Observer / Template Methods
The I.T. systems for the Olympics are complex and represent a software architecture challenge. Information about the events, such as the detailed scheduling, competitors and results are all stored on a centralized system. This information must then distributed to various subsystems (views), each used by a different category of people.
# Information Distribution

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Participant Profiles</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organizers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Judges</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Athletes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spectators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Press</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scheduling</td>
<td>Participant Profiles</td>
</tr>
<tr>
<td>------------------</td>
<td>------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Organizers</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>Judges</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>Athletes</td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td>Spectators</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>Press</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>
Data Source and Subsystems

Centralized Data Source

- Scheduling
- Participant Profiles
- Results

Event Website
- Event Website
- Event Website
- Event Website

Subsystems

- Organizers Scheduler
- Judges Intranet
- Press Intranet
- Athlete Intranet
- Event Website
Such a system must ensure **consistency** between different views be **efficient** (it will have to deal with a high load) the subsystems cannot **continuously poll** the data source for content. Should be **event-based**: “when change, keep overall consistency by propagation” the data source **cannot push all** the content to the subsystems.
The Observer Pattern defines an one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Also known as: Dependents, Publish/Subscribe
Part of Model/View/Controller (MVC)
Classic Example

Grade Distribution

<table>
<thead>
<tr>
<th>Grade</th>
<th># of students</th>
</tr>
</thead>
<tbody>
<tr>
<td>100-90</td>
<td>3</td>
</tr>
<tr>
<td>90-80</td>
<td>5</td>
</tr>
<tr>
<td>80-70</td>
<td>10</td>
</tr>
<tr>
<td>70-60</td>
<td>6</td>
</tr>
<tr>
<td>60-50</td>
<td>2</td>
</tr>
<tr>
<td>Less than 50</td>
<td>2</td>
</tr>
</tbody>
</table>

Student Grade

<table>
<thead>
<tr>
<th>Student</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>85.00%</td>
</tr>
<tr>
<td>Bob</td>
<td>95.00%</td>
</tr>
<tr>
<td>Jane</td>
<td>72.00%</td>
</tr>
<tr>
<td>Mary</td>
<td>63.00%</td>
</tr>
<tr>
<td>Cory</td>
<td>56.00%</td>
</tr>
<tr>
<td>Adam</td>
<td>90.00%</td>
</tr>
<tr>
<td>Nancy</td>
<td>21.00%</td>
</tr>
<tr>
<td>Stacy</td>
<td>55.00%</td>
</tr>
<tr>
<td>James</td>
<td>73.00%</td>
</tr>
<tr>
<td>William</td>
<td>75.00%</td>
</tr>
<tr>
<td>David</td>
<td>66.00%</td>
</tr>
<tr>
<td>Richard</td>
<td>68.00%</td>
</tr>
<tr>
<td>Patricia</td>
<td>78.00%</td>
</tr>
<tr>
<td>Linda</td>
<td>84.00%</td>
</tr>
<tr>
<td>Barbara</td>
<td>69.00%</td>
</tr>
<tr>
<td>Paul</td>
<td>83.00%</td>
</tr>
<tr>
<td>Elizabeth</td>
<td>75.00%</td>
</tr>
</tbody>
</table>
The main motivation behind the Observer Pattern is the desire to maintain consistency between related objects without making them tightly coupled.

In our spreadsheet example, we don't want the different representations to be coupled with each other. However, if the information changes in the spreadsheet, all the different representations should be updated to maintain consistency.
Participants

Subject
++++attach(o: Observer)
++++detach(o: Observer)
++++notify()

for each observers
  o.update()

ConcreteSubject
- state: State
++++getState(): State
++++setState(s: State)

ConcreteObserver
- state: State
- subject: ConcreteSubject
++++update()
attach/subscribe: observer “registers” with subject
detach/unsubscribe: let observer no longer observe subject
notify: subject method called when subject state changes
update: inform an observer that new data is available
getState: get subject state after notify (pull method)
update with data argument:
send data to observers (push method)
Sequence Diagram (Push)
Abstraction has multiple aspects/views, each independent.

Separation allows independent modification/re-use.

Unknown number of observers may change dynamically.

No assumptions made about observers except for presence of update().
Minimal coupling between Subject and Observer. The subject does not require knowledge of the observer. The observer only needs to know how to get new data.

Support for broadcast communication. An update() triggers a broadcast communication across all observers.

Unexpected updates. The subject is blind to its observer. Thus, the cost of an update() is unknown.

Observers have no control over when they will receive updates.
The Observer pattern has numerous alternative variants:

- Push vs. Pull
- Observing more than one subject.
- Who stores the subscription?
- Who triggers update?
- Deleting subjects and observers?
- Subject's self-consistency
- Complex subscriptions
- Observer/Subject combo
Push vs. Pull

What are the advantages, disadvantages?
In the pull model, observers are responsible for acquiring the new state after an update() is called.

+ Better transparency, subject doesn't need to know about observer.
+ Observer is free to determine whether it wants to acquire the new state.
- Observer must determine what is new without help from the subject.
In the push model, information about the subject's state change is sent in the update message.

+ **Efficient**: observer does need to determine what was updated.
- Requires the subject to know more about the observer (breaks abstraction).
- Observer **always** automatically receives the update, whether it wants it or not.
In some situations, it might make sense that an observer be attached to more than one subject.

Our current infrastructure is very poor for this.

We don't know which of the observed subjects called the update method.

How can we fix this?
Who stores the subscriptions?

In a traditional Observer Pattern, the subject manages the collection of observers (subscribers).
This adds overhead to that class:
Clutters the API.
Forces it to deal with attach/detach method calls.
In a system with a low number of subscriptions, this is not a problem.
However, this is a burden to the subject if there are many subscriptions.

Solution?
Subscription Manager

SubscriptionManager

subscriptions : HashTable
+publish(s: Subject)
+attach(o: Observer, s: Subject)
+detach(o: Observer, s: Subject)
+notify(s: Subject)

Subject

ConcreteSubject
-state: State
+getState(): State
+setState(s: State)

Observer

ConcreteObserver
-state: State
-subject: ConcreteSubject
+update()
Who can/should trigger notify?
When do we call a notify?
**Who triggers notify?**

**safety vs. performance tradeoff.**

Safety: after every `setState()` call, `notify()` is called (and hence `update()` messages are sent).

+ This ensures a **consistent** state at all times.
- It's very **expensive** when there are many `setState()` calls.

Performance: we do a `notify()` after having completed an appropriate number of `setState()` calls.

+ We don't flood the system with `update()` calls (**performance**).
- There is a danger of having **inconsistencies**.
- There is a danger that the call to `notify()` is **forgotten**.
If a subject is deleted, what should happen to its observers?
Deleting the Subject

We could delete the observers, but ...
- Other objects might refer to those observers.
- The observers might be attached to other subjects.

Subject should **notify** the observers before its destruction?
Deleting the Observer

If an observer is deleted, what should happen to its subject?
Deleting the Observer

detach() the observer before deleting it
(in observer's destructor)
An object could be both a subject and an observer. In our example, OS is an observer and a subject. What happens when OS calls S.getState()? Most likely it will update its state, triggering a notify() and subsequently an update() call to O3. What happens if S observes O3? We would get a loop. If an object can be both an observer and a subject, we need to deal with (detect/handle) loops. How?
As already mentioned, the subscription mechanisms could be altered to deal with specific interests. In other words, an observer could specify what part of the state it is interested in.

In a game, register with a player object, but only wish to receive updates about positions.

In an online investment application, register with the stock exchange object, but only wish to receive updates about stocks trading for more than 10$.

Trade-off: while the complete state does not need to be sent, we have to keep track of what each observer wants.
In the scenario where different observers have specific interests, each observer must be tracked separately. When the state of a subject is modified, each observer must be checked.

Information sent to the observers depends on their individual subscriptions.

In some cases, update() not even have to be called.

- This means we are no longer broadcasting information in a generic fashion.
- Preparing and sending each of these updates is very time consuming.
Do you see a problem?

ConcreteSubject
- state: State
  + setState(s: State)

state = s notify()

SpecialConcreteSubject
  + setState(s)

super.setState(s)
  state++
Self-consistency

Special care must be taken when extending the subject object.
The trick is that every method must respect self-consistency as a pre-condition and post-condition.
This means that before the state is changed, the system should be consistent.
This also means that after the state is changed, the system should also be consistent (or at least converge towards a consistent state).
Instead of sub-classing, the template method design pattern is much more secure (against inadvertedly introducing inconsistency).
Template Method Pattern

Define the **skeleton of an algorithm** in an operation, deferring some steps to subclasses.

Template methods **refine certain steps of an algorithm without changing an algorithm's structure**.
Every concrete class can have its own primitive operations; the template calls these functions.
The main challenge in template methods is making sure the method is used properly.
Users need to know and understand which methods need to be overridden and which method is the template.
Luckily, most OO programming have constructs that help us out with this.
Abstract methods, final methods, etc.
One of the most important things to keep in mind is to minimize the number of primitive operations.
Keeps things simple and easier to implement.
Template Method allows us to solve the self-consistency problem.
The idea is that the setState() method should be a template method with notify() as its last line.
Sub-classes can then vary the behaviour of the subject by changing the primitive operations.
Model View Controller (MVC) is an application architecture that heavily depends on the observer pattern.
Model: The domain-specific representation of the information on which the application operates.

View: Renders the model into a form suitable for interaction, typically a user interface element.

Controller: Processes and responds to events, typically user actions, and may invoke changes on the model.
MVC in action

- Client
- View
- Controller
- Model

- action() (Client)
- request() (Controller)
- query() (Model)
- return new data
- update() (Model)
- change() (Model)

- new view is sent to client
- controller chooses new view
Observer in MVC

- Publisher
- Model
- Controller
- View
- Observer
- Observer Pattern
Where is MVC used?

MVC is highly used in web application frameworks such as Struts, Spring, Django, Ruby on Rails, etc.
Most multi-player use a networking scheme that implements the observer pattern ... or a slight variation of it.

In this architecture

Game objects (players, items, surroundings) are the subjects

Game clients are the observers.
Board Games
Implementation
In another popular game ...
In a *massively multi-player distributed* game, broadcasting all state changes to every player of a game is not a viable solution.

Interest management is a techniques that only sends relevant state changes to each player.

Overhead ...
Who should see who?

A  B  C
E
D
B
A
Tiles
The Observer pattern is heavily used in data replication. In data replication, there exists one master copy of the data and several replicas.

- The master copy is considered the subject.
- The replicas are observers, attached to this subject.
- When the master copy is updated, so are the replicas.
- If the master copy is lost, then one of the replicas becomes the new subject.
The primitive operations in group communications are:
- Join a group
- Leave a group
- Send a message to the group.

These primitive operations are very similar to those found in the observer pattern.
- Attach, detach, notify

It turns out that networked observer patterns are often implemented using group communication protocols.