Template Method pattern

• Problem
  – You’re building a reusable class
  – You have a general approach to solving a problem,
  – But each subclass will do things differently

• Solution
  – Invariant parts of an algorithm in parent class
  – Encapsulate variant parts in template methods
  – Subclasses override template methods
  – At runtime template method invokes subclass ops

Structure
Example: JUnit

- Junit uses template methods pattern
  ```java
  Junit.framework.TestCase.run() {
      setUp(); runTest(); tearDown()
  }
  ```
- In class example, subclass (LogRecordTest) overrides runTest() and setUp()
Observer pattern

For all $o$ in observers

ConcreteObserver

observerState = subjectState

Use of Observer pattern

aConcreteSubject

aConcreteObserver

anotherConcreteObserver

Notify()

Update()

GetState()
Observer Pattern (cont’d)

• Consequences
  – low coupling between subject and observers
    • subject unaware of dependents
  – support for broadcasting
    • dynamic addition and removal of observers
  – unexpected updates
    • no control by the subject on computations by observers

Observer pattern (cont’d)

• Implementation issues
  – storing list of observers
    • typically in subject
  – observing multiple subjects
    • typically add parameters to update()
  – who triggers update?
    • State-setting operations of subject
      – Possibly too many updates
    • client
      – Error-prone if an observer forgets to send notification message
Observer pattern (cont’d)

• Implementation issues (cont’d)
  – possibility of dangling references when subject is deleted
    • easier in garbage-collected languages
    • subject notifies observers before dying
  – possibility of premature notifications
    • typically, method in Subject subclass calls inherited method which does notification
    • solve by using Template method pattern
      – method in abstract class calls deferred methods, which is defined by concrete subclasses

Observer pattern (cont’d)

• Implementation issues (cont’d)
  – how much information should subject send with update() messages?
    • Push model: Subject sends all information that observers may require
      – May couple subject with observers (by forcing a given observer interface)
    • Pull model: Subject sends no information
      – Can be inefficient
  – registering observers for certain events only
    • use notion of an aspect in subject
    • Observer registers for one or more aspects
Observer pattern (cont’d)

- Implementation issues (cont’d)
  - complex updates
    - use change managers
    - change manager keeps track of complex relations among (possibly) many subjects and their observers and encapsulates complex updates to observers

Implementation details

- Observing more than one subject.
  - It might make sense in some situations for an observer to depend on more than one subject. The subject can simply pass itself as a parameter in the Update operation, thereby letting the observer know which subject to examine.
  - Making sure Subject state is self-consistent before notification.
More implementation issues

- Implementations of the Observer pattern often have the subject broadcast additional information about the change.
  - At one extreme, the subject sends observers detailed information about the change, whether they want it or not. At the other extreme the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter.
- You can extend the subject's registration interface to allow registering observers only for specific events of interest.

Examples

- The standard Java and JavaBean event model is an example of an observer pattern.