Design patterns generic models

Jyothish Maniyath
CDC Software India Pvt Ltd
6th Floor, Canberra Block, UB City, #24
Vittal Mallya Road, Bangalore, India
+91 94482 46718
jyosh@maniyath.com

ABSTRACT
This paper discusses about generic models of software design patterns defined in terms of design patterns' programming meaning or effective execution behaviors. The study is based on 23 design patterns cataloged in the seminal book by famous GoF authors: Design patterns, Elements of Reusable Object-Oriented Software. The generic models defined in this paper are a different way to understand abstract intents of design patterns. And also it will be helpful to analyze and understand similarities and differences in among design patterns.

Categories and Subject Descriptors
D.2.2 [Design Tools and Techniques]: Object-oriented design methods

General Terms
Design, Language.

Keywords
Design patterns, generic models

1. INTRODUCTION
Programming language features are just building blocks and in programs they have to be used in combination to solve complex problems. Sometimes certain combinations or direct usage of certain features can make programs inefficient to modify or extend latter. The importance of design patterns is they describe optimal and proven combinatory usage of, especially, object-oriented language features to address recurring design and programming problems.

Decision of choosing appropriate design patterns is made primarily based on their abstract intents. Goal of this paper is to define generic models of design patterns based on their programming meaning and effective execution behaviors. The study is based on the 23 design patterns cataloged in the seminal book, Design patterns, Elements of Reusable Object-Oriented Software. The generic models will be useful to analyze and understand the similarities and differences in among design patterns

2. Design Patterns generic models catalog
In total nine generic models are identified and defined in this catalog. First a summary table of the generic models and related design patterns are given. For most of the models there are two or more variations. In the table basic models are given in bold letters and variations underneath.

Table 1. Summary table of models and related design patterns

Generic models and variations	Design patterns
In-memory object reuse	
Enforced	Singleton
Managed	Flyweight
Object indirection	
With different interfaces for compatibility concern	Adapter
With different interfaces for decoupling concern	Command
With common interface	Proxy
With collections	Iterator
Behavior indirection	Visitor
Callback dependency	
Unidirectional	Observer
Multidirectional	Mediator
Class type reuse with composition	
Behavior composition	Strategy
Matrix model	Bridge
Agent based object instantiation	
Inheritance oriented	Factory method
Meta-agent based	Abstract factory
Composition oriented	Builder
Self-agent	Prototype
Objects hierarchy with common interface	
Call forwarding	Chain of responsibility
Aggregated states	Composite
Aggregated behaviors	Interpreter
Behavior refinement	Decorator
Abstraction decomposition	

Monolithic presentation	Facade
Distinctly dynamic object abstraction	State
Procedure abstraction	Template
Object in-memory persistence	Memento

2.1 In-memory object reuse

Reuse of runtime objects can help to improve program performance or it is sometimes required for program correctness. With in-memory object reuse program performance can be improved by the optimal use of runtime resources. In the cases where a high chance for a significant number of instances of a class needs or happens to be created at runtime, reuse of in-memory objects is necessary for the optimal performance of the program. In some other cases, as part of program correctness it requires that either only single instance of a particular class can exists during the lifetime of the program or an object with a particular identity is once instantiated has to be reused everywhere in the program.

There are two main approaches to achieve in-memory object reuse, in first approach the target object itself ensures its reuse and in second one the client code, itself or with the help of an agent object manages the reuse of target objects.

In the first approach reuse of the target object is enforced by the class definition. So it can be called as enforced reuse. In the basic model of this approach only single instance of a class can be instantiated in a program. In a variation of it, it is allowed to create multiple instances with different identities but objects with a particular identity is once instantiated its reuse is enforced everywhere in the program. It can be implemented by maintaining a list of instantiated objects against a selected identity property based on key-value pairs.

The Singleton design pattern is an example for enforced in-memory object reuse. In the basic model the participant *Singleton* ensures only a single instance of the class is created in client code. A variation of it can be created to support multiple instances with unique identities by implementing a key-value pair list to store multiple static instances instead of single static instance like in the basic model.

In second approach either client code itself or with the help of an agent object, target object instances are reused. In such cases the target object class has to be designed in such a way that its instances can be reused independent of their identity. This approach can be called as managed reuse because it is possible to create as many target objects in client code but the reuse of objects is managed by client code logic with support of target object class design.

An example for managed in-memory reuse is the Flyweight design pattern. In that pattern the *FlyweightFactory* participant acts as the agent that manages the reuse and the *ConcreteFlyweight* participant class is designed to support reuse of its objects in a managed way. The name flyweight implies target objects' abstraction is designed to create object as lightweight by including only intrinsic properties and extrinsic properties required in behaviors has to be provided from client code.

2.2 Object indirection

Object indirection is about client objects interact with or access target objects indirectly using intermediate objects. There are basically three types of objects in this model – client object, in-directing intermediate object and in-directed target object. The in-directing object may implement a same or a subset of interface of the in-directed object or altogether a different interface that is compatible to the client code depends on the requirement of the client object.

One scenario is object indirection is used to support interaction of client object with a target object that has an incompatible interface. In such a case the in-directing object implement the interface that is compatible to the client code and the invocations to them are redirected to corresponding implementation in the target object. For redirection the in-directing object internally maintains a reference to the in-directed object. An example of this model is the Adapter pattern. The participant *Adapter* acts as the in-directing object to access the target object, the *Adaptee* participant. A different model of the same pattern is the abstraction indirection in which the in-directing object implements both the client code compatible interface and the in-directed object interface instead of maintaining a reference to the target object.

The Command pattern is also based on object indirection similar to the Adapter pattern in which the in-directing object internally maintains the in-directed object reference and implements an interface with more general actions like 'Execute'. The main application of the command pattern is that the specific actions with different names associated with different types of objects can be accessed from the client object uniformly with a common interface. So the major accomplishment of the Command pattern is loosely coupling so that the target objects themselves or their code can be changed without any impact on the client code.

So far we have discussed in-directing and in-directed objects with different interfaces with respect to the client object. Another possible variation of object indirection is both the objects implements a common interface in complete or a subset of it. An example for such an object indirection is the Proxy design pattern.

Applications of the proxy pattern include – lazy initialization, secure access, remote access etc. In the cases where target objects are heavyweight, lazy initialization is useful to defer their instantiation until they are actually required. In the scenarios when certain interfaces of a target object has to be hidden from the client code access to the target object can be controlled through a proxy object, which will implement only a subset of the interfaces. In the case of remote access when a client code wants to access target object on a different process in the same machine or on a different machine proxy objects are used to shield remote communication complexities from the client code.

The Iterator design pattern is a different type of object indirection than those are discussed so far. The iterator pattern is used to work with collection objects. Instead of client code keep track the position of objects in a collection an iterator is used for forward or backward sequential traversal of it. The *iterator* participant provides options to access the objects in a collection by hiding the details of the position information.

2.3 Behavior indirection

In this model the behaviors that are logically belong to the target object abstractions will be implemented in separate abstractions and the target object abstractions will provide an indirection method to invoke them. The method will redirect the call to appropriate behavior implemented externally. Applications of behavior indirection includes, allow implementing new virtual methods later without modifying already implemented class abstractions and simulating multi-methods.

The Visitor pattern is an example of behavior indirection. The *ConcreteEelement* participant implements a general (like 'Accept') method which redirects the call to a *ConcreteVisitor* method (which actually implements the behavior belongs to *ConcreteEelement*).

2.4 Callback dependency

Callback is implicit invocation of methods of one or a set of objects upon an event occurs in another object. The events can be some state change, execution of certain actions or a user /external application interaction. A dependency results from the usage of callback mechanism can be called as callback dependency. There are two types of objects in callback dependency relationship – publisher and subscriber objects. The publisher and subscriber objects are associated through a callback interface or an anonymous/delegate method. In the interface based callback dependency relationship, subscriber objects implement the callback interface and publisher object keeps the references of subscriber object based on the callback interface. Otherwise in a method based callback a method reference is passed to the publisher object.

In a simple model of the callback dependency the relationship is unidirectional in which there will be a dedicated publisher object that is associated with one or more subscriber objects. In a more complex model a set of objects acts as both publisher and subscriber and interconnected with one another through callback dependency, which can be called as multidirectional callback dependency.

The Observer design pattern is an example for unidirectional callback dependency. In it the *ConcreteObserver* participants can register with a *ConcreteSubject* participant and the *ConcretSubject* will callback all registered observers when the event, which they are subscribed for, occurs.

The Mediator design pattern is an example for multidirectional callback dependency. In it generally all participating objects are interested in events that occur in other participating objects. So to simplify the communication among these objects, a mediator object is created and participating objects register with it to get notified about other object's events. This avoids the complexity of interactions among the participants and makes fewer dependencies among them. In the mediator pattern, *Colleague* class participants register with a *ConcreteMediator* participant and *Colleague* classes include a reference of the *mediator* participant. And when an event happens in any colleague class it executes a callback to the mediator object and it will callback appropriate registered colleague classes methods.

2.5 Class type reuse with composition

Common states, behaviors and contracts shared across multiples class types can be encapsulated into a common base class and reused to avoid redundancy and improve maintainability. There are two options to integrate reusable abstractions – inheritance and composition. Inheritance based integration is a compile time operation so any change or addition requires recompilation. Composition based integration come into effect only at runtime so it can be applied dynamically.

In the inheritance-based approach, a reusable base abstraction is extended by adding variant states and behaviors to create different specializations of it. Different specialized abstractions created via inheritance basically represent different types. So if client code wants to work with different derived types dynamically the polymorphism feature can be utilized.

A basic model of composition-based approach consists of one context class with common states and behaviors and different variant classes with varying states and behaviors. Different types of target objects are created at runtime by combining instances of context and appropriate variant classes. With the composition based approach client code can create different variations of target objects without directly using polymorphism. One basic goal of composition based class type reuse is reduce sub-classing. There can be different specializations of the basic model with respect to how the variant abstractions are designed and/or context and variant classes' instances are combined to create target objects.

The Strategy design pattern is an example for composition-based class type reuse model. This pattern is applicable if a set of abstractions differs only in the implementation of one or few behaviors. In such cases, variant behaviors can be implemented as part of different abstractions and composed with context abstraction to create variations when required. If we want to implement a similar thing using inheritance, first need to create a base abstraction with variant behaviors as virtual methods together with invariants and then create different derived classes implementing the variant virtual method. In strategy pattern the *Context* participant is a configurable context abstraction and the *ConcreteStrategies* are variant abstractions that are passed to the *Context* to create variations of it.

In the Builder and the Prototype patterns a similar model can be observed but the main intent of them is 'agent based object instantiation' which is explained in the next section. In the State pattern also same model is followed but its main intent is 'decomposition of distinctly dynamic object abstraction' which is also explained latter.

The Bridge pattern is based on both inheritance and composition. In the bridge pattern there can be different specializations of base abstractions and variant abstractions. These specializations can be combined together to create different permutations among them. If this combination were performed through only the inheritance it would result in a much larger number of specialized abstractions. The *Abstraction* participant is the base abstraction for *RefinedAbstractions* participant. The *Implementor* participant is base abstraction for the *ConcreteImplementor* participant. Multiple specialized *RefinedAbstractions* and *ConcreteImplementor* can be created through inheritance and in client code a larger number of variations can be created by composing different combinations of instances of them.

2.6 Agent based object instantiation

A straightforward way to instantiate class abstractions is to use the new operator with the class constructors, but there are some

limitations associated with this approach. Direct use of the new operator makes the coupling between the abstractions tight and thereby introduces a direct dependency. One solution to avoid such a tight coupling is to use an agent object, which can create target objects on demand, by providing the necessary object initialization parameters.

The simplest example for agent based object instantiation is the Factory method design pattern. In it the *ConcreteCreator* participant acts as the agent object, which provides a method to create different *ConcreteProducts* and it returns a reference of type *Product* interface. Client code needs to pass necessary parameters while calling the object creation method and based on that the agent object decides which object to be created and returned.

The Abstract factory is a special case of the Factory method pattern. Basically it is a nested implementation of it. The *AbstractFactory* participant implements a method to create a *ConcreteFactory*. The *ConcreteFactory* implements a method to create *ConcreteProducts* on-demand and returns *AbstractProduct* type interface reference. Actually an Abstract factory is a meta-agent object, which creates the appropriate agent object on-demand and the newly created agent creates target objects required for the client code.

The Builder pattern unlike the Factory method pattern is not based on inheritance. In this pattern the *Director* participant acts as an agent to create target objects but client code can decide the internal structure of the target object, *Product*, through the *ConcreteBuilder* participant. The Product and the *ConcreteBuilder* participants can be designed to create objects with different internal structures based on composition or simple value change. Because it is not based on inheritance for creation of variations, composition based abstraction reuse approach, that is explained in the previous section, can be applied to create variations.

In the Prototype pattern the target object itself act as an agent to create new object. Once an object of same type is created through an existing object, client code can change the variant parts to create an object with different internal structure or state. The *ConcretePrototype* participant acts as an agent to create new objects. Similar to the Builder pattern it can be designed to create different variations using composition based abstraction reuse or simple value change.

2.7 Object hierarchy with common interface

Inheritance in multiple levels will create a class hierarchy in which derived classes aggregate all possible states and behaviors of all the base classes above to it in the hierarchy. In such a hierarchy we can observe two types of method binding. In first case, if a method is invoked with a derived class that is not implemented in it then it will try to bind to an implementation of its base class in bottom up order until a matching implementation is found. In second case a virtual method is invoked with a base interface reference pointing to a derived class object then it will try to bind an implementation by derived class in bottom up order.

Analogues to inheritance based class hierarchies it is possible to create object hierarchies based on composition. The object hierarchies with common interface are a special case of it in which all the objects in the hierarchy implements a common interface. There are two variations possible with the hierarchy – linear structure and tree structure. In the linear structure an object will be linked to only one object in the hierarchy but in tree structure an object can be linked to more than one object at one level and it can be implemented recursively. Some applications of the model 'object hierarchy with a common interface' are explained below.

Control forwarding: In this case when a common interface method is invoked with an object first it checks is it configured to execute it, if it is then it executes and returns else it forwards the control to upward or downward in the hierarchy depend upon the design.

The Chain of responsibility pattern is based on one directional control forwarding. A set of *ConcreteHandler* participants implements the *Handler* interface and forms a logical hierarchy by referencing another *ConcreteHandler*. The *Client* participant can invoke the method enabled with control forwarding using a header object in the hierarchy and control forwarding will happen internally until an implementation gets executed.

In the Chain of responsibility pattern control forwarding happens only in one direction and the forwarding is terminated when reached at an appropriate implementation. Different variations of control forwarding are possible like, perform control forwarding in top-to-bottom or bottom-to-top direction, by default always continue with control forwarding till reach to root or bottom of the hierarchy with an option to terminate the forwarding conditionally at any level etc. Such patterns can be observed in event tunneling and bubbling implementations.

Aggregated states, aggregated behaviors and refined behaviors: In this model when a common interface method is invoked with an object it internally recursively calls implementations of all the objects below to it in the hierarchy.

For some objects value of certain states can be an aggregation of values of similar state in all objects they are composed with. For example, cost of a machine can be total cost of all of its components or weight of a composite object where each sub-component will have its own weight and weight of the composite object will be a total of them. In such cases the composite object is represented by an object hierarchy with a common interface. And all the composing objects implement a state value computing method to calculate its value recursively.

In the case of expression evaluation or similar scenarios the result of a top level expression will be aggregation of result of evaluation of sub-expressions done recursively. It is an example of aggregated behaviors.

In case of refined behavior, behaviors of objects on top of the hierarchy get refined in objects at bottom of them. It is done by executing additional actions after or before the execution of behaviors of upper level objects in the hierarchy.

The Composite pattern is based on tree structure hierarchy, which is suitable to implement aggregated states. In it *Leaf* and *Composite* participants implement the common interface *Component*.

In the interpreter design pattern we can observe the aggregated behavior model. The interpreter pattern is a solution to the problems which can be expressed using a language and its actual functioning will be based on the interpretation of the language by expression evaluation. Generally expression trees are represented using object hierarches and their evaluations are done recursively. The evaluation result of top level expression will be aggregation

of evaluation results of sub-expressions. The participants *TerminalExpression* and *NonterminalExpression* implement *AbstractExpression* common interface. And the *Client* participant with the help of the *Context* participant creates the object hierarchy. The result of the 'evaluate' behavior of the top level *NonterminalExpression* will be aggregation of results of 'evaluate' behavior of other *NonterminalExpression* and *TerminalExpression* participants in the hierarchy.

The Decorator pattern is basically a linear structure hierarchy and is mainly used for behavior refinement. In this pattern, *ConcreteComponent* and *ConcreteDecorator* participants implement a common interface *Component*.

2.8 Abstraction decomposition

Data abstraction (class) and procedure abstraction (behavior or method) are the two important abstractions in object-oriented programing. It is always desirable to design abstractions with single or fewer cohesive responsibilities. Designing systems and subsystems based on granular abstraction units are helpful to manage complexity, modify and maintain them individually and as a composite group.

An opposite approach to it is designing them as big monolithic abstraction units. Monolithic design has certain advantages over the highly decomposed design of abstractions while considering the lesser-complicated interfacing option available to the client code. That means it is easy to provide the services or functionalities the client codes want to consume in more abstract form by hiding the details, in the monolithic design. In the case of decomposed approach using granular abstraction units in combination will be complicated to client codes.

Monolithic presentation is a technique to utilize the advantages of both approaches. In this model, systems or complex abstractions will be decomposed into granular abstraction units and a monolithic presentation abstraction is created to hide the complexity of granular decomposition. The Façade design pattern is an example for monolithic presentation. The Façade pattern describe about providing a packaging *Façade* abstraction around decomposed *Subsystems*.

Objects are dynamic entities because during runtime the states and behaviors of them can change. But an abstraction can be called as 'Distinctly dynamic objects abstraction' if significant behaviors of the objects of it behave distinctly differently when one or a small set of states value is changed. In such cases it is possible to decompose the abstraction into a base abstraction to encapsulate common states and behaviors and a set of variant abstractions to represent the distinct behaviors.

The State pattern is an example for decomposition of 'distinctly dynamic object' abstraction. In model a possible monolithic 'distinctly dynamic object' abstraction can be decomposed into one context abstraction and multiple variant abstractions that encapsulate distinct behaviors. The target objects are created by composing the participant *Context* class object and the participant *ConcreteState* subclass objects.

The Template method design pattern is about decomposition of procedure abstraction. A template method defined in the base class maintains an order for a sequence of sub-procedures, which can be implemented differently in sub-classes. The *AbstractClass* participant declares a *Template* procedure and *Abstract Primitive* abstractions. The Template procedure will be an ordered composition of primitive operations, and primitive operations can be overridden in *ConcreteClass* participants to create variations. So such a behavior also can facilitate behavior generalization and specialization

2.9 Object in-memory persistence

Certain languages or its base library provide a feature to serialize or de-serialize object states in order to save objects states before they are destroyed or changed and to recreate objects with same states later. If the states need not be saved across sessions it is a less efficient approach because it involves costly file operations. In such cases the persistence can be done optimally within primary memory itself (for example, undo/redo operations during editing).

The Memento design pattern describes how to accomplish in-memory persistence. One or more states of *Originator* object can be serialized in *Memento* object 'in memory' and restored later to the same or another object. Memento objects can be kept inside a *Caretaker* object for better management.

3. ACKNOWLEDGMENTS

The author would like to thank following individuals for their valuable comments and suggestions provided to improve the paper during the shepherding and writers' workshop in PLoP 2011. David Isaacs – shepherded the paper for PLoP 2011, Ademar Aguiar, Eduardo Guerra, Fernando Sergio Barbosa, Filipe Correia, Joseph Yoder, Rebecca Wirfs-Brock – reviewed the paper in writers workshop PLoP 2011.

4. REFERENCES

[1] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. 1995. *Design patterns - Elements of reusable object oriented design*. Addison Wesley.

[2] James.O. Coplien. 1992. *Advanced C++ - Programming Styles and Idioms*. Addison Wesley.

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal. 1996. *Pattern Oriented Software Architecture Volume 1*. John Wiley & Sons.