# Pattern Hybridization: Breeding New Designs Out of Pattern Interactions

D Janaki Ram, P Jithendra Kumar Reddy and M S Rajasree
Distributed and Object Systems Lab
Department of Computer Science and Engineering
Indian Institute of Technology Madras, Chennai - 600036
e-mail: {djram,jithendra,rajasree}@cs.iitm.ernet.in

## Abstract

*Class or object interactions form the basis of object-oriented design. However, design pattern interaction can be viewed as a higher level of abstraction for system design. The typical interactions among the patterns are a pattern uses another pattern to solve one of its sub problem, and a pattern combines with another pattern for completeness. This paper proposes a mechanism called pattern hybridization for breeding new patterns from the pattern interactions which solve more specialized problems than the original patterns do. Rules for generating hybrid patterns are also mentioned in the paper. This paper also views design pattern interactions for system design.*

## Introduction

Object-oriented system design focuses on building design solutions by identifying classes and their interactions. Design patterns have emerged as a way of capturing and reusing class or object interactions for recurring design problems. Thinking object-oriented software in terms of pattern is one of the dominant forms in recent times.

As the number of patterns increased, different people [GHJV], [Zim94], [FR95], [Mag] classified patterns in different ways. Identification and composition of patterns are the key steps involved in pattern oriented software development. [JAG97] proposed a technique called Pattern Oriented Technique (POT) for identifying patterns and [JAG$^+$00] proposed a way to quantify pattern oriented designs.

Object-oriented technology is centered around the class or object interactions. Similarly, pattern-oriented technology can be centered around pattern interactions. A closer look at pattern interactions helps in making pattern oriented technology a matured discipline. [Zim94], [JJR03a] describes about two types of pattern interactions called *uses* and *combines*. [JJR03b] explains quantitative estimations of pattern oriented designs considering pattern interactions. This paper proposes a new concept called *pattern hybridization* which is also based on pattern interactions.

Designing complex systems is always a tedious task. Pattern hybridization is an attempt to simplify this task. This technique tries to synthesize hybrid patterns from the interactions *viz uses* and *combines* among different classes of patterns. These hybrid patterns raise the granularity and abstraction of patterns. Hybridization can be applied recursively to raise the level of abstraction of patterns to application level. The paper provides rules that should be applied while generating hybrid patterns.

This paper is organized as follows. Next section gives a closer look at the problem being addressed in the paper and also explains some of the prior work done in this area. Then, we elaborate the pattern hybridization technique with the help of a Lexi case study. Finally we conclude the paper, giving pointers to future work.

## The Problem

We use Lexi editor [GHJV] in explaining the problem.

Consider the designing of a Lexi editor. Lexi should allow editing of documents comprising text, graphics and charts, support multiple formatting algorithms for text layout and provide a graphical user interface. The design problems involved in Lexi are:

- Document structure
- Formatting
- Embellishing the user interface
- Multiple look and feel standards
- User operations
- Spelling and hyphenation

Using the traditional way of designing, each of the design problems can be treated as the subject of interest. Exhaustive list of objects need to be identified for each of the subject. Next step is to find interactions and relationships among the meaningful objects. Then identify attributes and services for each of the object. This process is very systematic and efficient, but industry requires to come up with good designs in less time. Though we follow the traditional process, one is not guaranteed of arriving at a good design. Here, patterns come into picture as good reusable designs. However, pattern oriented technology is not matured like any other engineering technology. Reason behind this is presence of large number of patterns, difficulty in identifying right patterns and composing them to form final design of the system.

Pattern hybridization is a new approach to build systems out of patterns and their interactions. Figure 1 shows levels of interactions. Patterns are a result of class or object interactions. Higher level systems or architectures can be looked in terms of interactions among patterns rather than looking in terms of interactions at the elementary class or object level. This helps in more meaningful pattern oriented

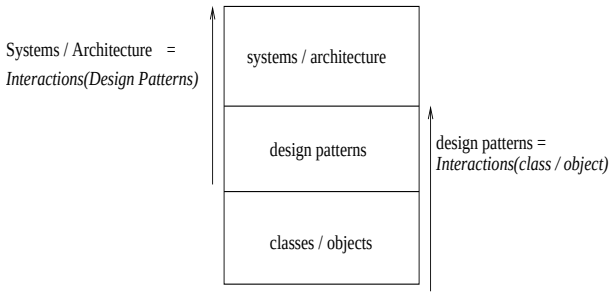


Figure 1: Levels of Interaction

software development. This approach also hatches hybrid patterns out of pattern interactions, which attempt to solve more specialized design problems than that of the original patterns. Rest of the paper is focused on these issues.

## Prior Art

This section gives a brief overview of the related works done. Hybridization is a mechanism to yield new designs from the interactions between patterns of different classes. There exists several pattern classification theories in the literature. Classification criteria should reflect what a pattern does. Following are some of the pattern classification schemes.

- **Gamma et.al:** Patterns [GHJV] are classified into *creational, structural and behavioral*. Patterns belonging to *creational* category deals with the process of class or object creation. *Structural* patterns concern with the composition of classes or objects. *Behavioral* patterns characterize the ways in which classes or objects interact and share responsibilities.

- **Buschman et.al:** According to Buschman et.al [FR95], patterns are classified based on criteria: *functionality and structural principles*. *Functionalities* can be *creation, communication, access, organizing the computation of complex tasks*. Criterion of structural principles allows to distinguish among *abstraction, encapsulation, separation of concerns, coupling and cohesion*.

- **Zimmer:** Zimmer [Zim94] classifies the relationships among Gamma et. al [GHJV] patterns. He divides the relationships into Pattern X *uses* Pattern Y in its solution, Pattern X *can be combined with* pattern Y, Pattern X *is similar to* Pattern Y. By arranging the patterns along these relationships, Zimmer identified three different layers:

  – Basic design patterns and techniques.
  – Design patterns for typical software problems.
  – Design patterns specific to an application domain.

  In this paper, Zimmer relationships *uses, combines* are treated as typical interactions between patterns for generating hybrid patterns.

- **Pree:** Pree [Wol95] classified the patterns based on their structures.

  – Patterns based on inheritance and interactions.
  – Patterns for structuring object-oriented systems.
  – Patterns based on abstract coupling.
  – Patterns relying on recursive structures.
  – Patterns related to MVC framework.

- **Magnus Kardell**: Magnus [Mag] classified the patterns based on four criteria :*purpose, applies to, scope, time to apply.*

  – Purpose: This reflects to what purpose does a pattern has. It is divided into seven categories: *functionality, interface, state, access, communication, physicality, instantiation*
  – Applies to: It reflects to what entity the patterns should be applied to.
    *Applies to* defines three categories: *object, object-families, related object-families*.
  – Scope: This indicates whether a pattern is applied statically or dynamically.
  – Time to apply: This categorizes patterns used at building time and patterns used at reusing time.

  In this paper, Magnus classification scheme is mainly used for explaining the concept of hybridization.

Besides these classification theory of patterns, people also worked on pattern oriented software development. [She] proposed a methodology called Pattern Oriented Analysis and Design(POAD) which follows structural composition approach to glue patterns at high level design. Patterns are treated as design components with interfaces. In the present paper, we explored in a new direction of pattern oriented software development by composing different patterns semantically for raising design granularity towards the overall system design.

[Dir97] proposes the concept of composite design pattern to extend the idea of pattern from single problem solutions to object-oriented frameworks. A composite design pattern is a pattern resulted by the composition of further patterns, the integration of which shows a synergy that makes the composition more than just the sum of its parts. The present paper attempts to systematically generate new patterns from the pattern interactions, whose intent is to solve a more specialized problem than that of the original patterns. Also, the mechanism proposed in this paper helps in designing complex systems based on patterns and their interactions.

[PDJ99] describes an approach to component-based software engineering based on a formal description of design patterns. Design patterns are represented declaratively and are treated as architectural building block design components in the development process. These design component descriptions can be instantiated, adapted, assembled and maintained. The present paper concentrates on the pattern interactions for building architectures.

# Pattern Hybridization

**Definition:** *Pattern hybridization is a mechanism to compose different patterns for generating a hybrid pattern whose intent is to solve a high level design problem in a generic context.*

## Pattern Tuple

To explain the technique, each pattern is represented using three tuple which is based on [Mag] pattern classification criteria.
$Pattern[Appliesto, Purpose, UniqueIntent]$

- **Applies to:** This represents the entity to which the pattern is applied to. Typical entities where a pattern can be applied are:
  - *object:* Pattern is applied on an object. For example, Singleton pattern [GHJV] is applied on an object to provide only one instance of the object.
  - *object-families:* Pattern is applied to a set of objects, where the objects are not necessarily related by inheritance. For example, Mediator pattern provides communication among a set of objects which are not necessarily related by inheritance.
  - *related-object families:* Pattern is applied to a set of objects where most of the objects are related by inheritance. For example , Composite pattern provides a common interface to a set of related objects.
- **Purpose:** This represents the purpose for applying the pattern to the entity. Brief description of different purposes are given below.
  - *interface:* This tells that the pattern is applied for providing interface characteristic to the pattern. For example, Adapter pattern is applied for working with incompatible interfaces.
  - *functionality:* This tells that the pattern is used for providing some kind of functionality to the entity. For example, Decorator pattern is used to provide the functionality of attaching dynamic responsibilities to an entity.
  - *state:* This tells that the pattern is concerned with the state of an entity. For example, Memento pattern is used for restoring the previous state.
  - *communication:* This tells that the pattern is used for the purpose of communication among the entities. For example, Mediator is used for the communication among different entities.
  - *access:* This tells that the pattern deals with entity access. For example, Proxy pattern provides restricted access to the entity.
  - *instantiation:* This tells that the pattern deals with instantiation of an entity. For example, Prototype pattern allows instantiation of an entity by cloning.
- **Unique Intent:** This gives the unique behavior, a pattern is providing.

Consider the tuple, Singleton[Object, instantiation, class needs to be instantiated in mutually exclusive manner] gives the description of the Singleton pattern. This way of representing the pattern in terms of tuple is needed for semantical hybridization, which will be clear from the following sections.

## Hybridization by Uses Interaction

This section explains about *uses* interaction between patterns and hybridization achieved by the same.

A pattern uses another pattern to solve one of its sub problem. *Uses* relationship between the patterns can be seen as semantic interaction which helps in generating hybrid patterns. Semantic in the sense, a pattern *uses* another pattern based on the following rules.

- One pattern can use another pattern only if the purposes of both the patterns are same. For example, prototype can use singleton since both are used for instantiation purpose.
- Pattern1[RO,P,A] uses Pattern2[O,P,B] generates hybridpattern[RO, P, A+$\Delta B$] where RO - Related Object families, O - Object, P - Purpose of Pattern1 and Pattern2, A and B denotes unique intent of Pattern1 and Pattern2 respectively. Hybrid pattern generated is applicable to related object families whose purpose remains same as that of original two patterns. The unique intent of the hybrid pattern is some $\Delta B$ of Pattern2 added to the intent of Pattern1. $\Delta B$ implies that behavior of Pattern2 gets added to the behavior of Pattern1 to solve a much higher problem than that of Pattern1 addresses.
- Pattern1[RO,P,A ] uses Pattern2[OF,P,B] generates hybridpattern[RO, P,A+$\Delta B$] where OF - Object Families. Hybrid pattern generated is applicable to related object families whose purpose remains same as that of original two patterns.
- Pattern1[OF,P,A ] uses Pattern2[O,P,B] generates hybridpattern[OF, P,A+$\Delta B$]. Hybrid pattern generated is applicable to object families.
- Pattern1[OF,P,A ] uses Pattern2[RO,P,B] generates hybridpattern[OF, P,A+$\Delta B$]. Hybrid pattern generated is applicable to object families.
- Pattern1[O,P,A ] uses Pattern2[O,P,B] generates hybridpattern[O, P,A+$\Delta B$]. Hybrid pattern generated is applicable to an object.

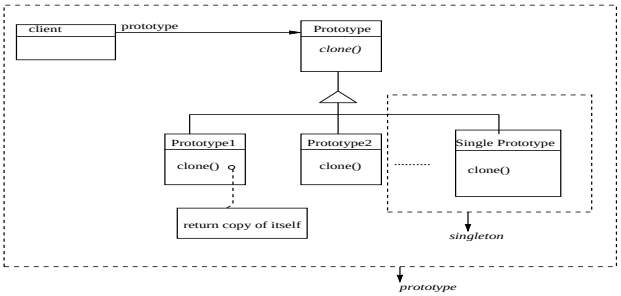


Figure 2: Uses Interaction by Subclass Sharing

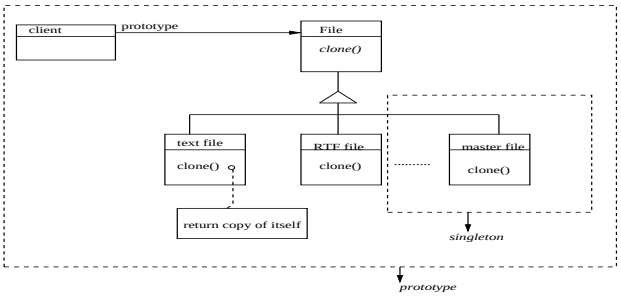


Figure 3: Hybrid of Prototype and Singleton Applied to File System Case

- Pattern applied for an object cannot use a pattern which is applicable for related object families or object families.

*Uses* interaction is conceptual. It should be brought down to the design level. One can generate the hybrid pattern out of uses interaction by any of the under mentioned ways.

**1.** Two patterns can have *uses* relationship by means of subclass interaction. For example, consider Prototype[O, instantiation, cloning] uses Singleton[O,Instantiation,mutually exclusive] which generates hybridPrS[O,Instantiation,cloning done in mutually exclusive manner]. Here, Singleton pattern is implemented in the subclass of Prototype pattern. Generalized hybrid of Prototype and Singleton is shown in figure 2.

Generalized hybrid of Prototype and Singleton is applied to file system case study as shown in figure 3. Here, the requirement is that master file needs to be cloned in mutually exclusive manner, which is solved by the hybrid pattern.

**2.** *Uses* relationship between two patterns can also be implemented by sharing a hierarchy. For example, consider Proxy[O, Access, surrogate to real subject] uses Iterator[O, Access, traverse aggregate] result in hybridPIt[O, Access,surrogate able to traverse its real aggregate]. Here, both the Proxy and Iterator patterns share the same hierarchy as shown in figure 4. This hybrid pattern allows the place holder object to traverse it's respective aggregate object.

The hybrid pattern hybridPIt is applied to a case where privileges proxy has to traverse it's respective privileges aggregate list as shown in figure 5.

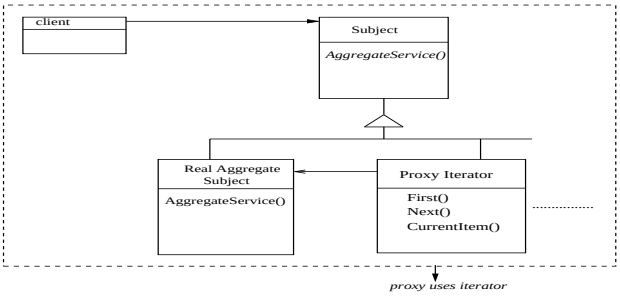


Figure 4: Uses Interaction by Sharing of Hierarchy

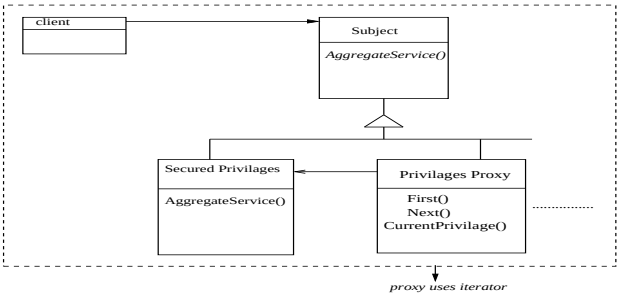


Figure 5: Hybrid of Proxy and Iterator Applied to Privileges Case

**3.** *Uses* relationship between two patterns can also be implemented by just a link between two patterns, both the patterns maintaining their structures independently. For example, consider Decorator[O, Functionality, attach responsibilities dynamically] uses Strategy[O, Functionality, varying algorithms] generates hybridDeSr[O,Functionality, attach responsibilities dynamically by following appropriate algorithm]. Figure 6 shows the hybrid of Decorator and strategy which are connected by just a simple association link, preserving their original structures.

This hybridDeSr pattern is applied to a case where visual component gets additional behavior of border dynamically, and the drawing of border may vary different algorithms depending upon the client's interest. This is shown in figure 7.

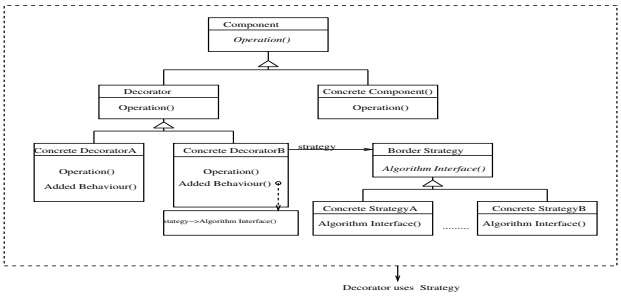


Figure 6: Uses Interaction by Simple Link

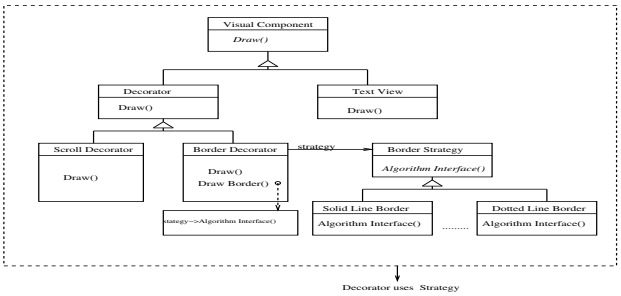


Figure 7: Hybrid of Decorator and Strategy Applied to Border Decorator

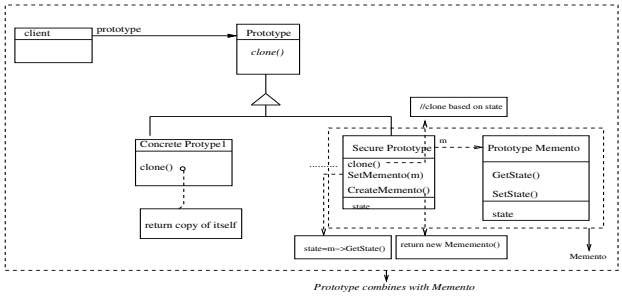


Figure 8: Combines Interaction by Subclass Sharing

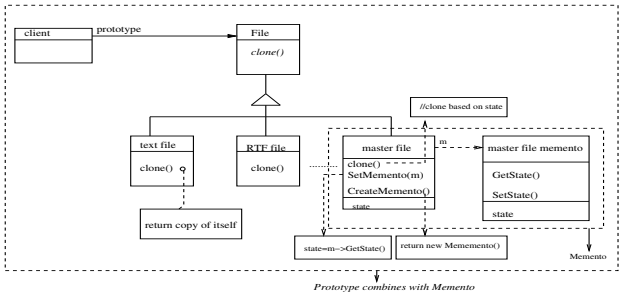


Figure 9: Hybrid of Prototype and Memento Applied to File Cloning Case

## Hybridization by Combines Interaction

This section explains about *combines* interaction between patterns and hybridization achieved by the same.

A pattern combines with another pattern to achieve mixed behavior. Just as *uses* relationship, *combines* relationship between the patterns can be seen as semantic interaction, which helps in generating hybrid patterns. Unlike *uses* interaction, *combines* interaction between two patterns results in a hybrid pattern which serves a combination of two different purposes. A pattern combines with another pattern based on the following rules.

- One pattern can combine with another pattern, if the purposes of both the patterns are different. For example, Factory Method combines Iterator, former is used for instantiation purpose and the latter for access purpose.
- Pattern1[RO,P1,A] combines Pattern2[O,P2,B] generates hybridpattern[RO, P,$\Delta A+\Delta B$] where RO - Related Object families, O - Object, P1 - Purpose of Pattern1, P2 - Purposes of Pattern2, P - new purpose as a result of combination of P1 and P2, A and B denotes unique intent of Pattern1 and Pattern2 respectively. Hybrid pattern generated is applicable to related object families which serves a new purpose as a result of the combination of both the purposes P1 and P2. The unique intent of the hybrid pattern is the mixture of both intents of Pattern1 and Pattern2. $\Delta A + \Delta B$ denotes the mixture of both the intents.
- Pattern1[RO,P1,A ] combines Pattern2[OF,P2,B] generates hybridpattern[RO,P,$\Delta A + \Delta B$] where OF - Object Families. Hybrid pattern generated is applicable to related object families which serves a purpose P. The unique intent of the hybrid pattern is again $\Delta A + \Delta B$, mixture of both the intents.
- Pattern1[OF,P1,A ] combines Pattern2[O,P2,B] generates hybridpattern[OF, P,$\Delta A + \Delta B$]. Hybrid pattern generated is applicable to object families.
- Pattern1[OF,P1,A ] combines Pattern2[RO,P2,B] generates hybridpattern[OF,P,$\Delta A + \Delta B$]. Hybrid pattern generated is applicable to object families.
- Pattern1[O,P1,A ] combines Pattern2[O,P2,B] generates hybridpattern[O, P,$\Delta A + \Delta B$]. Hybrid pattern generated is applicable to object.

*Combines* interaction is conceptual. It should be brought down to the design level. Same as *uses* interaction, one can generate the hybrid pattern out of *combines* interaction by any of the undermentioned ways.

**1.** Two patterns can combine by means of subclass interaction. For example, consider Prototype[O, instantiation, cloning] combines Memento[O, State, store state] which generates hybridPrM[O, state based instantiation, instantiation based on state condition where the state is stored frequently]. Here, one of the classes of Memento pattern is a subclass of Prototype pattern. Generalized hybrid of Prototype and Memento is shown in figure 8.

Generalized hybrid of Prototype and Memento is applied to file system case study as shown in figure 9. Here, the requirement is that master file needs to be cloned based on its previous state condition, which is solved by the hybrid pattern.

**2.** *Combines* relationship between two patterns can also be implemented by sharing a hierarchy. For example, consider FactoryMethod[O, Instantiation, instantiation decision is left to sub classes] combines Iterator[O, Access, traverse aggregate] result in hybridFMIt[O, Iteration after Instantiation, Iterate the aggregate object instantiated at sub class]. Here, both the Factory Method and Iterator pattens share the same hierarchy as shown in figure 10. This hybrid pat-

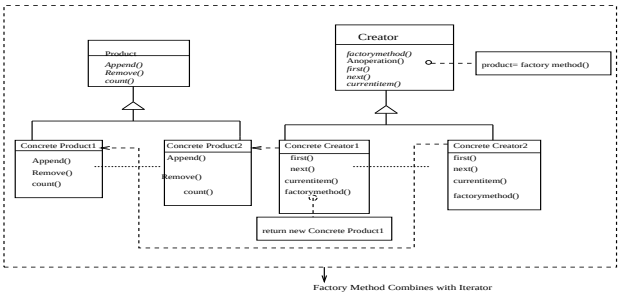


Figure 10: Combines Interaction by Sharing of Hierarchy

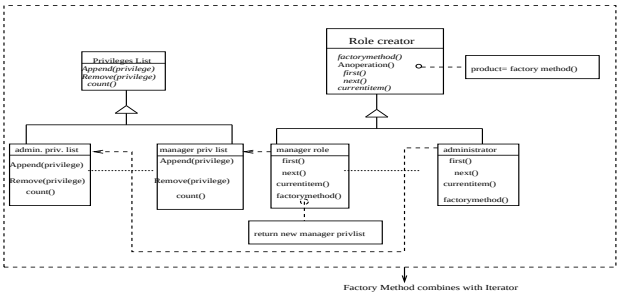


Figure 11: Hybrid of Factory Method and Iterator Applied to Role Case

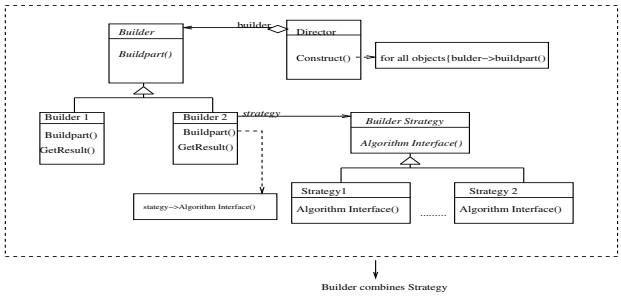


Figure 12: Combines Interaction by Simple Link

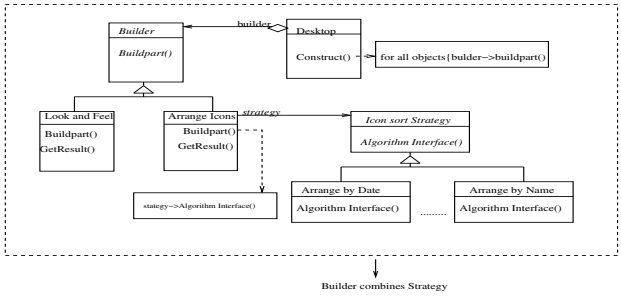


Figure 13: Hybrid of Builder and Strategy Applied to Desktop Case

tern allows the sub class not only to instantiate the aggregate object but also to traverse it's respective aggregate object.

The hybrid pattern hybridFMIt is applied to a case where instantiated privileges list needs to be traversed as shown in figure 11. Here, each role instantiate it's corresponding privileges list using Factory Method. Iterator pattern allows each role to traverse the instantiated privileges list.

**3.** *Combines* relationship between two patterns can also be implemented by just a link between two patterns, both the patterns maintaining their structures independently. For example, consider Builder[O, Instantiation, Complex object is constructed from its parts] combines Strategy[O,Functionality, varying algorithms] generates hybridBuSr[O, Algorithmic dependent Instantiation, Parts which build the complex object are built by varying algorithms ]. Figure 12 shows the hybrid of Builder and Strategy which are connected by just a simple association link, preserving their original structures.

This hybridBuSr pattern is applied to a case where complex desktop is built from its parts. Look and feel of desktop, arranging icons on the desktop are some of the functional parts which are required to build the desktop. Arranging of icons may vary its strategy, arrange by date or by name, depending upon the client's request. This is shown in figure 13.

## Hybridization Tree

Hybridization tree, in figure 14 shows different patterns arranged in the order of their purpose and applies to criteria.

*Uses* and *Combines* interaction among different patterns is also shown in the tree. The hybrid pattern generated by each of this interaction is also represented in the tree. However, for the sake of clarity, only one hybrid pattern is represented in the figure 14. Hybrid pattern generated by Combines interaction between Iterator and Memento is shown in the figure 14 in the tuple form. However, most of the hybrid patterns with their intent is shown in Table 1. This hybridization tree is not frozen. The interaction links gets added, if we find a meaningful interaction between the patterns in some domain. This meaningful interaction should be documented as a hybrid pattern, which helps in solving a bigger design problem in various domains.

### Design Systems

One can design their systems easily with the help of this hybridization tree. It is assumed that the design is completely based on patterns. Lexi editor design is taken as a case study to explain the importance of hybridization in designing complex systems. The following steps illustrate the technique.

- Clearly understand the design problems. The design problems of the Lexi are already mentioned in this paper. A detailed description for each of the Lexi design problems is given below
  - Document Interface: Lexi should support hierarchical document interface.

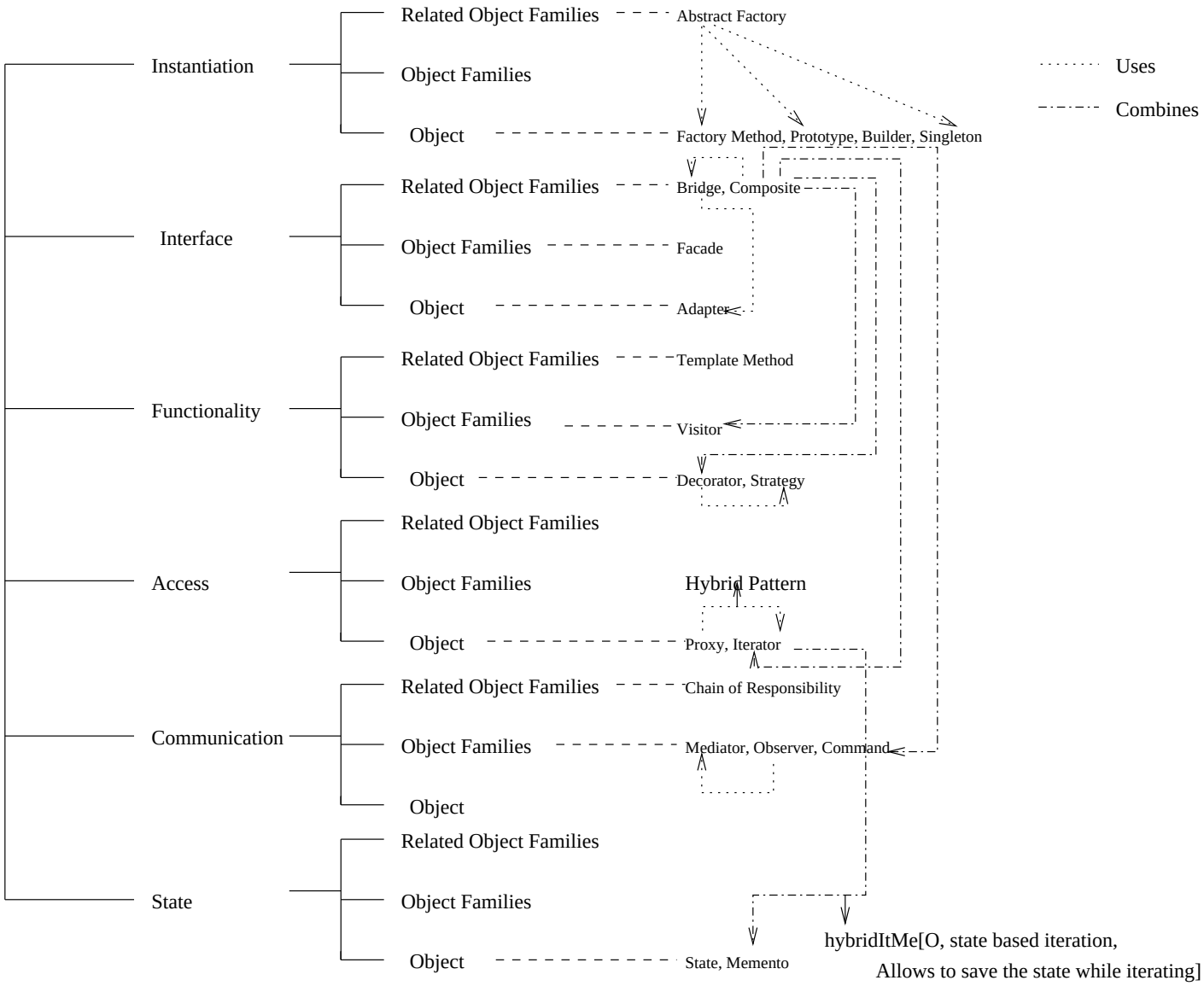


Figure 14: Hybridization Tree

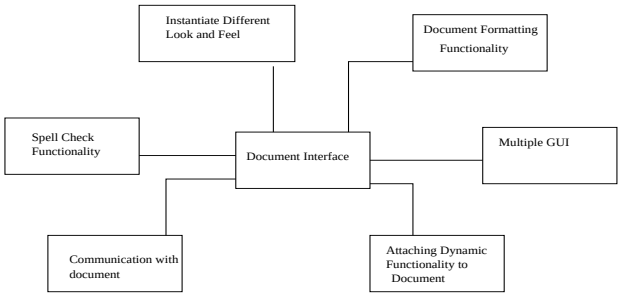


Figure 15: Problem Interactions for the Lexi

  - Formatting: There should be a way to format the content of the document depending upon the client's interest.
  - Embellishing the User Interface: The document user interface should be made attractive by adding various functionalities dynamically.
  - Multiple Look and Feel standards: The document interface should allow instantiating different factory windows.
  - User Operations: User should be able to communicate his requirements to the document editor, so that it can handle the appropriate operations.
  - Spelling and Hyphenation: The document editor should flexibly support the spelling and hyphenation functionalities for the content.

- Pictorially represent the various design problems. Figure 15 shows the problem interactions for the Lexi. The picture reveals that the document interface is interacting with several other problems.

- Identify the key problems which are interacting with several other problems. In this case, document interface design problem is interacting with several other design problems. This way of identifying the key problems helps us to concentrate on their design first.

- For each of the key problem, identify the purpose. This helps to narrow down our search for a pattern to one of the subtree in the hybridization tree. In this case-study, the key problem, document interface is dealing with the problem of providing hierarchical interface. So, our search for a pattern is confined to "Interface" subtree in the hybridization tree.

- Further, based on the applicability of the problem, one can move down to either of RO, OF or O trees. In this case, document interface problem is dealing with the related object families. So, in the RO tree of Interface, the pattern that best matches the hierarchical interfaces is the Composite pattern.

- Once the patterns for the key problems are identified, then try to solve the problems interacting with the key problem by raising the problem level. In the example, formatting functionality for the document can be seen as higher problem. Since, we already identified Composite pattern for solving document interface problem, try to see whether there is any hybrid pattern, Composite pattern being one of it's constituents, which solves the higher problem. The search for the hybrid pattern can be narrowed down, by knowing what exactly the interacting problem is doing. In this case, it is trying to provide formatting functionality to the document. So, the other constituent of the hybrid pattern will be from "functionality" subtree in the hybridization tree. Now from the table 1, we can see that hybrid pattern "Composite *combined with* Strategy" will solve the problem of providing formatting functionality to hierarchical document interface. In the same way, each of the problems interacting with the document interface problem is seen as a bigger problem and are searched for the corresponding hybrid patterns. Hybrid patterns which solve these bigger problems need to be found using Composite pattern as the key, just as explained above. Following are the design problems and the corresponding hybrid patterns, which solves the problem:

  - Document interface made attractive by adding functionalities dynamically: Composite *combined with* Decorator, a hybrid of interface and functionality trees.
  - Document interface allowing the instantiation of different factory windows: Composite *combined with* Abstract Factory, a hybrid of interface and instantiation trees.
  - User should communicate with document : Composite *combined with* Command, a hybrid of interface and communication tree.
  - Document should flexibly support spelling and hyphenation functionalities: Composite *combined with* Visitor, a hybrid of interface and functionality trees.

The above procedure is a systematic way of designing the systems with the help of hybrid patterns. However, one can also design the systems, by just mapping the domain requirements with the intent of the documented hybrid patterns, just as we do for normal patterns. The process of hybridization can be nested. For example, in the table 1, we have Abstract Factory *uses* Prototype and Prototype *uses* Singleton. Then, we can have one more level of hybridization Abstract Factory *uses* (Prototype *uses* Singleton), which addresses a much higher design problem as shown in table 1.

## Concluding Remarks and Future Work

Pattern hybridization is an approach to breed new designs from the existing good designs for solving much higher prob-

Table 1: Hybrid Patterns

| **Hybrid Pattern** | **Intent** |
|---|---|
| Abstract Factory uses Factory Method | Sub classes create Factory Products |
| Abstract Factory uses Prototype | configure factory products dynamically |
| Abstract Factory uses Singleton | Factory product should be a single instance |
| Prototype uses Singleton | Cloning should be mutually exclusive |
| Bridge uses Adapter | Abstraction and implementation can have incompatible interfaces |
| Proxy uses Iterator | surrogate object needs to traverse its respective real object |
| Decorator uses Strategy | Dynamic functionalities attached to the object may vary their strategies |
| Template Method uses strategy | Steps redefined at subclass based on some strategy |
| Observer uses Mediator | notify the observers by a mediator to manage complex dependencies |
| State uses Memento | object alter its behavior based on the state stored in memento |
| Prototype combined with Memento | cloning of the object is done based on state stored in memento |
| Factory Method combined with Iterator | Allows to traverse aggregate factory products at subclass level |
| Builder combined with Strategy | Parts are built based on some strategy |
| Builder combined with Composite | Needs to build a composite object from its parts |
| Composite combined with Decorator | Attach dynamic functionalities to the composites |
| Composite combined with Iterator | Allow traversing the composites |
| Command combined with Composite | Command is a macro composite command |
| Composite combined with Strategy | Composites functionality may vary from the clients that use them |
| Composite combined with Visitor | Operations are added to the composites flexibly |
| Iterator combined with Memento | State of iteration needs to be stored |
| Chain of Responsibility combined with Composite | Pass the request across the chain of composites for handling |
| Facade combined with Singleton | Unified interface to a set of interfaces is a singleton |
| Visitor combined with Iterator | Defines traversal operation on the elements of an object structure |
| Abstract Factory uses (Prototype uses Singleton) | Factory product is dynamically configured mutually exclusively |
| Proxy uses (Iterator uses Memento) | State based traversal of the real subject by the surrogate |

lems. These hybrids are also capable of solving design problems in various domains. The list of hybrid patterns given in this paper is not complete. The list can be extended, if we find a meaningful pattern interaction which is applicable in different contexts. The hybridization proposed in this paper is based on [Mag] pattern classification theory. It is always possible that a new pattern may not fall into any of the mentioned purposes. But the technique remains the same, as it is to explore for a new design from the interactions between different patterns.

As a part of future work, we are trying to see whether a design language can be formed, which supports to generate system designs out of patterns. *Uses* and *Combines* interactions can be seen as design language operators, and the patterns being the operands. Semantics of the language needs to be defined more concisely. We are also trying to quantify hybrid patterns in terms of some design attributes. This quantification helps in selecting best hybrid pattern when there exists several alternatives.

## References


[Dir97] Dirk Riehle. Composite design patterns. In *Proceedings of Conference on Object Oriented Programming Systems, Languages and Applications(OOPSLA)*, pages 218–228, 1997.

[FR95] Frank Buschman and Regine Meunier. A system of patterns, in pattern languages of program design. pages 325–343. Addison-Wesley, 1995.

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

[JAG97] Janaki Ram D, Anantharaman K N, and Guruprasad K N. A Pattern Oriented Technique for Software Design . *ACM Software Engineering Notes*, 22(4):70–73, 1997.

[JAG+00] Janaki Ram D, Anantharaman K N, Guruprasad K N, Sreekanth M, Raju S V G K, and Ananda Rao A. An approach for pattern oriented software development based on a design handbook. *Annals of Software Engineering*, 10:329–358, 2000.

[JJR03a] Janaki Ram D, Jithendra Kumar Reddy P, and Rajasree M S. A Layered View of Patterns for Capturing Pattern Interactions and Analyzing Pattern Oriented Applications . In *proceedings of 5th National Conference on Object-Oriented Technology*, pages 1–11, 2003.

[JJR03b] Janaki Ram D, Jithendra Kumar Reddy P, and Rajasree M S. An approach to estimate design attributes of interacting patterns. In *Proceedings of 7th ECOOP's Workshop on Quantitative Approaches for Object Oriented Software Engineering (QAOOSE)*, 2003.

[Mag] Magnus Kardell. Masters Thesis :A Classification of Object-Oriented Design Patterns, Umea University.

[PDJ99] Paulo Alencer, Donald Cowan, and Jing Dong. A pattern-based approach to structural design composition. In *Proceedings of Twenty- Third Annual International Computer Software and Applications Conference*, 1999.

[She] Sherif M.Yacoub. PhD Thesis : Pattern-Oriented Analysis and Design(POAD): A Methodology for Software Development, West Virginia University.

[Wol95] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

[Zim94] Walter Zimmer. Relationships between design patterns,in pattern languages of program design. pages 345–364. Addison-Wesley, 1994.