Observer Design Pattern

Intent

01
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
- The "View" part of Model-View-Controller.

Problem

02 A large monolithic design does not scale well as new graphing or monitoring requirements are levied.

Discussion

03 Define an object that is the "keeper" of the data model and/or business logic (the Subject). Delegate all "view" functionality to decoupled and distinct Observer objects.

Observers register themselves with the Subject as they are created. Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.

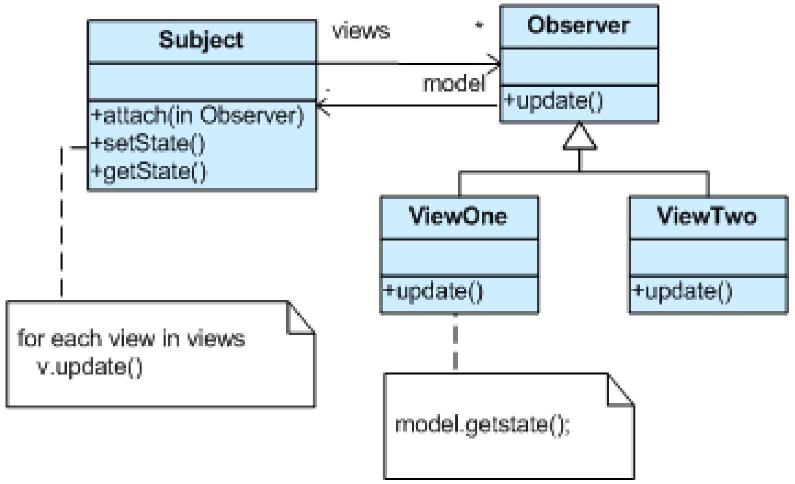
04 This allows the number and “type” of “view” objects to be configured dynamically, instead of being statically specified at compile-time.

05 The protocol described above specifies a “pull” interaction model. Instead of the Subject “pushing” what has changed to all Observers, each Observer is responsible for “pulling” its particular “window of interest” from the Subject. The “push” model compromises reuse, while the “pull” model is less efficient.

06 Issues that are discussed, but left to the discretion of the designer, include: implementing event compression (only sending a single change broadcast after a series of consecutive changes has occurred), having a single Observer monitoring multiple Subjects, and ensuring that a Subject notify its Observers when it is about to go away.

07 The Observer pattern captures the lion's share of the Model-View-Controller architecture that has been a part of the Smalltalk community for years.

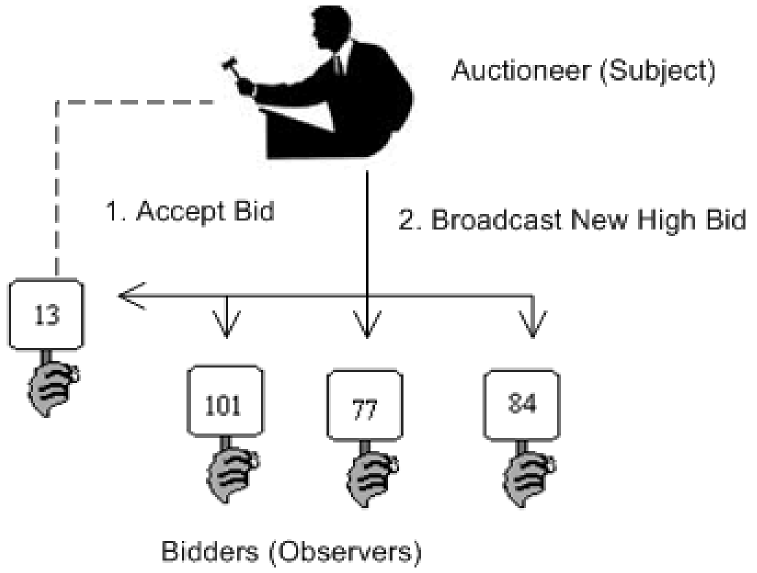
Structure



08 Subject represents the core (or independent or common or engine) abstraction. Observer represents the variable (or dependent or optional or user interface) abstraction. The Subject prompts the Observer objects to do their thing. Each Observer can call back to the Subject as needed.

Example

09 The Observer defines a one-to-many relationship so that when one object changes state, the others are notified and updated automatically. Some auctions demonstrate this pattern. Each bidder possesses a numbered paddle that is used to indicate a bid. The auctioneer starts the bidding, and “observes” when a paddle is raised to accept the bid. The acceptance of the bid changes the bid price which is broadcast to all of the bidders in the form of a new bid.



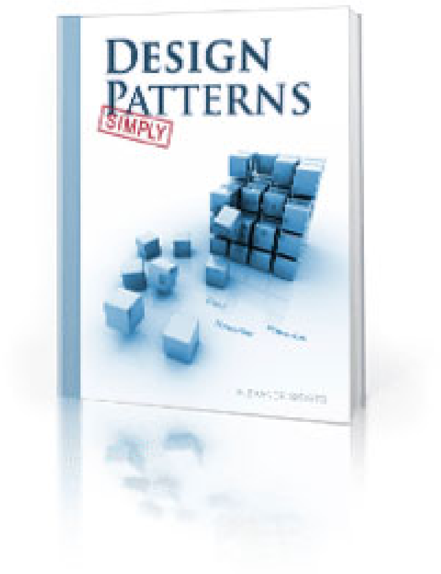
Check list

1. Differentiate between the core (or independent) functionality and the optional (or dependent) functionality.
2. Model the independent functionality with a “subject” abstraction.
3. Model the dependent functionality with an “observer” hierarchy.
4. The Subject is coupled only to the Observer base class.
5. The client configures the number and type of Observers.
6. Observers register themselves with the Subject.
7. The Subject broadcasts events to all registered Observers.
8. The Subject may “push” information at the Observers, or, the Observers may “pull” the information they need from the Subject.

Rules of thumb

- Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers. Command normally specifies a sender-receiver connection with a subclass. Mediator has senders and receivers reference each other indirectly. Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time.
- Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators.
- On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them.

Read next



This article is taken from the book **Design Patterns - Simply**.

ALL design patterns are compiled there. The book is written in a clear and simple language that makes it easy to read and understand (just like this article).

It is a part of our Design Patterns Course. We distribute it in PDF format, so it will be available for downloading in 10 seconds!

Learn more…

12

Observer code examples

C# examples

- Observer in C#

C++ examples

- Observer in C++
- Observer in C++: Class inheritance vs type inheritance
- Observer in C++: Before and after

Delphi examples

- Observer in Delphi

Java examples

- Observer in Java
- Observer in Java

PHP examples

- Observer in PHP

‹ Null Object Design Pattern

State Design Pattern ›