Patterns and Design Adaptability

C. W. Irving and D. Eichmann
University of Houston – Clear Lake

Abstract

The successful use of patterns in a software development organization depends on a number of issues. One such issue is the classification of patterns in a manner supportive of the organization's model of software development. In modern reuse–based organizations adaptability of software products is a capital issue. Given this context, this paper discusses the issues and rationale for pattern classification based upon software adaptability, presents two such classification schemes and discusses their effectiveness when applied to two popular pattern catalogues.

Introduction

Patterns in general, and design patterns in particular, have recently become a very popular scheme for encapsulating and reusing good software engineering practice within software development organizations. The commonly available pattern literature such as [Gamma et al. 1995] and [Buschmann et al. 1996] firmly establishes the value of the technique and presents an elaborate set of practical patterns that can be used by designers working in many fields. This, however, is only the first step in the assimilation of patterns into the software development process: organizations then begin writing and using their own patterns, making them an integral part of the process. This is to say, the organization's "design heritage" is recorded in the pattern catalog that ensues.

In such a context, being able to find patterns becomes an issue. If the organization's pattern catalog is left as a "big bag of patterns," the only realistic search mechanism for pattern users is an exhaustive traversal — neither efficient nor pleasant for the prospective pattern user. A lesson that we can learn from the field of reusable software repositories is to classify patterns so that they are easier to identify without costly exhaustive searches. But how does one classify patterns?

Published pattern catalogues are often accompanied by classification schemes that were chosen to bring out the salient features of patterns in the catalog (such as the two–dimensional classification introduced in [Gamma et al. 1995]) but, as with the classification of any group of objects, different schemes support different search criteria. We definitely want schemes that support the way in which pattern users go about their business, as abstract feature–based classifications can only go so far to help users find patterns. In this paper, we will focus on one particular area that is valuable to organizations developing software using modern reuse–based techniques: design adaptability — the ability to make (usually limited) alterations to a design to customize it to a specific context.

Design adaptability is important to reuse–based design: anything built to be reusable is effectively built to be adapted. Conversely, reuse of existing designs usually means adapting them to their new home, they rarely fit perfectly "out of the box." This relationship between reuse and adaptation is even more predominant when considering advanced design paradigms such as software architectures [Shaw and garlan 1995] that are intended to be used over and over in a domain. This approach to developing applications in a domain revolves around the design of one or more adaptable architectural designs embodying the commonality present in the domain [Weisskopf et al. 1996]. This commonality is exploited not by parametrization of an overly generic application but rather as an extensible framework that readily can be adapted and completed into actual applications. That is, the approach leverages the assumption that similar applications in a single domain will exhibit a high degree of similarity in structure and function, by explicitly factoring out similarities in the form of a domain–specific architecture.

For instance, an organization charged with the design of anti–lock braking systems for a range of cars — going from a cheap econobox, to a family sedan, to a sportscar — is building a family of systems that basically do the same thing, but with different performance and costs requirements such that one single solution will not fit all uses. Making a single generic system would be nice, but would either end up being too expensive for the econobox or not good enough for the sportscar. What these systems share, though is the same physics, the same abstractions, the same fundamental modes of operation, etc. *These* are the things that can be pulled together in an architecture for the domain. This architecture still needs to be adaptable — the systems being developed are substantially different (in hardware, in modes of control, in capabilities, etc.).

As we can see, adaptability is a key attribute. Software architectures need to be adaptable for applications to be built around them, meaning that the domain engineers building the architecture must have a design "vocabulary" that is well–suited to adaptable software. The application engineers building actual system instances, on the other hand, require a vocabulary that allows them to adapt architectures and reusable components in manners that may not have been foreseen by their original designers.

Classification based on design adaptability is thus a worthwhile focus, both from the perspective of adaptable design and the adaptation of existing designs. Designers faced with these problems will, of course, develop their own set of favored techniques and patterns but will also rely on published pattern catalogues. This implies that any classification with respect to adaptability could (and should) be applicable to existing pattern catalogues. The remainder of this paper explores two such adaptability–based classification schemes and applies them to some well–known patterns from the Gamma et al. [Gamma et al. 95] and Siemens pattern catalogues [Buschmann et al. 96].

Our classification approach is based upon the examination of the actual design models present in each pattern. These models are the expression of what the system's design will look like following the pattern's application, so they can be reasoned about using our knowledge of object–oriented design techniques.

Adaptability Classification

An initial, very rough, classification may be obtained by simply asking whether patterns produce adaptable designs. The patterns in the catalogues were initially classified using three Yes/No questions. These questions were intended to highlight the relationship between particular patterns and the adaptability of solutions resulting from their application. Given that adaptability is a strong characteristic of software architectures, expectations were that such questions would allow the pattern space to be partitioned into patterns that are particularly suitable to adaptable architecture design, those that assist in the adaptation of designs and those that are neutral to adaptation.

The initial questions asked of each pattern collected were:

- Does the application of this pattern intentionally make the resulting design adaptable?
- Does the application of this pattern allow an existing design to be rendered adaptable?
- If it does not directly render the design adaptable, does the application of this pattern help the creation of an adaptable design?

Table 1 lists answers to these questions for the patterns published in Gamma, et al.

In some respects, these results are not all that enlightening. For starters, the partitioning of the pattern–space is very subjective. Many patterns could have been classified in different ways based upon the personal interpretation of the reader. Without an explicit model of architectures and their adaptation, it is very difficult to ensure that different readers will classify patterns consistently. Also, the partitioning is not very revealing of differences between patterns. The results show that many Gamma et al. patterns are well suited to the design of adaptable products while only a few are apparently means of adapting an existing design.

Pree, in his comparison of pattern approaches, described the Gamma et al. approach as not "pursuing the goal of describing how to use a framework" [Pree 1995, 64]. The lack of patterns providing means of adaptation is definitely in agreement with such a statement. Also, Pree's opinion that this approach "focuses on achieving the goal of capturing the design of frameworks so that the design can be reused in the development of new frameworks" is borne out by the high proportion of patterns that either render the design directly adaptable or help its future adaptation.

Category	**Pattern name**	**Makes adaptable?**	**Means of adaptation?**	**Help or hindrance?**
Creational	Abstract Factory	Yes	No	n/a
	Builder	Maybe	No	Maybe
	Factory Method	Yes	No	n/a
	Prototype	Yes	No	n/a
	Singleton	No	No	n/a
Structural	Adapter	No	Yes	Help
	Bridge	Yes	No	n/a
	Composite	Yes	Yes	n/a
	Decorator	Yes	Yes	n/a
	Facade	No	No	Help
	Flyweight	No	No	n/a
	Proxy	No	No	Help
Behavioral	Chain of Responsibility	Yes	No	n/a
	Command	Yes	No	n/a
	Interpreter	Yes	No	n/a
	Iterator	Yes	No	n/a
	Mediator	Yes	No	n/a
	Memento	No	No	Help
	Observer	Yes	No	n/a
	State	Yes	No	n/a
	Strategy	Yes	Maybe	n/a
	Template Method	Yes	No	n/a
	Visitor	Yes	No	n/a

Table 1: Adaptability of Gamma et al. patterns.

The results also illustrate the fact that a practical pattern catalog — however overtly aimed towards adaptable framework design — will contain patterns that are not directly means of rendering the design adaptable. Not every individual area of a good, adaptable design is itself adaptable. The *Singleton* is a case in point: It neither helps nor hinders[1] the adaptability of design products it is applied in, yet is a pattern that has a definite place in an adaptable architecture. In other words, not every problem faced by the domain designer is one where adaptation is an overriding factor.

Asking the same questions for the Siemens patterns published electronically to date also reveals interesting results (see table 2). All the Siemens patterns are deliberately intended to produce adaptable designs. From this we can deduce that, in the mind of the Siemens pattern writers, adaptability is an overriding pattern quality factor. This is, of course, a very laudable goal but patterns discovered in the course of an organization's work are likely to display varying degrees of adaptability. The lack of adaptability of the patterns is not fundamentally incompatible with their usefulness.

---

[1] There are published views to the contrary, such as Vlissides' discussion of the *Singleton* pattern in his C++ Report column [Vlissides 1996] where adaptable implementations of *Singleton* are discussed. However, the Singleton pattern's implementation as published in Gamma et al. is distinctly less aimed at adaptation than most other patterns in the catalog.

Both the Gamma et al. and Siemens pattern catalogues reflect the thesis that good design stems from built–in adaptability. Adaptation of existing designs is not given much attention. Unfortunately, designers working with reusable components or having to adapt an existing domain architecture to challenging new uses may be faced with the problem nonetheless.

Pattern name	**Makes adaptable?**	**Means of adaptation?**	**Help or hindrance?**
Blackboard	No	No	Yes
Broker	Yes	Yes	n/a
Command Processor	Yes	No	n/a
Layers	Yes	No	n/a
Microkernel	Yes	No	n/a
Model–View–Controller	Yes	No	n/a
Presentation–Abstraction–Coltrol	Yes	No	n/a
Pipes and Filters	Yes	No	n/a
Reflection	Yes	No	n/a
View Handler	Yes	No	n/a

Table 2: Adaptability of Siemens patterns.

The practical problem we are faced with when applying this initial classification scheme to the Gamma et al. and Siemens pattern catalogues is that it does not create a reasonable number of partitions: virtually all the patterns end up in a single "adaptable" partition. On one hand, this is interesting — we learn to appreciate the qualities of published patterns in a new light — but on another, this is terrible because haven't helped designers find patterns. We do, therefore, need to address the weaknesses of this scheme. In order to do so, we need to better understand what we mean by adaptability and learn to recognize the forms it takes.

"Thermal" Classification

A key concept introduced by Pree is that of "hot" and "frozen" spots in architectural frameworks [Pree 1995]:

- Hot spots are those areas of the architecture that have to remain flexible. Various applications of the architecture may exhibit variabilities in these areas.
- Frozen spots are the areas of the architecture that remain unchanged across applications.

Patterns that result in adaptable designs should naturally focus on one or more hot spots. Were the designs resulting from applying these patterns fully frozen, there would be no means by which to adapt them. It is thus possible to examine the members of pattern catalogues in terms of their hot and frozen areas.

In other words, hot spots are those parts of an architecture where things are expected to happen when it is adapted. Understanding when a pattern introduces a hot spot and what it deliberately leaves frozen is potentially very useful. The "things" that "happen" can take a number of forms, of course. That is precisely what we will be looking at in the following section.

The Building Blocks of Architectures

Architectures are composed of collaborating classes. Instances of these classes interact via protocols defined as part of the architecture and their participation in relationships allows them to be composed.

Another model of architectures is that of a set of interconnected components, each interacting via well–defined protocols [Allen and Garlan 1994]. This is a theoretically pleasing model but its rigor makes it quite impractical for our purposes. What we need is a more informal, yet valid, model of the ways in which architectures display hot and frozen spots.

Classes have semantics — However, the semantics of each actual class instance is the result of the combination of the semantics provided by its class definition and that of all its superclasses. An adaptable architecture can be designed in a manner that allows new application subclasses to be defined that subtly or substantially modify the semantics of those provided by the architecture. In doing so, the designer purposefully inserts a hot spot in the architecture.

Objects interact with each other — Interactions between instances of the same class or different classes follow a protocol. This protocol is the direct result of the various interfaces provided by the class(es) and the sequences of interactions that they are expected to perform. An adaptable architecture can be designed in such a manner that the architecture class interfaces and protocols between their instances can be changed as part of application design.

Objects can be composed — Participation of class instances in relationships allows them to be composed into a greater whole. Adaptability in an architecture can be provided by allowing the useful composition of instances in applications.

Hot vs. Frozen Aspects of Patterns

We can carry over the concepts of hot and frozen spots within architectures into the analysis of individual patterns supporting adaptability. By looking at whether class semantics, interaction protocols and composition are hot or frozen, we can classify patterns in a manner that is representative of the adaptation that they support. Furthermore, identifying the individual participants that are hot allows designers to match their expectations for adaptability with the actual pattern.

In order to verify the practicality of such a classification, consider the hot and frozen spots present in the solutions of Gamma et al. patterns. Each of the four areas of class semantics, object composition, type compatibility and protocol are designated as "hot" if they are intended to be adapted in future design steps or, in the case of patterns that are means of adaptation, if they are the area where the adaptation takes place. The actual pattern participants that are the focus of adaptation in the "hot" areas are also identified.

Table 3 shows the hot and frozen spots for the creational patterns in Gamma et al. With the exception of *Singleton*, all the creational patterns presented effectively trade the ability to change the concrete product being created and its factory, against freezing the creation protocol and type compatibility of the participants. Viewed from the standpoint of hot or frozen participants, the patterns, save for *Singleton*, uniformly allow the adaptation of the concrete product and factory classes.

Name	Hot	Frozen	Hot participants
Abstract Factory	Semantics	Type compatibility Protocol Composition	ConcreteFactory ConcreteProduct
Builder	Semantics	Type compatibility Protocol Composition	ConcreteBuilder (implicitly) Product
Factory Method	Semantics	Type compatibility Protocol Composition	ConcreteCreator ConcreteProduct
Prototype	Semantics Composition	Type compatibility Protocol	ConcretePrototype
Singleton		Semantics Type compatibility Protocol Composition	n/a

Table 3: Creational pattern hot and frozen spots.

Table 4 shows the hot and frozen spots for the structural patterns in Gamma et al. These exhibit greater diversity than the creational patterns just examined.

Name	Hot	Frozen	Hot participants
Adapter	Type compatibility Protocol	Semantics Composition	Adapter Adaptee
Bridge	Semantics	Type compatibility Protocol Composition	ConcreteImplementor
Composite	Semantics Composition	Type compatibility Protocol	Composite Leaf
Decorator	Semantics Composition	Type compatibility Protocol	ConcreteDecorator ConcreteComponent
Facade		Semantics Type compatibility Protocol Composition	n/a
Flyweight	Composition	Semantics Type compatibility Protocol	Client
Proxy	Semantics	Type compatibility Protocol Composition	Proxy

Table 4: Structural pattern hot and frozen spots.

The *Adapter* pattern essentially makes the opposite trade–off to the creational patterns: adaptability of type compatibility and the invocation protocol are given precedence over semantics and composition. The Adapter class encapsulates the Adaptee so that its class interface and protocol are changed while preserving the basic class semantics.

The *Bridge* protocol, on the other hand, brings adaptability to the design via the introduction of an abstract Implementor class. This technical solution is quite similar to the use of abstract class factories in the creational patterns and has the same adaptability trade–offs. The semantics of subclasses are free for adaptation, but at the cost of frozen type compatibility, invocation protocol and composition.

The *Composite* and *Decorator* patterns are quite similar in their structure and thus in their scope for adaptation. Both are structured as a recursive connection metapattern [Pree 1995, 126] which allows the semantics and composition (via aggregation relationships) of subclasses to be adapted at will. Once again, in order to do so, the type compatibility and protocol of the participants is frozen.

The *Facade* pattern is an example of a structural pattern that does not explicitly make any part of the design adaptable. It does, however, help adaptation through a localization of the inter–class couplings, but none is immediately adaptable.

The *Flyweight* pattern could easily be considered adaptable or non–adaptable as the reader sees fit. It is intended as a means to provide efficient storage in a complex object composition scheme. In doing so, the Client participants in the pattern gain the ability to compose Flyweights in any arbitrary fashion.

The *Proxy* pattern is a close relative of the *Adapter* pattern. From an adaptability standpoint, is very similar to the degenerate *Adapter* class where no adaptation of the participant's type compatibility is required. In this particular pattern though, it is the subject class' **semantics** that is being adapted, not its protocol.

Table 5 shows the hot and frozen spots for the behavioral patterns in Gamma et al. Like the structural patterns above, they show greater diversity than is immediately apparent in table 1. Looking at the hot and frozen spots present in pattern solutions makes more of the technical diversity in the Gamma et al. pattern catalog apparent while still focusing on each pattern's support for later adaptation.

Name	Hot	Frozen	Hot participants
Chain of Responsibility	Semantics Composition	Type compatibility Protocol	ConcreteHandler
Command	Semantics Composition	Type compatibility Protocol	ConcreteCommand
Interpreter	Semantics Composition	Type compatibility Protocol	TerminalExpression NonterminalExpression
Iterator	Semantics	Type compatibility Protocol Composition	ConcreteAggregate ConcreteIterator
Mediator	Semantics	Type compatibility Protocol Composition	ConcreteColleague
Memento		Semantics Type compatibility Protocol Composition	n/a
Observer	Semantics Composition	Type compatibility Protocol	ConcreteSubject ConcreteObserver
State	Semantics	Type compatibility Protocol Composition	ConcreteState
Strategy	Semantics	Type compatibility Protocol Composition	ConcreteStrategy
Template Method	Semantics	Type compatibility Protocol Composition	ConcreteClass
Visitor	Semantics Composition	Type compatibility Protocol	ConcreteVisitor ConcreteElement

Table 5: Behavioral pattern hot and frozen spots.

Discussion

From a purely technical standpoint, we can see that there are two main techniques used in the adaptable solutions in the Gamma et al. pattern catalog. The first and most popular is the use of abstract or semi–abstract classes allowing semantically different, yet type–compatible classes sharing the same protocol to be added to the design. The second technique is to use to use the frozen type–compatibility and protocol in order to allow composition of pattern participants in unforeseen ways.

The prevalence of these techniques in Gamma et al. patterns apparently correlates well to the patterns' intent: patterns intended to make the resulting design adaptable use these techniques almost exclusively. Patterns that are intended to adapt existing designs, such as *Adapter*, trade off extensibility of semantics and composition for the adaptation of type compatibility and protocol. The other major factor in the appearance of hot and frozen spots in the pattern solutions is their implementation constraints: these patterns are intended to be implementable in a typed programming language so the distinction between type compatibility and class protocol is practically non–existent.

Since the hot and frozen spots in pattern solutions are directly tied to the actual design–level constructs employed, identifying hot and frozen spots is a relatively easy task. Specific design structures (such as abstract classes) are reliable indicators of hot areas while the textual portion of the patterns provide plenty of information to confirm these choices. This leads us to believe that practical classification in this manner is quite feasible and relatively painless.

An issue of greater importance is whether or not this distinction between hot and frozen spots is of value to the pattern user. When looking for useful patterns, the user intends to locate as many patterns as possible which match the problem at hand and differentiate between them in a manner that is meaningful to him/her.

A designer working on a domain architecture is concerned with the freedom of adaptation exhibited by the resulting design. The analysis of the domain and constraints imposed on the design define the problem to solve by the prospective patterns and the context where they will be applied. The domain also defines expectations of variability between applications which the domain designer tries to fulfill by building adaptability into areas of the domain architecture. In practice, this means that for every major design decision impacting areas of the architecture where variability is present, the designer asks him/herself "how can I support the variations I expect applications will need for this part of the architecture?" It is clear that total adaptability is neither feasible nor desirable, so a deliberate informed choice has to be made.

Design pattern classification using the hot and frozen spots of the pattern solutions exhibits some shortcomings. First, it shares the common failing of any partitioning of pattern space: partition boundaries are sharp. Subtle differences between means of adaptation simply are not picked up. Second, patterns whose value rests in consequences difficult to quantify in terms of the solution–space also do not classify particularly well. For instance, *Composite* and *Decorator* share virtually identical solution design models yet serve different purposes. Lastly, it is not always clear when comparing similar patterns where trade–offs are made to allow the adaptability associated with hot spots.

The second shortcoming mentioned above is illustrated by a number of the Siemens patterns such as *Reflection* and *Broker*. These patterns are classified as architectural patterns by their authors and include a discussion of a design meta–level. For instance, *Broker* includes the construction of an IDL to programming language translator for use by the system implementors as part of the pattern solution. Given the degree of freedom provided by such patterns, it is meaningless to talk about individual hot areas in the solution — anything within reason is possible.

The distinction in the Siemens pattern catalog between design patterns and architectural patterns is not directly related to our inability to distinguish hot and frozen design aspects. Fundamentally, the Siemens patterns form a continuum of abstraction levels from pure design structures upwards. For instance, the *Pipes and Filters* (architectural) pattern clearly trades hot semantics and composition against frozen type compatibility and interaction protocols. It is the introduction of a design meta–level in some architectural patterns that does, however, render this classification mechanism practically useless in their case.

Conclusions

Even without the advantage of a pattern collection written by practitioners "in the trenches," we can see that a classification scheme, such as ours, based on hot and frozen spots is an useful addition to a pattern collection whenever adaptability is an issue for designers. Our application of the classification to patterns in the Gamma et al. catalog reveals technical differences between patterns that are useful yet also independent of the classification scheme used by their authors. The success of the classification with existing pattern catalogues, even when they initially appear relatively uniform when it comes to adaptable design, is an encouraging sign that pattern classification with respect to adaptability is both feasible and valuable.

A supportive and useful pattern classification scheme is but one aspect of the successful use of patterns in a practical dual–lifecycle software development process (one comprised of domain engineering and application engineering) [Irving 1996]. Other critical issues include support for the whole pattern lifecycle (from discovery/writing, to their identification and application) in the organization's process, and pattern measurement and management.

References

Allen, R., and D. Garlan. 1994. Formalizing Architectural Connection. *Proceedings 16th International Conference on Software Engineering*. IEEE.

Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. 1996. *Pattern–Oriented Software Architecture — A system of Patterns*. Wiley & Sons.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison Wesley.

Irving, C.W. 1996. *Dual–Lifecycle Software Development Processes: A Pattern–Based Approach*. Master's Thesis. School of Natural and Applied Sciences. University of Houston – Clear Lake. Available electronically as: http://ricis.cl.uh.edu/cwirving/thesis/

Pree, W. 1995. *Design Patterns for Object–Oriented Software Development*. Addison–Wesley.

Shaw, M., and D. Garlan. 1995. *Software Architecture*. Prentice–Hall.

Vlissides, J. 1996. Pattern Hatching: To Kill a Singleton. *C++ Report*. 8(6). SIGS.

Weisskopf, M., C.W. Irving, C.W. McKay, C. Atkinson, and D. Eichmann. 1996. Maintenance in a Dual–Lifecycle Software Engineering Process. *International Conference on Software Maintenance*. Monterey, CA.