Comp-304 : Observer / Template Methods (cont.)
Lecture 27

Alexandre Denault
Original notes by Hans Vangheluwe
Computer Science
McGill University
Fall 2007
What is a design pattern?
What are the participants of the observer pattern?
Class Diagram

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

Observer
+update()

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

for each observers
  o.update()

ConcreteObserver
-state: State
-subject: ConcreteSubject
+update()

state = subject.getState()
The Observer pattern has numerous implementation concerns:

- Push vs Pull?
- Who stores the subscription?
- Observing more than one subject.
- Who triggers update?
- Deleting subjects and observers?
- Subject's self-consistency
- Complex subscriptions
- Observers/Subject
In a traditional Observer Pattern, the subject manages its own subscriptions.

This adds overhead to that class.
- Clusters the API.
- Forces it to deal with attach/detach method calls.

In a system with a low number of subscription, this is not a problem.

However, this is a burden to the subject if there are many subscriptions.

What can I do?
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
+update()
```

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

- It's a question of safety vs performance.
- Safety: after every setState(), we do a notify() and update() are sent.
  - This insures a consistent state.
  - It's very expensive when there are many setState() calls.
- Performance: we do a notify() after having completed the necessary setStates().
  - We don't flood the system with update() calls.
  - There is a danger of having inconsistencies.
  - There is a danger that the call to notify is omitted.
If a subject is deleted, what should happen to its observer?
Deleting the Subject

- We could delete the observers.
- It's never that simple.
  - Other objects might refer to those observers.
  - The observers might be attached to other observers.
- Maybe the subject could warn the observer?
If an observer is deleted, what should happen to its subject?
Deleting the Observer

- It's important to detach() the observer before deleting it.
- Is there anything different between this detach() and a normal detach() call?
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 an update() call to O3.
- What happens if S observes O1? We would get a loop.
- If an object can be both an observer and a subject, we need to deal with loop.
As we have 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.

- Register with a player object, but only wish to receive updates about positions.
- Register with the stock exchange object, but only wish to receive updates about stocks trading for more than 10$.

While the complete state doesn't need to be sent, we have to keep track of what each observer wants.
In the scenario where subscriptions deal with specific interest, each subscriptions must be tracked separately.

When the state of a subject is modified, each subscriptions must be checked.

- Information sent to the observers depends on their individual subscriptions.
- In particular case, we might need to check the subscription to see if update() is even 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?
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.
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 algorithms structure.
Every concrete class can have its own primitive operations and template calls these functions.
The biggest 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.
Solution to Observer Problem

- 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 it's last line.
- Sub-classes can then vary the behavior of the subject by changing the primitive operations.