Analysis of Design Patterns in Java with Rational Rhapsody as the New Building Blocks of Software Engineering

Bhushan Bharat Ghatkar[1], Amruta Ramchandra Pandhare[2]

[1] *Student, MCA SEM VI, DES's NMITD, Mumbai, Maharashtra, India*
[2] *Student, MCA SEM VI, DES's NMITD, Mumbai, Maharashtra, India*

ABSTRACT

The use of object-oriented design patterns in Software development is being analysis in this paper. Design patterns are solutions to common problems that developers faced during software development. In this study, we analysis Design Patterns by using Rhapsody in Java. We show how Java language features affect the implementation of Design Patterns using Rhapsody, analysis some potential Java programming patterns, and demonstration of how studying design patterns can contribute to provide the reusable solution to a commonly occurring problem within a given idea in software design.

Keywords*: Design Patterns, Java, Rational Rhapsody, Object-Oriented.*

1. INTRODUCTION

As we developed more complex computer systems, we are going to face difficulties in construction rather than difficulties in analysis. The difficulties in developments are solved by designing programming solutions for such difficulty in the context of the software application we are trying to develop.
Some constructional problems occur over and over again across a wide range of different Software. Obviously, we can design a common solution to such repeating problems, and then try to adjust such a common solution to the specific need of the software we are currently building. Such common solutions are usually referred to as design patterns.

Design Patterns are very popular among software developers. A design pattern is a well described solution to a common software problem. We have analysis all design patterns type in Java Programming language.
In this paper, we use design patterns to evaluate a programming language. There are a number of books describing the GoF (Gang Of Four) design patterns in different programming languages (Smalltalk, Java, JavaScript, Ruby, PHP). Implementations of design patterns differ due to specifics of the language used. There is a need for programming language specific pattern descriptions. Our use of patterns should give programmers insight into how Java-specific language features are used in everyday programming, and how gaps compared with other languages can be bridged.

We have implemented all 23 patterns from Design Patterns in Java: for space reasons we discuss only the Factory, proxy, Strategy and Composite Method patterns in this paper. The implementations of these patterns illuminate specific Java language features: Factory demonstrates for creating instances for your classes[1]; Composite allows clients to operate in generic manner on objects that may or may not represent a hierarchy of objects[1]; Proxy provides the control for accessing the original object[1];and we create objects which represent various strategies and a context object whose behavior varies as per its strategy object[1]. To analysis how design patterns integrate in Java, we ported the core of the well-known IBM Software Rational Rhapsody into Java.

This paper is organized as follows: in Section 2 we give a brief overview of Design Patterns in Java; in Section 3 we give a brief overview of Rational Rhapsody; in Section 4 we present case studies implementing four design patterns and Rational Rhapsody in Java; in Section 5 we discuss our experience using Java and Rational Rhapsody, with a focus on design patterns; in Section 6 we discuss future work; in Section 7 we conclusion.

2. OVERVIEW OF DESIGN PATTERNS IN JAVA

2.1 Concept of Design Pattern

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [AIS+77]. Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns. Our solutions are expressed in terms of objects and interfaces instead of walls and doors, but at the core of both kinds of patterns is a solution to a problem in a context.[1]
In general, a pattern has four essential elements:

1. The pattern name is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and even to ourselves. It makes it easier to think about designs and to communicate them and their trade-offs to others. Finding good names has been one of the hardest parts of developing our catalog. [1]
2. The problem describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern. [1]
3. The solution describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it. [1]
4. The consequences are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them. [1]

2.2 Desirable feature of Design Patterns

1. make software development easier
2. design patterns help developers correctly work through the application
3. improve coding efficiency and final product readability
4. Improving existing system maintenance to create a better help tool.
5. Knowledge of design patterns often brings insight in designing flexible software.

2.3 Limitation of Design Patterns

1. Design patterns are hard to understand. It takes a while for non-experienced developers to understand patterns.
2. Preoccupation with design patterns sometimes keeps developers from finding simpler solutions. In some case, there is a simpler solution to a facing problem than one using a pattern. If a programmer believes a solution with patterns is an ultimate one, he or she may miss small, simple, and straightforward solution.

2.4 Design Pattern Categories

There are 23 design pattern and these patterns are classified into categories .They are divided into three parts Creational, Structural and Behavioral patterns.

Following table gives description about three major types of Design pattern.

Table -1: Design Pattern Description

Category	Description
Creational Patterns	These design patterns hide the creation logic while creation of object, rather than ins tainting object using NEW operator. This gives program more flexibility in deciding which objects need to be created for a given use case.
Structural Pattern	These design patterns concern with the class and object composition. Inheritance is used in it in order to compose interfaces and define ways to create object in order to obtain new functionalities.
Behavioral Patterns	These design patterns are specifically concerned with communication between objects.

2.5 Design Pattern Space (SCOPE)

The term SCOPE specifies whether the pattern applies to class or object. The pattern under the class label is those that focus on class relationship and that are in object label are focus on the object relationship. Class patterns deal with the relationship between classes and their child classes and these relationships are made by inheritance and almost all patterns use inheritance to some extent. Object patterns deal with object relationships, which can be changed at run -time and are more dynamic.[4]

Creational class patterns defer some part of object creation to subclasses, while Creational object patterns defer it to another object. The Structural class patterns use inheritance to compose classes, while the Structural object patterns describe ways to assemble objects. The Behavioral class patterns use inheritance to describe algorithms and flow of control, whereas the Behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone.[5]

Following table shows the scope of Design pattern:

Table -2: Design Pattern Scope

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory method	Adapter (class)	Interpreter, Template
	Object	Abstract factory, Builder, Prototype, Singleton	Adapter (object), Bridge, Composite, Decorator, Facade, Flyweight, Proxy	Chain of responsibility, Command , Iterator, Mediator, Memento, Observer , State, Strategy, Visitor

Following figure 2.5.1 shows the Design pattern types:

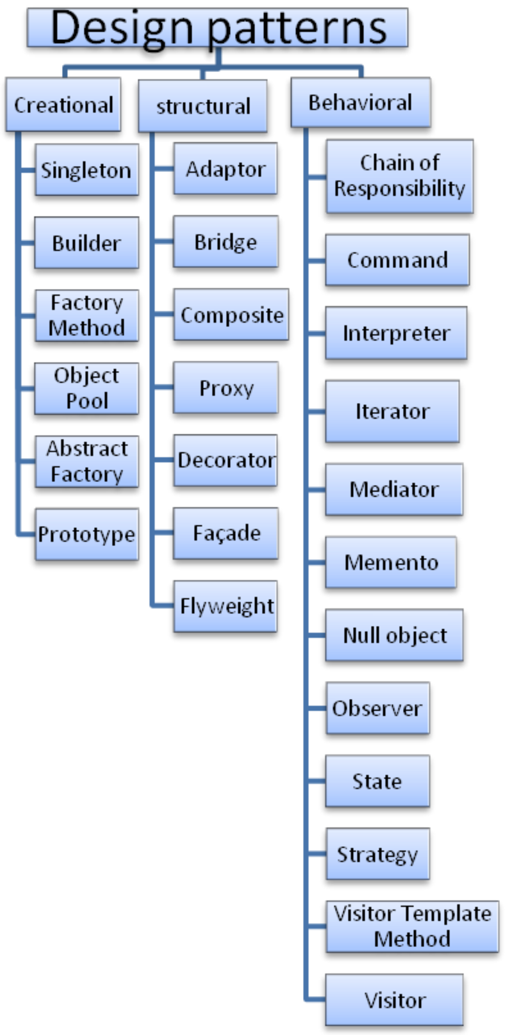


Fig-1: Design Pattern Types

2.6 Design Patterns Subtypes and Definition

Singleton: Define a class that has only one instance and provides a global point of access to it.[1]

Builder: construct a complex object from simple objects using step-by-step approach. [1]

Factory Method: define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate. [1]

Abstract Factory: define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes. [1]

Object Pool: to reuse the object that are expensive to create. [1]

Prototype: Cloning of an existing object instead of creating new one and can also be customized as per the requirement. [1]

Adapter: converts the interface of a class into another interface that a client wants. [1]

Bridge: decouple the functional abstraction from the implementation so that the two can vary independently. [1]

Composite: allow clients to operate in generic manner on objects that may or may not represent a hierarchy of objects. [1]

Proxy: provides the control for accessing the original object. [1]

Decorator: attach flexible additional responsibilities to an object dynamically. [1]

Façade: just provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client.[1]

Flyweight: to reuse already existing similar kind of objects by storing them and create new object when no matching object is found.[1]

Chain of Responsibility: avoid coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request.[1]

Command: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.[1]

Interpreter: to define a representation of grammar of a given language, along with an interpreter that uses this representation to interpret sentences in the language[1]

Iterator: to access the elements of an aggregate object sequentially without exposing its underlying implementation.[1]

Mediator: Allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other. Allows for the actions of each object set to vary independently of one another.[1]

Memento: Memento pattern is used to restore state of an object to a previous state. Memento pattern falls under behavioral pattern category.[1]

Null object: a null object replaces check of NULL object instance. Instead of putting if check for a null value, Null Object reflects a do nothing relationship.[1]

Observer: Multiple observer objects registers with a subject for change notification. When the state of subject changes, it notifies the observers. Objects that listen or watch for change are called observers and the object that is being watched for is called subject.[1]

State: State design pattern provides a mechanism to change behavior of an object based on the object's state. [1]

Strategy: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.[1]

Template Method: Define an algorithm as skeleton of operations and leave the details to be implemented by the child classes. The overall structure and sequence of the algorithm is preserved by the paren t class.[1]

3. OVERVIEW OF RATIONAL RHAPSODY

The Rational Rhapsody product family offers visual development environments tailored for systems engineers and embedded software developers that integrate into the overall development lifecycle—from requirements capture to implementation and system acceptance testing. Based on industry-standard SysML/UML languages and providing visual development of C, C++, Java, C# and Ada languages from model-based designs, Rational Rhapsody solutions promote early validation of design behavior through simulation and execution to identify design errors when they're introduced—and less costly to fix.IBM Rational Rhapsody products address a broad range of system, software and test development challenges, including targeting multicore processors, safety critical development, Android, UTOSAR, UPDM and DDS. Designed for ease of use, early design validation and increased productivity—including integration within the Eclipse platform—these solutions can help embedded and real-time developers to more quickly and easily build and deliver the complex, robust, high-quality products that today's marketplace demands.[3]

Rhapsody in J (I-Logix offers implementations of Rhapsody for generating Java, C++, C, and Ada) provides more than just an environment for developing Java code. It's a visual design environment that enforces a comprehensive methodology for building formally correct system models. Like most UML design tools, it lets you build formal models that prevent ambiguities and illogicalities. Unlike others, it takes those designs and creates Java code that implements the UML design.[2]

Compiling a UML model into Java code works in two parts. First, Rhapsody checks the model's syntax and to a certain extent its semantics. This is possible because the model should be logically consistent; the editor doesn't allow things that are illegal in the modeling language. It can also do some checking for internal consistency, primarily to make sure that the model as represented by one diagram coheres with any related diagrams.[2]
Once the model is verified, the next step is to produce Java code. This works through a mapping between UML constructs and Java classes derived from the Java class library. What results from this process is a software implementation that uses the standard Java object model.[2]
Rhapsody includes an editor for four different diagram types: sequence, object model, use case, and state charts. Rhapsody diagrams are combined into one model, ensuring consistency across the different views. You can inspect the entire model or automatically review it for consistency and syntactical accuracy.[2]

4. CASE STUDIES IMPLEMENTING FOUR DESIGN PATTERNS WITH RATIONAL RHAPSODY IN JAVA

We have implemented all 23 GoF design patterns in GoF. In this section, we describe our experience in implementing four of them: Factory, Proxy, Strategy and Composite Method.

4.1 Factory pattern

The Factory Design Pattern is probably the most used design pattern in modern programming languages like Java, Also known as Virtual Constructor. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

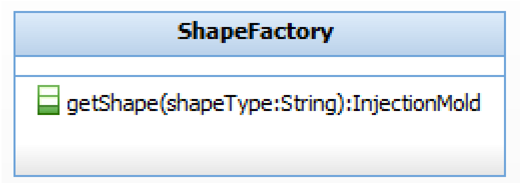
Problem statement:
It is a standard way that objects are created by calling the new keyword. Consider a scenario that there are several cases in the consumer class and we call several new keywords for creating new objects.

If we now have to add new toy mold we have to keep on modifying the client code and further add the new keyword. This makes a dependency on the consumer code and it is difficult to maintain.

Another problem is that the consumer application has to know how many types of concrete classes are available in advance. Later if we have to add another concrete class e.g. toy mold then consumer code has to be changed and recompiled.
Intent:

- We accessing several new keywords can be resolved by using a Factory class.
- Then we using an interface which the concrete classes will implement and the consumer will always point to the interface class rather than the concrete classes. So in this way consumer will be completely unaware of various types of concrete classes which will be required.

Implementation:
Consider a Shape Factory which produces various types of Toy mold like Duck, Car, Bike, etc. The consumers can request for the required types of Toy through the factory. However from consumer's perspective they are completely unaware of who is creating this object. They just know that the Factory is providing them the required Toy.



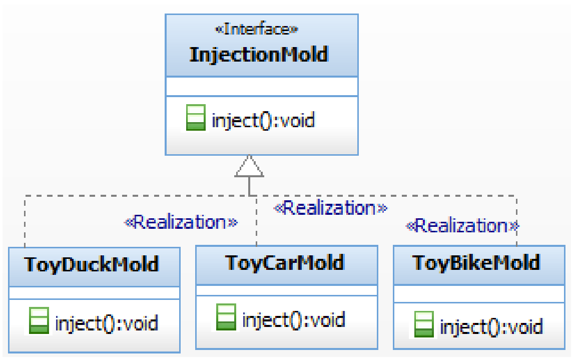
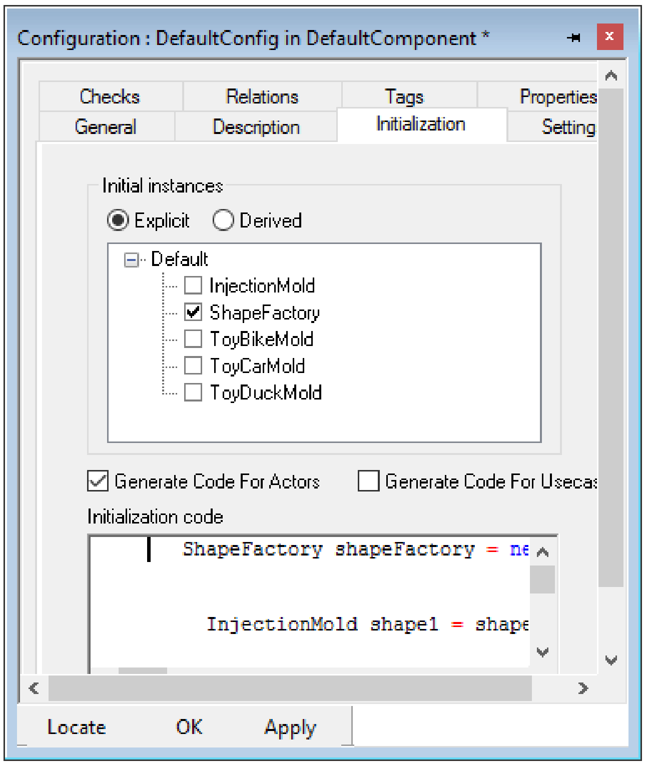


Fig-2: Class Diagram of Toy Factory (Factory Pattern Implementation)

In Rhapsody, with the component, configuration, and environment defined, you can generate code for the Toy Factory model.
Following is the default configuration window and initialization code:



Initial instances
[✓ Explicit] Derived

Default
- [ ] InjectionMold
- [x] ShapeFactory
- [ ] ToyBikeMold
- [ ] ToyCarMold
- [ ] ToyDuckMold

[✓] Generate Code For Actors [ ] Generate Code For Usecas

Initialization code

```
    ShapeFactory shapeFactory = ne
      InjectionMold shape1 = shape
```

Fig-3: Configuration of Toy Factory

Here's the Code:

```
ShapeFactory(main.java)
ShapeFactory shapeFactory = new ShapeFactory();
InjectionMold shape1 = shapeFactory.getShape("DUCK");
shape1.inject();
InjectionMold shape2 = shapeFactory.getShape("CAR");
    shape2.inject();
InjectionMold shape3 = shapeFactory.getShape("BIKE");
    shape3.inject();
```

4.2 Composite pattern

Composite pattern is applied where we need to treat a group of objects in similar way as a single object. This is type of structural pattern as this pattern creates a tree structure of group of objects. It is used to make hierarchical, recursive tree structures of related objects where any element of the structure may be accessed and applied in a standard manner. This includes individual leaf objects and those at any branch of the tree.

Problem statement:
We have Employee with multiple Grades. We are have to prepare a design which can be useful to generate the Employee's to show Employee grade as well as departmental hierarchy after merging all the Employee. So, application should be able to generate:

1) Employee Grade
2) Departmental hierarchy.

If we follow traditional way we have to create multiple objects for retrieving data and it is difficult to get hierarchy structure.

Intent:
A program needs to manipulate a tree data structure and it is necessary to treat both Branches as well as Leaf nodes uniformly.
Make CEO as root of tree and project manager and head marketing are the branches.

Implementation:
Now, Define abstract class Employee which acts as composite pattern actor class. Employee class to add Grade level hierarchy and print all employees. Base component defines the common methods for leaf and composites, we can create a class Employee with a method getSubordinates() to display all Employee name.

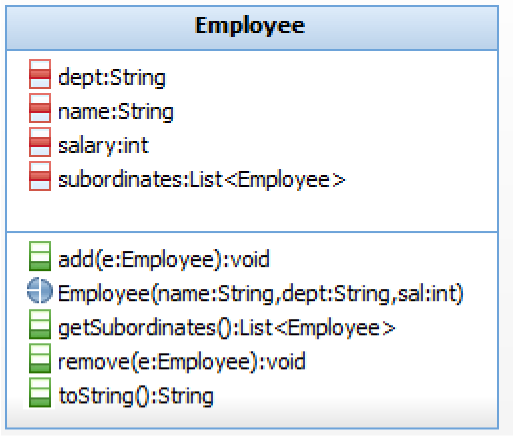
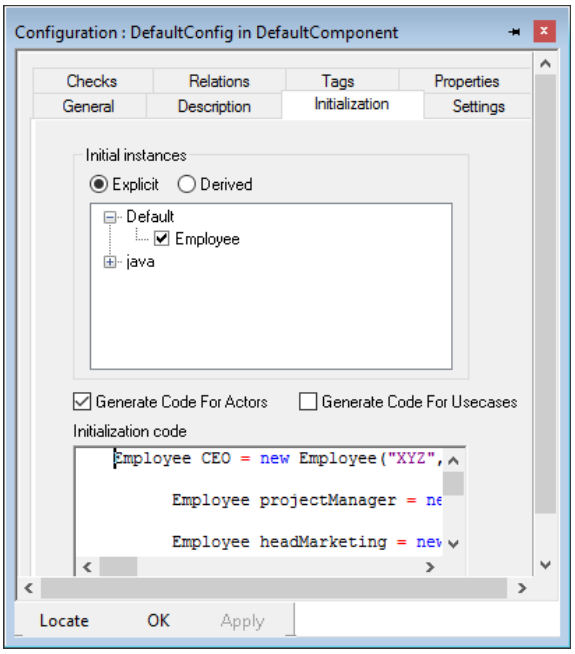

Employee
dept:String
name:String
salary:int
subordinates:List<Employee>
add(e:Employee):void
Employee(name:String,dept:String,sal:int)
getSubordinates():List<Employee>
remove(e:Employee):void
toString():String

Fig-4: Class Diagram of Employee Department(Composite Pattern Implementation)

In Rhapsody, with the component, configuration, and environment defined, you can generate code for the Employee.

Following is the default configuration window and initialization code:



Initial instances

[✓ Explicit] Derived

- Default
  - ✓ Employee
- java

✓ Generate Code For Actors ☐ Generate Code For Usecases

Initialization code

```
Employee CEO = new Employee("XYZ",
        Employee projectManager = ne
        Employee headMarketing = nev
```

Fig-5: Configuration of Employee

Here's the Code:
Employee(main.java)

```
Employee CEO = new Employee("XYZ","CEO", 30000);

Employee projectManager = new Employee("JS","Project Manager", 20000);

Employee headMarketing = new Employee("ARP","Head Marketing", 20000);

Employee clerk1 = new Employee("AD","Marketing", 10000);
Employee clerk2 = new Employee("PL","Marketing", 10000);

Employee TL1 = new Employee("SC","Team Leader", 10000);
Employee TL2 = new Employee("AW","Team Leader", 10000);
Employee TL3 = new Employee("BBG","Team Leader", 10000);

CEO.add(projectManager);
CEO.add(headMarketing);

 projectManager.add(TL1);
 projectManager.add(TL2);
 projectManager.add(TL3);

 headMarketing.add(clerk1);
 headMarketing.add(clerk2);

 //print all employees of the organization
 System.out.println(CEO);

for (Employee headEmployee : CEO.getSubordinates()) {
System.out.println(headEmployee);
for (Employee  employee : headEmployee.getSubordinates()) {
          System.out.println(employee);
        }
     }
```

4.3 Proxy pattern

In proxy pattern, a class represents functionality of another class. This is structural pattern that we create object having original object to interface its functionality to outer world.

Problem statement:
We need to use only a few methods of some heavy objects we'll initialize those objects when we required them entirely. Until that point we can use some light object exposing the same interface as the heavy object. This light objects are called proxies of object and they will initialize that heavy object when they are really needed and by then we'll use some light objects instead.

Intent:
Intent of this pattern is to add a wrapper and delegation to protect the real component from undue complexity.

Implementation:
Let' say we need to withdraw money to make some purchase. The way we will do it is, go to an ATM and get the money, or purchase straight with a cheque. Here, ATMAccess is act as proxy object and BankAccess is Real object.
AcctounAcess is interface where grantAccess() method is declared and call at ATMAccess.

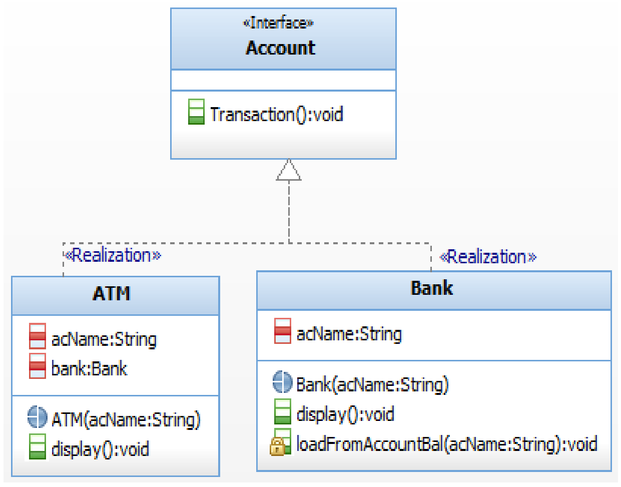
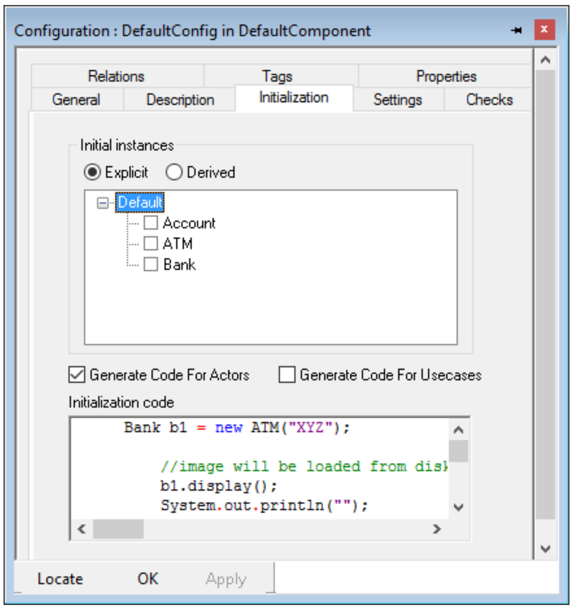


Fig-6*:* Class Diagram of Bank and ATM (Proxy Pattern Implementation)

In Rhapsody, with the component, configuration, and environment defined, you can generate code for the ATM.

Following is the default configuration window and initialization code:



Initial instances
(● Explicit) ○ Derived

- Default
  - ☐ Account
  - ☐ ATM
  - ☐ Bank

☑ Generate Code For Actors ☐ Generate Code For Usecases

Initialization code

```
    Bank b1 = new ATM("XYZ");

        //image will be loaded from dis
        b1.display();
        System.out.println("");
```

Fig-7: Configuration of Bank and ATM

Here's the Code:

```
ATM (main.java)
Bank b1 = new ATM("XYZ");
 b1.display();
 System.out.println("");
 b1.display();
```

4.4 Strategy pattern

The Strategy pattern is known as a behavioural pattern - it's used to manage algorithms, relationships and responsibilities between objects.

Problem statement:
We work on Planes program which show large variety of Planes type (Boeing, Helicopter, Model, etc.) Take up and Take off. At the beginning we used standard OO techniques and created one Planes super class from which all other planes inherit but not all planes subclasses are Take up and Take off.

Intent:

- We realized that inheritance probably wasn't answer because all planes type can't Take up and Take off.
- We could take doTakeup() and doTakeoff() method out Planes superclass and make TakeUp interface doTakeup() method. That way, only the Planes that are supposed to TakeUp and TakeOff will be implement that interface and have a doTakeup() and doTakeoff() method.

Implementation:
So we know using inheritance hasn't worked out very well, since the planes behavior keep changing across the subclasses, and it's not appropriate for all subclasses to have those behaviors.[6]

As far as we can tell ,other than the problems with doTakeup() and doTakeoff(), the planes class is working well and there are no other parts of it that appear to vary or change frequently. so, other than a few slight changes we're going to pretty much leave the Planes class alone.

Now, we are going to create two sets of classes one for Takeup and one for Takeoff .Each set of classes will hold on implementation of their respective behavior. For instance ,We might have one class that implements Takeup, another that implements Takeoff , and another that implements steady.

Integrating the plane behavior :

Planes feel now delegate its Takeup and Takeoff behavior, instead of using doTake() and doTakeoff() Methods. Defined in the Plane class:

1. First we'll add two instance variables to the planes class called Tup and toff, that are declare as the interface type each plane object will set these variables polymorphically to refrence the specific behavior type it will like at run time.The behavior variables are declare as the Takeup and Takeoff interface type .
2. doTakeup and doTakeoff methods are replaced by executeStrategy() and executeStrategyNoTake().
3. Now we implement executeStrategy() and executeStrategyNoTake().

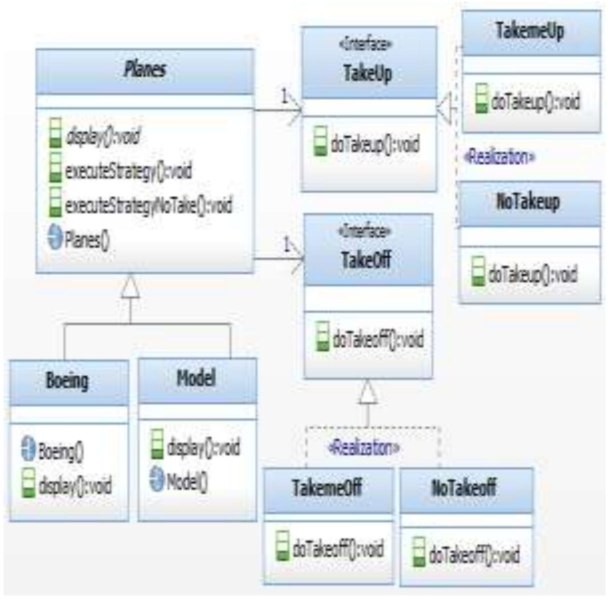
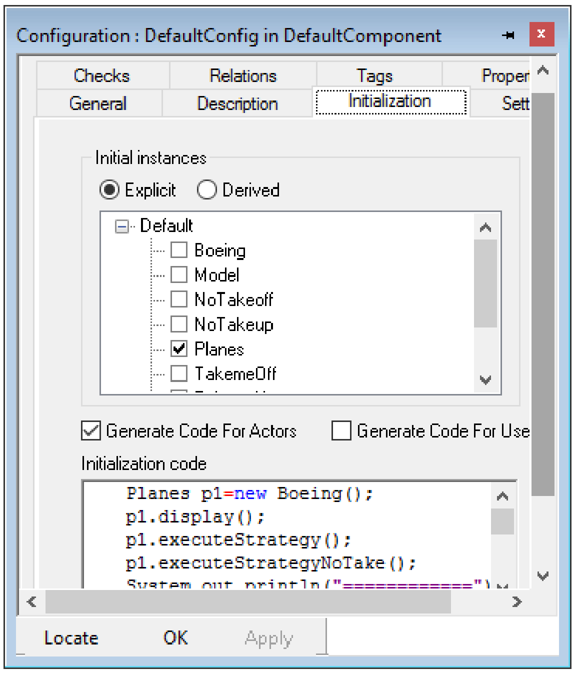


Fig-8 Class Diagram of Planes(Strategy Pattern Implementation)

In Rhapsody, with the component, configuration, and environment defined, you can generate code for the Planes.
Following is the default configuration window and initialization code:



Configuration : DefaultConfig in DefaultComponent

Initial instances

[✓ Explicit] Derived

- Default
  - ☐ Boeing
  - ☐ Model
  - ☐ NoTakeoff
  - ☐ NoTakeup
  - ☑ Planes
  - ☐ TakemeOff

☑ Generate Code For Actors ☐ Generate Code For Use

Initialization code

```
Planes p1=new Boeing();
p1.display();
p1.executeStrategy();
p1.executeStrategyNoTake();
```

Fig-9: Configuration of Planes

Here's the Code:
Planes(main.java):

```
Planes p1=new Boeing();
p1.display();
p1.executeStrategy();
p1.executeStrategyNoTake();
System.out.println("============");
Planes p2=new Model();
p2.display();
p2.executeStrategy();
p2.executeStrategyNoTake();
```

5. Discuss our experience using Java and Rational Rhapsody

In this section, we reflect on what we have learnt about design patterns in Java with Rational Rhapsody: all four type which discuss in paper that how existing patterns are implemented in Java, and what new, Java-specific patterns we may have implemented in Rational Rhapsody.

Patterns are a valuable tool for helping to design software that is easy to maintain and extend.
This paper validates the design patterns using the tool IBM Rational Rhapsody. In Rational Rhapsody we create a class and set their design fundamentals and generate the code for the design patterns and validate the design patterns using 'make and build' option. Thus the functionality of design patterns can be verified during the design phase and reduce the number of anomalies in software.

This validation of design patterns for functional correctness was not possible in static UML diagrams. Rational Rhapsody also enables the animation of sequence diagram and statecharts by generating events to check the flow and behavior of the component.

6. Future work

We have attempted to analyzed java design patterns using IBM Rational Rhapsody. The Gang of Four patterns are general-purpose programming patterns, and so our analysis is of java as a general-purpose language. Java is specifically targeted at the development of Software.
Our most immediate future work is thus to Analysis design patterns for ADA using IBM Rational Rhapsody.

7. Conclusion

In this paper we have analysis Java with IBM Rational Rhapsody, and analyzed it using design Patterns. Our implementations of design patterns have highlighted Java-specific features. Rational Rhapsody allows for an easy implementation of all design patterns.

This paper illustrates an approach for analysis design pattern from Rational Rhapsody. This approach improves the quality of Software development. The executable design pattern templates help an developer when building software architectures and also provide the foundation for performing design time validation on the software architecture produced using this approach. The developers also can use these design patterns to form the core base for building the software architecture of any other system in this domain.

8. References

[1] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1996.

[2] http://archive.oreilly.com/pub/a/onjava/2001/01/25/uml_rhapsody.html.

[3] http://adneurope.com/fileadmin/user_upload/Partenaire/IBM/Rhapsody.PDF

[4] G. Rossi,D.Schwabe andF.Lyardet,"Improving Web Information Systemswith Design Patterns". In Proc. of the 8th International World Wide Web Conference ,Toronto (CA), May 1999, Elsevier Science, 1999, pp. 589 -600.

[5] D.C. Schmidt, R. E. Johnson and M. Fayad. "Software Patterns".Communications of the ACM, Special Issue on Patterns and Pattern Languages, Vol. 39, No. 10

[6] Head First Design Patterns By Eric Freeman, Elisabeth Freeman, Bert Bates, Kathy Sierra.

[7] https://sourcemaking.com/design_patterns/composite

BIOGRAPHIES

	Bhushan Bharat Ghatkar Student, MCA SEM VI, DES's NMITD, Dadar(Mumbai). Email ID: bhushanghatkar15@gmail.com
	Amruta Ramchandra Pandhare Student, MCA SEM VI, DES's NMITD, Dadar(Mumbai). Email ID: amupandhare@gmail.com