing objects are notified of state changes.

Structure:



Roles **Observer** specifies the responsibility and interface for being able to be notified. **Subject** is responsible for holding state information, for maintaining a list of all observers, and for invoking the `update` method on all observers in its list. **ConcreteObserver** defines concrete behavior for how to react when the subject experiences a state change.

Cost - Benefit The benefits are: *Loose coupling between Subject and Observer* thus it is easy to add new observers to the system. *Support broadcast communication* as it is basically publishing information in a one-to-many relation. The liabilities are: *Unexpected/multiple updates*: as the coupling is low it is difficult for observers to infer the nature of subject state changes which may lead to the observers updating too often or spuriously.

Model-View-Controller: The Problem

Vector graphics editors (e.g., Inkscape, Visio) present a palette of shapes that a user can add to a drawing and then manipulate with the mouse. The drawing can be viewed in multiple windows simultaneously showing different parts at different scales.

Model-View-Controller: The Problem

Vector graphics editors (e.g., Inkscape, Visio) present a palette of shapes that a user can add to a drawing and then manipulate with the mouse. The drawing can be viewed in multiple windows simultaneously showing different parts at different scales.

- The drawing needs to be rendered in the windows simultaneously
- The drawing needs to process user inputs from multiple sources (windows, keyboard, mouse, etc.)

Model-View-Controller: The Problem

Vector graphics editors (e.g., Inkscape, Visio) present a palette of shapes that a user can add to a drawing and then manipulate with the mouse. The drawing can be viewed in multiple windows simultaneously showing different parts at different scales.

- The drawing needs to be rendered in the windows simultaneously
- The drawing needs to process user inputs from multiple sources (windows, keyboard, mouse, etc.)

The application must be structured with **low coupling** between domain objects and graphical views.

Model-View-Controller: The Solution

The drawing needs to be rendered in the windows simultaneously

Model-View-Controller: The Solution

The drawing needs to be rendered in the windows simultaneously

→ **Observer** pattern: update windows when state changes

Model-View-Controller: The Solution

The drawing needs to be rendered in the windows simultaneously

→ **Observer** pattern: update windows when state changes

The drawing needs to process user inputs from multiple sources (windows, keyboard, mouse, etc.)

Model-View-Controller: The Solution

The drawing needs to be rendered in the windows simultaneously

→ **Observer** pattern: update windows when state changes

The drawing needs to process user inputs from multiple sources (windows, keyboard, mouse, etc.)

→ Response to mouse click input should change depending on the type of object that was clicked

Model-View-Controller: The Solution

The drawing needs to be rendered in the windows simultaneously

→ **Observer** pattern: update windows when state changes

The drawing needs to process user inputs from multiple sources (windows, keyboard, mouse, etc.)

→ Response to mouse click input should change depending on the type of object that was clicked

→ **State** pattern: alter behavior when the state changes

Model-View-Controller: The Solution

Observer Role	MVC Role
Subject	
Observer	

Model-View-Controller: The Solution

Observer Role	MVC Role
Subject	Model
Observer	View

Model-View-Controller: The Solution

Observer Role
Subject
Observer

MVC Role
Model
View

State Role
Context
State

Model-View-Controller: The Solution

Observer Role
Subject
Observer

MVC Role
Model
View
Controller (controls model state)

State Role
Context
State

Model-View-Controller: The Solution

Model

- Store application state.
- Maintain the set of Views associated.
- Notify all views in case of state changes.

View

- Visualize model state graphically.
- Accept user input events, delegate them to the associated Controller.
- Manage a set of controllers and allow the user to set which controller is active.

Controller

- Interpret user input events and translate them into state changes in the Model.

Model-View-Controller: The Solution

Model

- Store application state.
- Maintain the set of Views associated.
- Notify all views in case of state changes.

View

- Visualize model state graphically.
- Accept user input events, delegate them to the associated Controller.
- Manage a set of controllers and allow the user to set which controller is active.

Controller

- Interpret user input events and translate them into state changes in the Model.

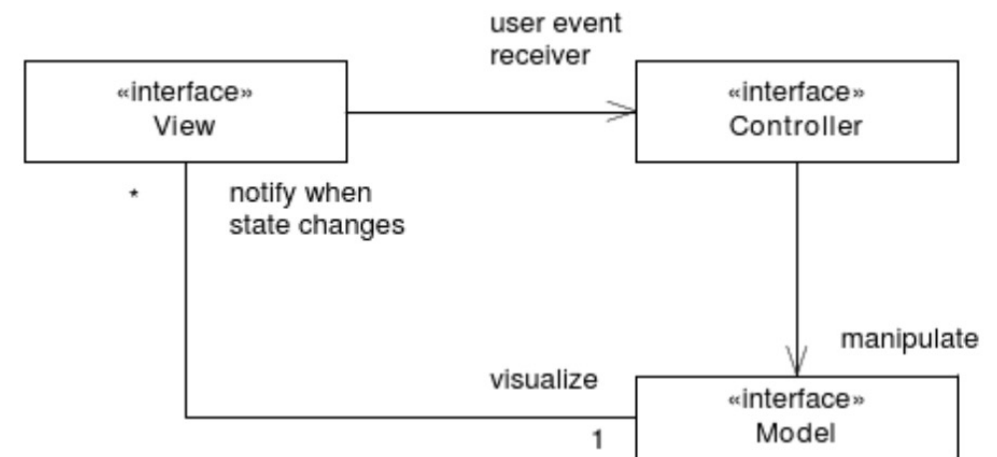


Figure 29.2: MVC role structure.

Model-View-Controller: The Solution

Model

- Store application state.
- Maintain the set of Views associated.
- Notify all views in case of state changes.

View

- Visualize model state graphically.
- Accept user input events, delegate them to the associated Controller.
- Manage a set of controllers and allow the user to set which controller is active.

Controller

- Interpret user input events and translate them into state changes in the Model.

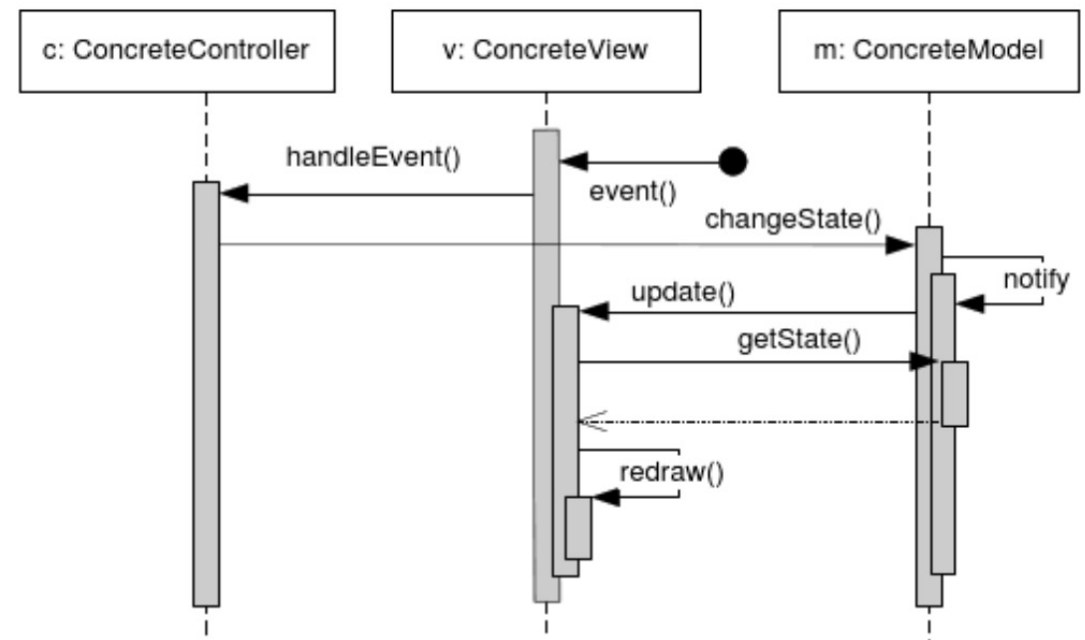


Figure 29.3: MVC protocol.

Views register in the model to receive update events
View must be associated with the proper controllers

Model-View-Controller: The Solution

Model

- Store application state.
- Maintain the set of Views associated.
- Notify all views in case of state changes.

View

- Visualize model state graphically.
- Accept user input events, delegate them to the associated Controller.
- Manage a set of controllers and allow the user to set which controller is active.

Controller

- Interpret user input events and translate them into state changes in the Model.

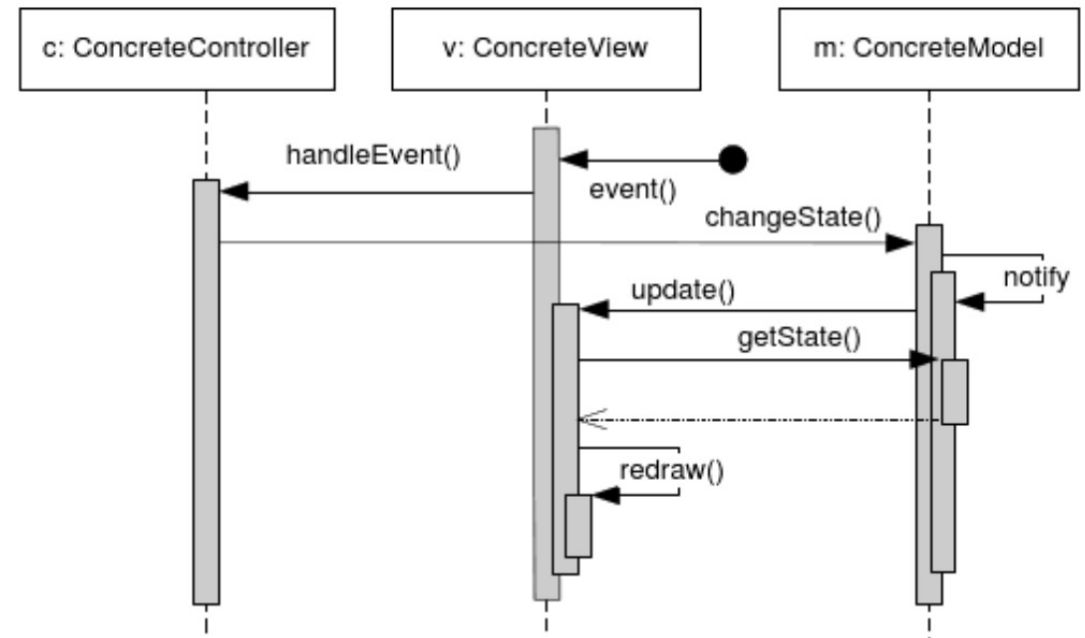


Figure 29.3: MVC protocol.

Model-View-Controller: define a loosely coupled design to form the architecture of graphical user interfaces with multiple windows and user input from multiple input sources.

Model-View-Controller Pattern

Problem: A GUI must support multiple windows rendering representations of an underlying set of objects; the user must be able to manipulate the objects using mouse/keyboard

Solution: A Model contains the application's state and notifies all Views when state changes; Views receive user inputs and forward to the associated Controllers, which make calls to the Model

[29.1] Design Pattern: Model-View-Controller

Intent	Define a loosely coupled design to form the architecture of graphical user interfaces having multiple windows and handling user input from mouse, keyboard, or other input sources.
Problem	A graphical user interface must support multiple windows rendering different visual representations of an underlying set of state-full objects in a consistent way. The user must be able to manipulate the objects' state using mouse and keyboard.
Solution	A Model contains the application's state and notifies all Views when state changes happen. The Views are responsible for rendering the model when notified. User input events are received by a View but forwarded to its associated Controller . The Controller interprets events and makes the appropriate calls to the Model .

Structure:



Roles **Model** maintains application state and updates all associated **Views**. **View** renders the model graphically and delegates user events to the **Controller** that in turn is responsible for modifying the model.

Cost - Benefit The benefits are: *loose coupling between all three roles* meaning you can add new graphical renderings or user event processing. *Multiple views/windows* are supported. It is possible to *change event processing at run-time*. The liabilities are: *unexpected/multiple updates*: as the coupling is low it is difficult for views to infer the nature of model state changes which may lead to graphical flickering. *Design complexity* is another concern if the development team is untrained.

Model-View-Controller Pattern

Similar pros/cons as observer and state:



- Observer: user can open many windows and all are updated to reflect changes in the model
- Loose coupling
 - Each window can render the model differently as necessary
 - User event delegation to Controllers means model changes are not coupled to a window object

Example: Choosing a shape drawing tool changes the controller associated with the window, can be changed at runtime

[29.1] Design Pattern: Model-View-Controller

Intent	Define a loosely coupled design to form the architecture of graphical user interfaces having multiple windows and handling user input from mouse, keyboard, or other input sources.
Problem	A graphical user interface must support multiple windows rendering different visual representations of an underlying set of state-full objects in a consistent way. The user must be able to manipulate the objects' state using mouse and keyboard.
Solution	A Model contains the application's state and notifies all Views when state changes happen. The Views are responsible for rendering the model when notified. User input events are received by a View but forwarded to its associated Controller . The Controller interprets events and makes the appropriate calls to the Model .

Structure:



Roles **Model** maintains application state and updates all associated **Views**. **View** renders the model graphically and delegates user events to the **Controller** that in turn is responsible for modifying the model.

Cost - Benefit The benefits are: *loose coupling between all three roles* meaning you can add new graphical renderings or user event processing. *Multiple views/windows* are supported. It is possible to *change event processing at run-time*. The liabilities are: *unexpected/multiple updates*: as the coupling is low it is difficult for views to infer the nature of model state changes which may lead to graphical flickering. *Design complexity* is another concern if the development team is untrained.

Model-View-Controller Pattern

Similar pros/cons as observer and state:



- Structure and protocol are complex, must be programmed with care
- Multiple updates, circular update risk similar to Observer

[29.1] Design Pattern: Model-View-Controller

Intent	Define a loosely coupled design to form the architecture of graphical user interfaces having multiple windows and handling user input from mouse, keyboard, or other input sources.
Problem	A graphical user interface must support multiple windows rendering different visual representations of an underlying set of state-full objects in a consistent way. The user must be able to manipulate the objects' state using mouse and keyboard.
Solution	A Model contains the application's state and notifies all Views when state changes happen. The Views are responsible for rendering the model when notified. User input events are received by a View but forwarded to its associated Controller . The Controller interprets events and makes the appropriate calls to the Model .

Structure:



Roles **Model** maintains application state and updates all associated **Views**. **View** renders the model graphically and delegates user events to the **Controller** that in turn is responsible for modifying the model.

Cost - Benefit The benefits are: *loose coupling between all three roles* meaning you can add new graphical renderings or user event processing. *Multiple views/windows* are supported. It is possible to *change event processing at run-time*. The liabilities are: *unexpected/multiple updates*: as the coupling is low it is difficult for views to infer the nature of model state changes which may lead to graphical flickering. *Design complexity* is another concern if the development team is untrained.

Model-View-Controller Pattern

MVC is an **architectural** pattern

- **Architectural patterns** focus on a particular architectural problem in a particular domain (e.g., graphical user interfaces)

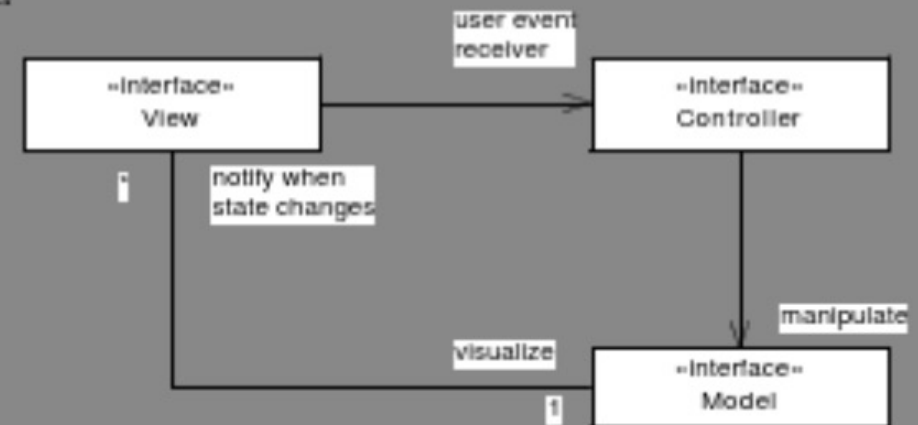
[29.1] Design Pattern: Model-View-Controller

Intent Define a loosely coupled design to form the architecture of graphical user interfaces having multiple windows and handling user input from mouse, keyboard, or other input sources.

Problem A graphical user interface must support multiple windows rendering different visual representations of an underlying set of state-full objects in a consistent way. The user must be able to manipulate the objects' state using mouse and keyboard.

Solution A **Model** contains the application's state and notifies all **Views** when state changes happen. The **Views** are responsible for rendering the model when notified. User input events are received by a **View** but forwarded to its associated **Controller**. The **Controller** interprets events and makes the appropriate calls to the **Model**.

Structure:



Roles **Model** maintains application state and updates all associated **Views**. **View** renders the model graphically and delegates user events to the **Controller** that in turn is responsible for modifying the model.

Cost - Benefit The benefits are: *loose coupling between all three roles* meaning you can add new graphical renderings or user event processing. *Multiple views/windows* are supported. It is possible to *change event processing at run-time*. The liabilities are: *unexpected/multiple updates*: as the coupling is low it is difficult for views to infer the nature of model state changes which may lead to graphical flickering. *Design complexity* is another concern if the development team is untrained.

Model-View-Controller Pattern

MVC is an **architectural** pattern

- **Architectural patterns** focus on a particular architectural problem in a particular domain (e.g., graphical user interfaces)

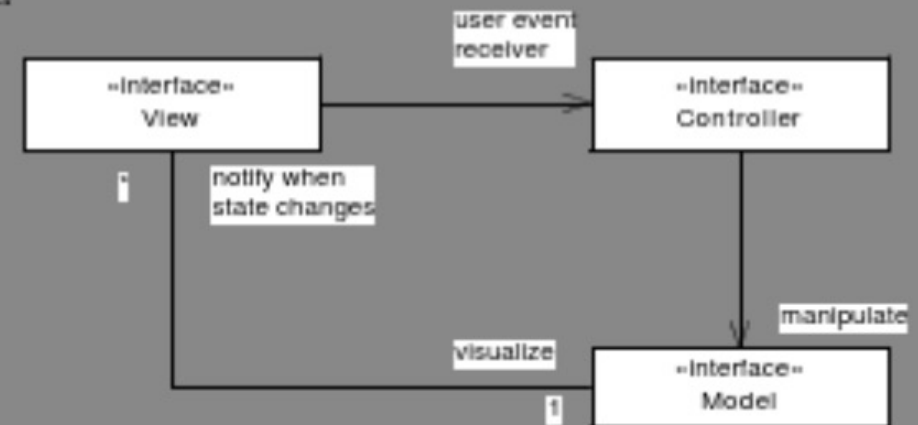
Model may consist of many objects with different interfaces

- If a single interface is used to manipulate all aspects of the model, this is **Façade**

[29.1] Design Pattern: Model-View-Controller

Intent	Define a loosely coupled design to form the architecture of graphical user interfaces having multiple windows and handling user input from mouse, keyboard, or other input sources.
Problem	A graphical user interface must support multiple windows rendering different visual representations of an underlying set of state-full objects in a consistent way. The user must be able to manipulate the objects' state using mouse and keyboard.
Solution	A Model contains the application's state and notifies all Views when state changes happen. The Views are responsible for rendering the model when notified. User input events are received by a View but forwarded to its associated Controller . The Controller interprets events and makes the appropriate calls to the Model .

Structure:



Roles	Model maintains application state and updates all associated Views . View renders the model graphically and delegates user events to the Controller that in turn is responsible for modifying the model.
-------	--

Cost - Benefit	The benefits are: <i>loose coupling between all three roles</i> meaning you can add new graphical renderings or user event processing. <i>Multiple views/windows</i> are supported. It is possible to <i>change event processing at run-time</i> . The liabilities are: <i>unexpected/multiple updates</i> : as the coupling is low it is difficult for views to infer the nature of model state changes which may lead to graphical flickering. <i>Design complexity</i> is another concern if the development team is untrained.
----------------	--

Template Method: The Problem

Consider the pay station algorithm to receive payment:

1. Check that input is a valid coin
2. Add coin value to payment sum
3. Calculate minutes of parking from payment sum

Template Method: The Problem

Consider the pay station algorithm to receive payment:

1. Check that input is a valid coin
2. Add coin value to payment sum
3. Calculate minutes of parking from payment sum

There is a strict **sequence** of steps required for the algorithm to be correct.

→ What if we want to change behavior of the individual steps without changing the overall algorithm sequence?

→ Sounds like a job for Strategy! (e.g., variability in step 3)

Template Method: The Solution

Template Method: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses or delegates, let behavior of certain steps of an algorithm be varied without changing the algorithm structure.

Template Method: The Solution

Template Method: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses or delegates, let behavior of certain steps of an algorithm be varied without changing the algorithm structure.

→ Can be polymorphic or compositional!

- Define the structure of the algorithm
- Define the fixed steps
- Call methods that encapsulate behavior that varies

Template Method: The Solution

Template Method: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses or delegates, let behavior of certain steps of an algorithm be varied without changing the algorithm structure.

→ Can be polymorphic or compositional!

- Define the structure of the algorithm
 - Define the fixed steps
 - Call methods that encapsulate behavior that varies
- } **Template method**
- Hook methods**

Template Method: The Solution

Polymorphic

```
abstract class AbstractClass {
    public void templateMethod() {
        [fixed code part 1]
        step1();
        [fixed code part 2]
        step2();
        [fixed code part 3]
    }
    protected abstract void step1();
    protected abstract void step2();
}
class ConcreteClass extends AbstractClass() {
    protected void step1() {
        [step 1 specific behavior]
    }
    protected void step2() {
        [step 2 specific behavior]
    }
}
```

Compositional

```
class Class {
    private HookInterface1 hook1;
    private HookInterface2 hook2;
    public void setHook( HookInterface1 hook1,
                        HookInterface2 hook2) {
        this.hook1 = hook1;
        this.hook2 = hook2;
    }
    public void templateMethod() {
        [fixed code part 1]
        hook1.step1();
        [fixed code part 2]
        hook2.step2();
        [fixed code part 3]
    }
}
interface HookInterface1 {
    public void step1();
}
interface HookInterface2 {
    public void step2();
}
class ConcreteHook1 implements HookInterface1() {
    public void step1() {
        [step 1 specific behavior]
    }
}
class ConcreteHook2 implements HookInterface2() {
    public void step2() {
        [step 2 specific behavior]
    }
}
```


Template Method: The Solution

Unification Variant

```
abstract class AbstractClass {
    public void templateMethod() {
        [fixed code part 1]
        step1();
        [fixed code part 2]
        step2();
        [fixed code part 3]
    }
    protected abstract void step1();
    protected abstract void step2();
}
class ConcreteClass extends AbstractClass() {
    protected void step1() {
        [step 1 specific behavior]
    }
    protected void step2() {
        [step 2 specific behavior]
    }
}
```

Separation Variant

```
class Class {
    private HookInterface1 hook1;
    private HookInterface2 hook2;
    public void setHook( HookInterface1 hook1,
                        HookInterface2 hook2) {
        this.hook1 = hook1;
        this.hook2 = hook2;
    }
    public void templateMethod() {
        [fixed code part 1]
        hook1.step1();
        [fixed code part 2]
        hook2.step2();
        [fixed code part 3]
    }
}
interface HookInterface1 {
    public void step1();
}
interface HookInterface2 {
    public void step2();
}
class ConcreteHook1 implements HookInterface1() {
    public void step1() {
        [step 1 specific behavior]
    }
}
class ConcreteHook2 implements HookInterface2() {
    public void step2() {
        [step 2 specific behavior]
    }
}
```

Template Method: The Solution

Unification Variant

```
abstract class AbstractClass {
    public void templateMethod() {
        [fixed code part 1]
        step1();
        [fixed code part 2]
        step2();
        [fixed code part 3]
    }
    protected abstract void step1();
    protected abstract void step2();
}
class ConcreteClass extends AbstractClass() {
    protected void step1() {
        [step 1 specific behavior]
    }
    protected void step2() {
        [step 2 specific behavior]
    }
}
```

The Template Method pattern is focused on method abstractions

Separation Variant

```
class Class {
    private HookInterface1 hook1;
    private HookInterface2 hook2;
    public void setHook( HookInterface1 hook1,
                        HookInterface2 hook2) {
        this.hook1 = hook1;
        this.hook2 = hook2;
    }
    public void templateMethod() {
        [fixed code part 1]
        hook1.step1();
        [fixed code part 2]
        hook2.step2();
        [fixed code part 3]
    }
}
interface HookInterface1 {
    public void step1();
}
interface HookInterface2 {
    public void step2();
}
class ConcreteHook1 implements HookInterface1() {
    public void step1() {
        [step 1 specific behavior]
    }
}
class ConcreteHook2 implements HookInterface2() {
    public void step2() {
        [step 2 specific behavior]
    }
}
```

Template Method Pattern

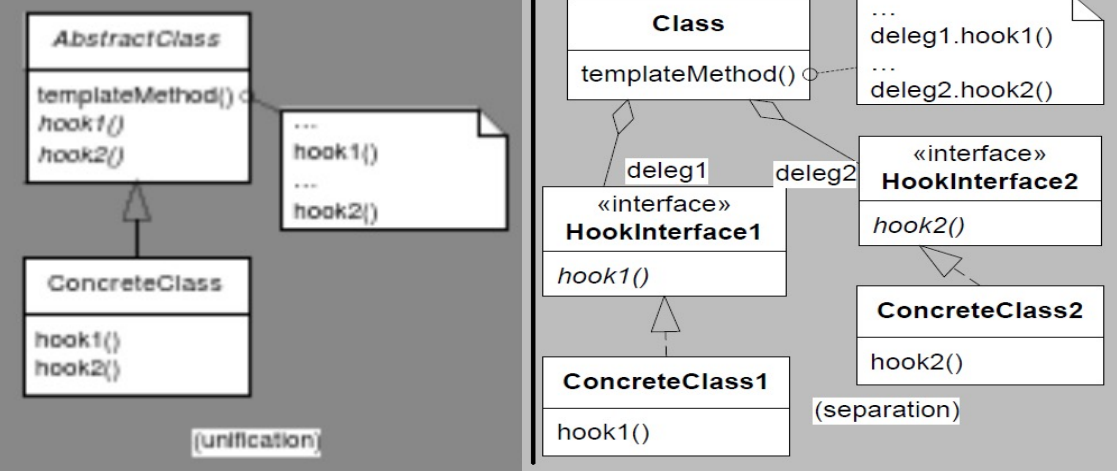
Problem: Need to have different behaviors of some steps of an algorithm, but algorithm's overall structure is fixed

Solution: Define the algorithm structure in a template method, let it call hook methods that encapsulate steps with variable behavior

[31.1] Design Pattern: Template Method

Intent	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses or delegates. Template Method lets the behavior of certain steps of an algorithm be varied without changing the algorithm's structure.
Problem	There is a need to have different behaviors of some steps of an algorithm but the algorithm's structure is otherwise fixed.
Solution	Define the algorithm's structure and invariant behavior in a template method and let it call hook methods that encapsulate the steps with variable behavior. Hook methods may either be abstract methods in the same class as the template method, or they may be called on delegate object(s) implementing one or several interfaces defining the hook methods. The former variant is the <i>unification</i> variant, the latter the <i>separation</i> variant.

Structure:



Roles	The roles are method abstractions: the template method defines the algorithm structure and invariant behavior. Hook methods encapsulate variable behavior. The HookInterface interface defines the method signatures of the hook methods.
Cost - Benefit	The benefits are that the <i>algorithm template is reused</i> and thus avoids multiple maintenance; that the <i>behavior of individual steps, the hooks, may be changed</i> ; and that <i>groups of hook methods may be varied independently</i> (separation variant only). The liability is added complexity of the algorithm as steps have to be encapsulated.

Template Method Pattern



- Avoids multiple maintenance
- Hook behavior can be easily changed
- Hook methods can be varied independently

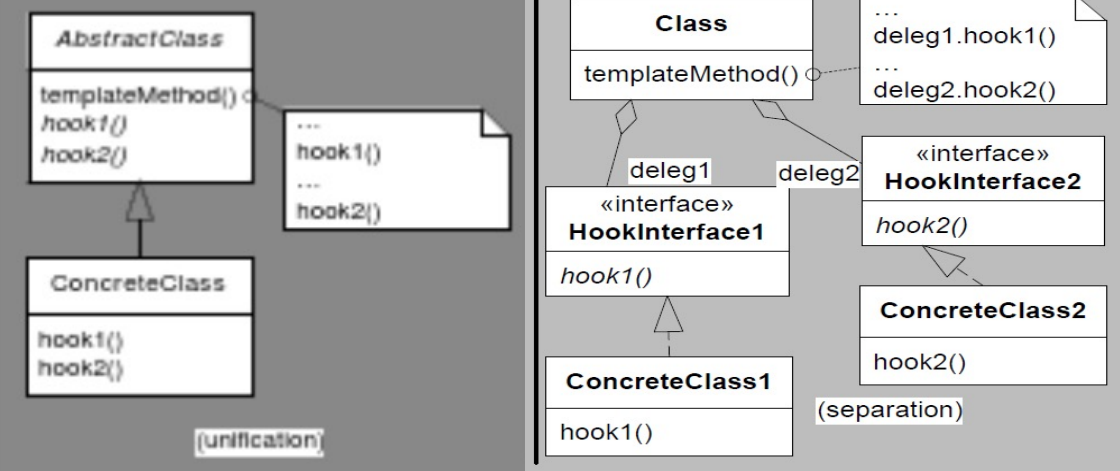


- Added complexity
- Encapsulation of steps

[31.1] Design Pattern: Template Method

Intent	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses or delegates. Template Method lets the behavior of certain steps of an algorithm be varied without changing the algorithm's structure.
Problem	There is a need to have different behaviors of some steps of an algorithm but the algorithm's structure is otherwise fixed.
Solution	Define the algorithm's structure and invariant behavior in a template method and let it call hook methods that encapsulate the steps with variable behavior. Hook methods may either be abstract methods in the same class as the template method, or they may be called on delegate object(s) implementing one or several interfaces defining the hook methods. The former variant is the <i>unification</i> variant, the latter the <i>separation</i> variant.

Structure:



Roles	The roles are method abstractions: the template method defines the algorithm structure and invariant behavior. Hook methods encapsulate variable behavior. The HookInterface interface defines the method signatures of the hook methods.
Cost - Benefit	The benefits are that the <i>algorithm template is reused</i> and thus avoids multiple maintenance; that the <i>behavior of individual steps, the hooks, may be changed</i> ; and that <i>groups of hook methods may be varied independently</i> (separation variant only). The liability is <i>added complexity of the algorithm</i> as steps have to be encapsulated.

Summary

- Composite: similar handling of all objects in a part-whole hierarchy
- Observer: notify and update dependents according to state changes
- Model-View-Controller: loosely coupled architecture for GUIs
- Template Method: algorithm structure with variability in steps

Next time: Frameworks, MiniDraw GUI,
putting it all together