Module 45: Software Engineering Design Patterns

Intructors: Abir Das and Sourangshu Bhattacharya

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

{*abir, sourangshu*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Object-Oriented Analysis & Design

by **Prof. Partha Pratim Das**

Table of Contents

Design Pattern

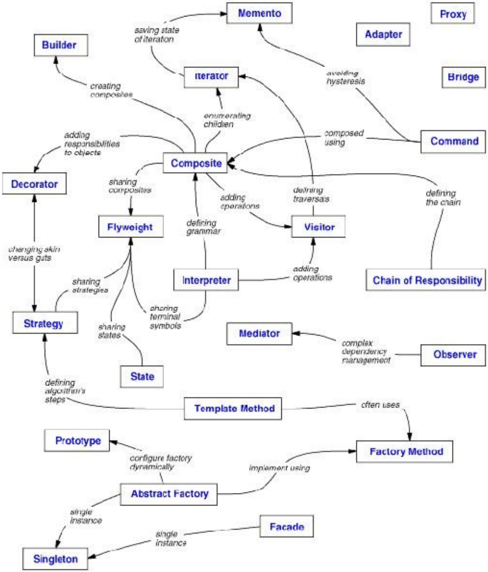
- A **Design Pattern**
  - **describes a problem**
    - *Occurring over and over again (in software engineering)*
  - **describes the solution**
    - *Sufficiently generic*
    - *Applicable in a wide variety of contexts*

Recurring Solution to a Recurring Problem

Catalogue of Design Patterns (GoF)

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Relationships of Design Patterns (GoF)



Describing a Design Pattern

- **Pattern Name and Classification**
  - The pattern's name conveys the essence of the pattern succinctly
- **Intent**
  - What does the design pattern do?
  - What is its rationale and intent?
  - What particular design issue or problem does it address?
- **Also Known As**
  - Other well-known names for the pattern
- **Motivation**
  - A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem
- **Applicability**
  - What are the situations in which the design pattern can be applied?
  - What are examples of poor designs that the pattern can address?
  - How can you recognize these situations?
- **Structure**
  - A graphical representation of the classes in the pattern UML
- **Participants**
  - The classes and/or objects participating in the design pattern and their responsibilities
- **Collaborations**
  - How the participants collaborate to carry out their responsibilities?
- **Consequences**
  - How does the pattern support its objectives?
  - What are the trade-offs and results of using the pattern?
  - What aspect of system structure does can be varied independently?
- **Implementation**
  - What pitfalls, hints, or techniques should you be aware of when implementing the pattern?
  - Are there language-specific issues?
- **Sample Code**
  - Code fragments to implement the pattern in specific language (C++ or C# or Java)
- **Known Uses**
  - Examples of the pattern found in real life
- **Related Patterns**
  - What design patterns are closely related to this one?
  - What are the important differences?
  - With which other patterns should this one

Describing a Design Pattern: Example of Iterator

- **Pattern Name and Classification**: Iterator
- **Intent**: Provide a way to *access* the *elements* of an *aggregate object (container) sequentially without exposing* its underlying representation
- **Also Known As**: Cursor
- **Motivation**
  - An aggregate object (list) should have a way to access its elements without exposing its internal structure
  - There is a need to traverse the list in different ways, depending on a specific task
  - Multiple traversals may be pending on the same list
  - The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an iterator object
- **Applicability**
  - to access an aggregate object's contents without exposing its internal representation
  - to support multiple traversals of aggregate objects
  - to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration)
- **Structure**: Given in Iterator section
- **Participants**
  - *Iterator* defines an interface for accessing and traversing elements
  - *ConcreteIterator* implements the Iterator, keeps track of the current position
  - *Aggregate* defines interface for Iterator
  - *ConcreteAggregate* implements the Iterator to return an instance of ConcreteIterator
- **Collaborations**: A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal
- **Consequences**
  - Variety of the traversals of an aggregate
  - Iterators simplify the Aggregate interface
  - Multiple traversal on an aggregate
- **Implementation**
  - Who controls the iteration?
  - Who defines the traversal algorithm?
  - How robust is the iterator? (insert / delete)
  - Additional Iterator functionality including more operations, polymorphic iterators in C++, optional privileged access, Iterators for composites & Null iterators
- **Sample Code**: Given in Iterator section
- **Known Uses**: Iterators are common in OOP
- **Related Patterns**: Composite, Factory Method, and Memento

Pros & Cons of Design Pattern

- **Pros**
  - Help capture and disseminate expert knowledge
    - Promotes reuse and avoid mistakes
  - Provide a common vocabulary
    - Help improve communication among the developers
  - Reduce the number of design iterations:
    - Help improve the design quality and designer productivity
  - Patterns solve software structural problems attributable to:
    - Abstraction,
    - Encapsulation
    - Information hiding
    - Separation of concerns
    - Coupling and cohesion
    - Separation of interface and implementation
    - Single point of reference
    - Divide and conquer
- **Cons**
  - Design patterns do not directly lead to code reuse
  - To help select the right design pattern at the right point during a design exercise
    - At present no methodology exists

Example Design Patterns

- Iterator
- Singleton
- Factory Method
- Abstract Factory
- Visitor

Iterator Pattern

- **Pattern Name**: Iterator
- **Problem**: How to serve Patients at a Doctor's Clinic?
- **Solution**: Front-desk manages the order for patients to be called
  - By Appointment
  - By Order of Arrival
  - By Extending Gratitude
  - By Exception
- **Consequences**:
  - Patient Satisfaction
  - Clinic's Efficiency
  - Doctor's Productivity

Iterator Pattern: Intent

- Pattern Name and Classification:
  - Iterator
  - Behavioral
- Intent
  - Provide a way to access
  - the elements
  - of an aggregate object (container)
  - sequentially
  - without exposing its underlying representation.

ACCESS
- Read
- Write
- Read-Write

CONTAINERS
- Array
- Vector
- List
- Stack
- Queue
- Tree

SEQUENTIAL
- Forward
- Backward
- Bidirectional
- Random

Iterator Pattern: Sample Code


```
template <class Item>
class List { public: List(long size = DEFAULT_LIST_CAPACITY);
    long Count() const;
    Item& Get(long index) const; // ...
};
template <class Item>
class Iterator { public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected: Iterator();
};
template <class Item>
class ListIterator : public Iterator<Item> { public:
    ListIterator(const List<Item>* aList);
    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;
private: const List<Item>* _list; long _current;
};
template <class Item>
ListIterator<Item>::ListIterator (const List<Item>* aList) : _list(aList), _current(0) { }
template <class Item> void ListIterator<Item>::First() { current = 0; }
template <class Item> void ListIterator<Item>::Next() { current++; }
template <class Item> bool ListIterator<Item>::IsDone() const { return _current >= _list->Count();
}
template <class Item> Item ListIterator<Item>::CurrentItem () const {
    if (IsDone()) { throw IteratorOutOfBounds; } return _list->Get(_current);
}
```

Iterator Pattern: Sample Code


```
// Application using Iterator

void PrintEmployees (Iterator<Employee*>& i) {
for (i.First(); !i.IsDone(); i.Next()) {
    i.CurrentItem()->Print();
    }
}

List<Employee*>* employees;
// ...

ListIterator<Employee*> forward(employees);

ReverseListIterator<Employee*> backward(employees);

PrintEmployees(forward);

PrintEmployees(backward);
```

Iterator Pattern: Sample Code (STL `list`)

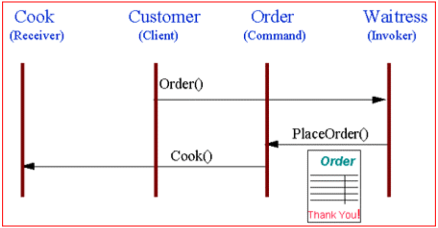
```
#include <iostream>
#include <list>

int main () { // constructing lists
    std::list<int> first;                                  // empty list of ints
    std::list<int> second (4,100);                         // four ints with value 100
    std::list<int> third (second.begin(),second.end());    // iterating through second
    std::list<int> fourth (third);                         // a copy of third

    // the iterator constructor can also be used to construct from arrays:
    int myints[] = {16,2,77,29};
    std::list<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

    std::cout << "The contents of fifth are: ";
    for (std::list<int>::iterator it = fifth.begin(); it != fifth.end(); it++)
        std::cout << *it << ' ';

    std::cout << '\n';

    return 0;
}

-----
Normal Constructor (2 Params): (1, 1)
The contents of fifth are: 16 2 77 29
```

Iterator Pattern: Sample Code (STL map)


```
#include <iostream>
#include <map>

int main () { // map::begin/end
    std::map<char,int> mymap;
    std::map<char,int>::iterator it;

    mymap['b'] = 100;
    mymap['a'] = 200;
    mymap['c'] = 300;

    // show content:
    for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
        std::cout << it->first << " => " << it->second << '\n';

    return 0;
}

-----
Normal Constructor (2 Params): (1, 1)
a => 200
b => 100
c => 300
```

"Check @ Diner" – A Command Pattern

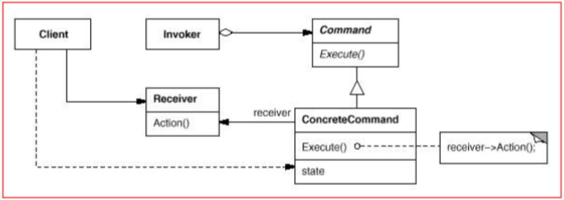
- Customer places an Order with Waitress
- Waitress writes Order on check
- Order is queued to Cook



Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*

Command

- **Command pattern's intent is to encapsulate a request in an object**



- The pattern's main piece is the *Command* class itself. Its most important purpose is to reduce the dependency between two parts of a system: the *invoker* and the *receiver*
- A typical sequence of actions is as follows:
  - The application (`Client`) creates a `ConcreteCommand` object (The dotted line), passing it enough information to carry on a task.
  - The application passes the `Command` interface of the `ConcreteCommand` object to the `Invoker`. The `Invoker` stores this interface.
  - Later, the `Invoker` decides it's time to execute the action and fires `Command`'s *`Execute`* virtual member function. The virtual call mechanism dispatches the call to the `ConcreteCommand` object. `ConcreteCommand` reaches the `Receiver` object (*the one that is to do the job*) and uses that object to perform the actual processing, such as calling its *`Action`* member function
  - Alternatively, the `ConcreteCommand` object might carry the processing all by itself. In this case, the receiver disappears

Command

Module 45

Design Pattern

Iterator

Command

Singleton

- The invoker can invoke Execute at its leisure
- Most important, at runtime you can plug various actions into the invoker by replacing the Command object that the invoker holds. Two things are worth noting here
  - **Interface Separation**: The invoker is isolated from the receiver. The invoker is not aware of how the work is done
    - The invoker only calls for Execute for the Command interface it holds when certain circumstances occur
    - On the other side, the receiver itself is not necessarily aware that its Action member function was called by an invoker or otherwise
    - The invoker and receiver may be completely invisible to each other, yet communicate via Commands
    - Usually, an Application object decides the wiring between invokers and receivers
    - We can use different invokers for a given set of receivers, and we can plug different receivers into a given invoker – all without their knowing anything about each other
  - **Time Separation**: Command stores a ready-to-go processing request to be started later
    - In usual programming tasks, when we want to perform an action, we assemble an object, a member function of it, and the arguments to that member function into a call. For example:

      ```
      window.Resize(0, 0, 200, 100); // Resize the window
      ```

      The moment of initiating such a call is conceptually indistinguishable from the moment of gathering the elements of that call (the object, the procedure, and the arguments)
    - In the Command pattern, however, the invoker has the elements of the call, yet postpones the call itself indefinitely. The Command pattern enables delayed calls as in the following example:

      ```
      Command resizeCmd(                                  // GATHERING THE COMMAND (TASK)
      window, // Object
      &Window::Resize,     // Member function
      0, 0, 200, 100);     // Arguments
      // Later on...
      resizeCmd.Execute(); // Resize the window   // PERFORMING THE COMMAND
      ```

Command: Implementation

- From an implementation standpoint, two kinds of concrete Command classes can be identified.
  - **Forwarding Commands**: Some simply delegate the work to the receiver. All they do is call a member function for a Receiver object. They are called forwarding commands
  - **Active Commands**: Others do tasks that are more complex. They might call member functions of other objects, but they also embed logic that's beyond simple forwarding. They are called active commands
- Separating commands into active and forwarding is important for establishing the scope of a generic implementation
- Active commands cannot be canned – the code they contain is by definition application specific, but we can develop helpers for forwarding commands.
- Forwarding commands act much like pointers to functions and their C++ colleagues, functors, we call them **Generalized Functors** (Refer to *Functor* module for details.

What is a Singleton?

Module 45

Design Pattern

Iterator

Command

Singleton

- *Ensure a class only* **has** *one instance, and* **provides** *a global point of access to it* (GoF Book)
- We should use Singleton when we model types that conceptually have a unique instance in the application, such as `Keyboard`, `Display`, `PrintManager`, and `SystemClock`
  - Being able to instantiate these types more than once is unnatural at best, and often dangerous
- *A singleton is an improved global variable*
  - The improvement that Singleton brings is that *we cannot create a secondary object of the singleton's type*
  - *The Singleton object owns itself*
  - There is no special client step for creating the singleton – *the Singleton object is responsible for creating and destroying itself*
  - Managing a singleton's lifetime causes the most implementation headaches
- The Singleton design pattern is queer in that it's a strange combination:
  - *Its description is simple, yet its implementation issues are complicated*
- There is no **best** implementation of the Singleton design pattern
- Various Singleton implementations, including non-portable ones, are most appropriate depending on the problem at hand

Static Data + Static Functions != Singleton

- Can a Singleton be implemented by using static member functions and static member variables?

```
class Font { ... };
class PrinterPort { ... };
class PrintJob { ... };

class MyOnlyPrinter { public:
    static void AddPrintJob(PrintJob& newJob) {
        if (printQueue_.empty() && printingPort_.available()) {
            printingPort_.send(newJob.Data());
        }
        else { printQueue_.push(newJob); }
    }
private: // All data is static
    static std::queue<PrintJob> printQueue_;
    static PrinterPort printingPort_;
    static Font defaultFont_;
};

PrintJob somePrintJob("MyDocument.txt");
MyOnlyPrinter::AddPrintJob(somePrintJob);
```

- However, this solution has a **number of disadvantages** in some situations
  - The main problem is that static functions *cannot be virtual*, which makes it *difficult to change behavior* without opening `MyOnlyPrinter`'s code
  - A subtler problem of this approach is that it makes *initialization and cleanup difficult*. There is no central point of initialization and cleanup for `MyOnlyPrinter`'s data. Initialization and cleanup can be nontrivial tasks; for instance, `defaultFont_` can depend on the speed of `printingPort_`
- Singleton implementations therefore concentrate on *creating and managing a unique object* while *not allowing the creation of another one*

C++ Idioms for Singleton

- Most often, singletons are implemented in C++ by using some variation of the following idiom:

```
// Header file Singleton.h
class Singleton {
public:
    static Singleton* Instance() { // Unique point of access
        if (!pInstance_)
            pInstance_ = new Singleton;
        return pInstance_;
    }
... operations ...
private:
    Singleton(); // Prevents creating a new Singleton
    Singleton(const Singleton&); // Prevent creating a copy of the Singleton
    static Singleton* pInstance_; // The one and only instance
};
// Implementation file Singleton.cpp
Singleton* Singleton::pInstance_ = 0;
```

- *All the constructors are private, user code cannot create Singletons*
- *Singleton's own member functions,* `Instance()` *in particular, are allowed to create objects*
- *The uniqueness of the Singleton object is enforced at compile time*
- **This is the essence of implementing the Singleton design pattern in C++**
- *If it's never used (no call to* `Instance()` *occurs), the Singleton object is not created*
  - The cost of this optimization is the (usually negligible) test incurred at the beginning of `Instance()`
  - The advantage of the build-on-first-request solution becomes significant if Singleton is expensive to create and seldom used