The Observer Pattern

- **Intent**
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
  - Subject notifies registered observers of state changes
  - Dependents (observers) can query subject state

- This pattern resolves the following forces:
  1. An observer has state that should mirror that of its subject, though not continuously
  2. Observers must be decoupled from subject
     - A subject can have any number of observers
     - An observer can monitor multiple subjects
  3. Allows any number of observers
Observer Abstract Interface

```cpp
class Sort_Observer {
public:
    // Manager functions.
    Sort_Observer () {}
    virtual ~Sort_Observer ();

    // Implementor functions.
    virtual void update_sort (const Sort_Response &sort_observer);
    // Callback, to receive a sort observation.
};
```

Subject Abstract Interface

```cpp
// Implementor functions, for use by Sorter clients.
int register_observer (Sort_Observer &sort_observer);
// Register an observer for the sort. Returns 0 on success,
// -1 on failure (reached maximum number of observers).

static unsigned int maximum_observers ();

protected:
    // Manager functions, for use by Sort algorithms.
    Sorter ();

    // Implementor functions, for use by Sort algorithms.
    void publish_sort_update (const Sort_Response &sort_observer);
    // Used by Sort algorithms to publish updates on Sort state
    // to registered observers.
```

Observer Benefits

- Decouples observer from subject
  - Allows an observer to monitor multiple subjects
- Allows any number of observers

The Memento Pattern

- **Intent**
  - Capture and externalize an object’s internal state, without violating encapsulation, so that the object state can be restored later.
    - Hides state storage implementation details from the object.
    - Useful for persistent storage and transaction rollback operations.
- This pattern resolves the following forces:
  1. Hide (and therefore allows transparent change of) state storage implementation details.
  2. Externalize an objects internal state without breaking encapsulation.
### Structure of the Memento Pattern

<table>
<thead>
<tr>
<th>Originator</th>
<th>Memento</th>
<th>Caretaker</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createMemento()</code></td>
<td><code>setState()</code></td>
<td><code>getState()</code></td>
</tr>
<tr>
<td><code>setMemento(Memento &amp;m)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>state</code></td>
<td><code>return new Memento(state)</code></td>
<td></td>
</tr>
</tbody>
</table>

```cpp
state = m.getState();
```

### Use of the Memento Pattern

```cpp
// IterationState is the Memento for our Iterator.
template <class T, class IterationState>
class Iterator
{
public:
    virtual IterationState * first () = 0;
    // Resets the iterator to the beginning.
    virtual int next (IterationState *) = 0;
    // Advance the iterator to the next item.
    virtual int is_done (const IterationState *) const = 0;
    // Returns 1 if we've seen all the items, else 0.
    virtual int current_item (const IterationState *,
        T &item) const = 0;
    // Accesses the current item.
};
```

### Advantages of Memento-based Iterator
- Derived (concrete) `Iterators` need not store any state.
  - The iteration state is stored in the `IterationState` class.
- An `Iterator` does not need to be a `friend` of its collection. Collection-specific private details are confined the Memento (`IterationState` class).
- Readily enables support for multiple iterators on one collection.

### Case Study with the Command Pattern
- Example Use of Command Pattern: Database Operations and Transactions
- Command Pattern
- Example Use of Command Pattern: Java `MenuItem`
- Template Method Pattern, to encapsulate atomicity
- Composite Pattern, to combine Commands into macros
Database Operations and Transactions

- Database operations query or update the state of the database.
- Database systems support transactions, atomic groups of operations.
- Transactions can be arbitrarily complex, but are constructed using simpler building blocks, i.e., database operations such as account debit and credits.
- Operations are verbs, but it would be nice to treat them as nouns (objects).
  - To support logging for audits, crash recovery, rollbacks, etc.
  - To support queuing for batch or remote processing
  - To support undo of completed operations
  - To support security via encryption

Database Operations are Commands

- Database operations are query or update requests on the database.
- The Command Pattern encapsulates operations as objects.
  - Each database operation and transaction manages its own undo operation.
  - Allows composition of operations to create (atomic) transactions.

Database Transactions are Commands

- Database transactions can be complex, and composed of operations or other transactions.
- Database transactions must be atomic, e.g.,
  - Consider moving funds from a savings account to a checking account, implemented by a debit from the savings account and a credit to the checking account.
  - If either the debit or credit fail, then the entire transaction must fail.
  - If the transaction fails but part of it had succeeded, then the successful part must be undone (rolled back).

The Command Pattern

- Intent
  - Encapsulate an operation as an object.
    * parameterize clients with different operations
    * buffer or log operations
    * support reversal (undo) of operations
    * also known as Action and Transaction
- Resolves the following forces:
  - Decouple the object invoking an operation from the object that performs it.
  - Bind an operation to the Receiver (of any class!) that performs it.
  - Treat commands as first-class objects, so that they can be extended, encapsulated, combined, etc.
  - Make it easy to change or add new operations.
Other Example Uses of Command Pattern

- Toolkit objects provide a mechanism for invoking an operation, but the toolkit has no knowledge of the operation.
  - Window buttons and menus: the toolkit provides the mechanism for activating an operation, but the application must supply the actual operation.
  - Encryption/decryption, where part of the decryption algorithm is passed along with the encrypted text.
  - Contrast with Strategy pattern, where the algorithm must be known in advance.
- Downloadable code, e.g., with Java. The code is the operation(s), but it is encapsulated as an object.

Structure of the Command Pattern

1. Application creates the Command and associates it with a Receiver
2. Invoker requests operation
3. Receiver performs operation

Java MenuItem Command Pattern Example

1. Application creates the Command and associates it with a Receiver
2. Invoker requests operation
3. Receiver performs operation

Java MenuItem Participants

- Our Sort Viewer has a Quit MenuItem:

```java
    Sort Viewer quit MenuItem
    Application Command
    Invoker
    Receiver
    Concrete Command
    execute ()
    1a. implements
    Receiver
    action ()
    execute ()
    state receiver
    receiver_->action ()
    3. Receiver performs operation
```

```java
    Menu menu = new Menu ("Sort Viewer");
    // "Quit" is the Command.
    MenuItem quit = new MenuItem ("Quit");
```
Java MenuItem Participants, (cont’d)

- For the Quit Command, we create an ActionListener whose actionPerformed terminates the program.
  - The Receiver is Java’s System class.
  - The Receiver's action is its static exit method.

  ```java
  new ActionListener () {
    public void actionPerformed (ActionEvent e) {
      System.exit (0);
    }
  }
  ```

- Our concrete ActionListener is of an anonymous (unnamed) type.

Limitations of Command Pattern

- Must create one Concrete Command class per type of command.
  - Usually acceptable for MenuItem, because screen real estate limits those to a manageable number.
  - But for, e.g., database operations, there could be many more types of commands.

- Consider adding atomicity to database operations.
  - Database systems already provide atomicity through transactions.
  - Could provide an atomic version of each database operation by subclassing a transaction version of it, e.g.,

    ```java
    Atomic_Debit_Operation () {
      begin_transaction ();
      perform_Debit_operation ();
      end_transaction (); }
    ```

- The Quit MenuItem Invoker calls the the abstract ActionListener’s actionPerformed method.

- The anonymous ActionListener is the Concrete Command.
  - Its addActionListener binds it to its Receiver.
  - Its actionPerformed method calls its Receiver's operation, i.e., exit ()

  ```java
  quit.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e) {
      System.exit (0);
    }
  });
  ```

- However, that doubles the number of Concrete Command classes.

- And, each Atomic_Operation looks similar: begin/operation/end.

- If that similar code needed update, e.g., to add exception handling support, the update would be required for each of the derived Atomic classes.

- Enter the Template Method pattern.
The Template Method Pattern

• **Intent**
  - Define an algorithm skeleton in an operation, but defer some steps to subclasses.
  - Subclasses redefine those steps.
  - The algorithm structure (ordering of steps) are usually fixed.

• Resolves the following forces:
  - Algorithm structure and commonality should be factored out.
  - Only certain steps of the algorithm need to be customized.

• Very useful for avoiding code duplication, of the invariant steps of the algorithm.

• Contrast with Strategy pattern, which uses delegation instead of inheritance. With Template Method, the algorithm steps must be defined at compile time.

---

Combining Command and Template Method Patterns

• Instead of creating a separate version of each database operation that's atomic, use a Template Method, e.g.,

```java
Atomic_Operation () {
    begin_transaction ();
    perform_operation ();
    end_transaction ();
}
```

• Subclasses of abstract `Atomic_Operation` fill in their own `perform_operation()` implementation.

---

Structure of the Template Method Pattern

- **AbstractClass**
  - TemplateMethod ()
    - operation1 ()
    - operation2 ()
    - ...

- **ConcreteClass**
  - operation1 ()
  - operation2 ()

---

Combining Command and Template Method Patterns, (cont’d)

• Can implement the command steps as helper functions in the abstract Template Method base class, and allow subclasses to selectively use or not use them.
  - Maintains fixed algorithm structure.
  - If a step isn’t used, it can be viewed as a no-op.
Reducing the Number of Classes

- The Template Method factors out common steps of an operation, but doesn’t reduce the number of classes.
- We doubled the number of classes when we added atomicity to each Command, by using inheritance.
  - Composition can often be a useful alternative to inheritance.
  - And, it’s useful to be able to add functionality without modifying existing classes.

Back to Transactions

- Transactions are composable
  - May want to perform multiple operations, hierarchically composed
  - Atomically, at various levels
    - Consider disk backup operations at large companies, performed by a central organization:
      - Need a boolean indication of whether the entire backup succeeded or failed
      - If it failed, need to know where
      - And, want to preserve any successful backup transactions
- Enter the Composite pattern

The Composite Pattern

- Intent
  - Compose (aggregate) objects, hierarchically, into tree structures.
    - Represent any part of, or an entire, hierarchy
    - Treat individual objects and recursive compositions uniformly
    - Call an object or composition a component
- Resolves the following forces:
  - Provide a uniform component interface, regardless of its internal complexity.
  - Provide an extensible component interface
Composite Pattern Participants

- Component
  - Provides the abstract interface for composed objects
- Leaf
  - Concrete (primitive) Components
- Composite
  - Non-leaf, concrete Components
- Client
  - User of Composites, via the Component interface

Composite Pattern Collaborations

- Clients call methods on objects, using the Component (abstract base class) methods
- If the object is a Leaf participant, it must have provided implementations for the abstract methods. Therefore, its methods are used.
- If the object is a Composite participant, then it can forward the method call to its subclass objects
  - Using Iteration, for all subclass objects

Example Uses of Composite

- GUIs, again.
- Compound graphical objects (shapes) are perfect examples.
- So are nested menus.
- Modelling manufacturing processes
  - Assemblies are composed of subassemblies . . .
- Compiler parse trees
- Nested transactions
  - Together with Command pattern, to compose operations while supporting atomicity at various levels

Composite Pattern Summary

- Structural
  - Ties objects together, hierarchically
- Commonly used, like Iterator
  - OO approaches factor out interface commonality into (abstract) base classes, to allow polymorphic treatment of instances
  - Composite supports aggregation of objects with such common interfaces, allowing uniform treatment by users