Using Patterns to Define a Software Solution

4 out of 7 rated this helpful Rate this topic

Joseph Hofstader

November 2006

Applies to:
Software Patterns

Summary: Patterns exist in all mature engineering disciplines, including software engineering. Learning how to leverage patterns to define a software solution takes time and effort. Knowing the vocabulary of patterns is not enough to be proficient in design. The ability to apply patterns to create domain-specific frameworks is what is most important. The best way to gain proficiency in applying patterns is by actually designing and developing software solutions. With experience and constant reinforcement, designing a patterns-based software solution will become second nature. (10 printed pages)

Contents

"Miles Davis presents here frameworks which are exquisite in their simplicity and yet contain all that is necessary to stimulate performance with sure reference to the primary conception."

–Bill Evans, from the liner notes of the Miles Davis album Kind of Blue

Introduction

The first time I read the liner notes of Miles Davis's *Kind of Blue* album, I was surprised by the similarities between the frameworks Davis created for his legendary recording and the attributes of good software frameworks: simple, extensible, and focused. The frameworks created by Davis for *Kind of Blue* are frameworks in time, which defines the patterns for the pulse of a composition. The musical patterns in time are general-purpose and reusable over many compositions. In a similar vein, a software solution needs frameworks to define the solution's

architecture in the context of the problem domain. Good software frameworks are usually defined by using general-purpose patterns that are reusable over a number of domains.

Patterns exist in all mature engineering disciplines, including software engineering. Learning how to leverage patterns to define a software solution takes time and effort. Some developers have the good fortune of working with experts in patterns who provide mentoring in the disciplines of architecture and design; others are left on their own to acquire these skills. With or without mentors, developing a proficiency in patterns does not happen by accident and requires dedication. There is a large learning curve in acquiring the patterns vocabulary, and an even larger learning curve in understanding where patterns can be applied in a software solution.

The benefits of learning patterns are worth the effort. Patterns give software architects and designers the ability to design a robust solution rapidly by using proven techniques. Patterns also give software professionals a common vocabulary from which they can convey ideas without having to describe every detail of the intent of the design.

A Brief History of Software Patterns

Object-oriented patterns for software engineering became a significant subject about two decades ago. In 1995, patterns moved to the mainstream with the publication of the seminal *Design Patterns: Elements of Reusable Object-Oriented Software* [GOF]. This book cataloged a number of patterns for designing extensible object-oriented solutions.

In 1996, the Pattern-Oriented Software Architecture (POSA) series began publication. The POSA books catalog architectural patterns and patterns for specialized software problems, like concurrency, networked objects, and resource management. As of October 2006, there are three volumes in the POSA series, with a fourth volume in the works.

Since the mid-1990s, software patterns have become an industry in themselves. A recent search of books on "software patterns" at Amazon.com returned close to 2,000 results. There are also a number of companies that provide consulting services to train and mentor software professionals on patterns.

Learning to Think in Patterns

A critical first step in becoming proficient in learning patterns is to learn the vocabulary of the domain. This requires studying material that thoroughly covers the subject. The book *Design Patterns* [GOF] is the basis of most of the subsequent literature on patterns, and should be a candidate for reading by any aspiring software architect or designer. Even with examples defined by using OMT (a predecessor to UML) and C++, this volume was obviously written to teach patterns from the ground up. There are many other resources that can give the reader examples in Java, C#, and UML that can be used to supplement the legacy examples in *Design Patterns* [GOF].

Design Patterns [GOF] begins with an outstanding summation on the benefits of object-oriented design and the motivation for using patterns. *Design Patterns* [GOF] then continues with an example application in which patterns are incorporated into the design. The example points to a number of patterns in the book and puts them into the context of a software solution. Because vocabulary is only half of what is necessary to be proficient in patterns (the application of patterns being the other half), this chapter is critical in educating the reader on how to apply patterns. After reading through the example and the referenced patterns, the reader can read through the other sections sequentially, as they would any other book. This helps the reader learn the other patterns, and reinforces what was read in the example.

An approach to learning patterns is to include them in a solution you are developing while reading the patterns material. This solution can be a project done inside or outside of work.

Defining a Solution with Patterns

Using patterns to define a software solution is an analytical task that requires abstract thinking. There are a number of documented patterns that allow the structure and behavior of a solution to be defined at different levels of abstraction: for the solution architecture, and for the frameworks contained within the solution.

Along with other cross-solution concerns like security and logging, the architect must define the following attributes of the solution:

1. The solution's boundaries
2. The solution's structure
3. The frameworks that support the solution's domain

Defining the Solution's Boundaries

The first step in defining a software solution is to understand the boundaries of the solution. Defining the system boundaries adds focus to the solution and allows the solution to be designed and developed. Without clear boundaries, a solution will be a moving target and can become mired in ambiguity.

There are usually some formal or informal system requirements from different stakeholders from which you can derive the solution's boundaries. Along with having a thorough understanding of the overall intent of the solution, the following system requirements are critical in defining the boundaries of the solution: the external events that trigger the solution's functionality, and the external processes on which the solution has dependencies.

Taken holistically, these requirements allow the architect to define the context of a solution. Awareness of the events that trigger the behavior of the solution and knowledge of the data that resides in the dependencies of the solution allows the architect to "normalize" the solution by separating the objects that reside internally within a solution and the objects that reside externally. Understanding the processes on which a solution has dependencies allows the

architect to narrow the behavior of the solution and leverage the behavior of the external processes.

Defining the Solution's Structure

After the solution's boundaries have been defined, the architect is ready to decide how to structure the solution. The structure of the solution provides a conceptual understanding of the solution space. Defining the structure of the solution very early in the process is a critical step in defining the architectural framework for the solution. The architectural framework will help ensure consistency throughout the solution, with a goal of making it more extensible and maintainable.

Similar to defining the boundaries of a solution, there are a number of system requirements that let an architect define the structure of the solution. Sometimes, these requirements will seemingly contradict each other, and a number of trade-offs will have to be weighed. It is important to note that the structural patterns chosen for a solution do not imply the implementation of any particular technologies. The technologies that you use to implement the solution should be based on the technology's ability to fulfill the functional and nonfunctional system requirements within the structure of the defined solution.

Along with the system requirements used to define the solution boundaries, there are a few other requirements that can weigh on the stricture of the solution: the solution's security, performance, and transactional requirements.

Security requirements can necessitate that additional components be added to the solution. Those additional components can influence the defined structure of the solution. In a similar vein, the solution's performance requirements might allow or prohibit the addition of structural elements that can add overhead to the solution. The transactional requirements of the solution might also require additional parts of the solution that can affect its structure.

There are a few publications on software architecture that catalog different structural architecture patterns. Based on the defined systems boundaries and the aforementioned system requirements, the architect can select the appropriate structure for the solution. It should be noted that not all architectural patterns can be formally documented, and that a solution might require a variation of one that currently exists.

The following examples describe the drivers of system requirements and how they apply to different architectural patterns.

The "Layers" [POSA 1] architectural pattern might be appropriate, if the requirements contain the following provisions:

- Distinct parts of the solution can be conceptualized at different levels:
    - If the solution lends itself to reuse at each conceptual layer.
    - If a common protocol can be used to access the services contained at each layer.

- The solution will be exposed publicly and wants to hide the complexities of the implementation.
- There are a number of external processes on which the solution has dependencies, requiring technical adapters.

The "Model View Controller" architectural pattern [POSA 1] might be appropriate, if the following are requirements of the solution:

- The solution contains a user interface and wants to display objects of the same type in different formats.
- The solution can be accessed by consumers other than a user interface, like an external process:
    - If the solution wants to expose the same objects (Model) to all consumers.

This should not imply that the Model View Controller and Layers patterns are mutually exclusive. For example, a public layer within a layered solution can use the Model View Controller pattern for the user-interface consumer.

Defining the Domain Frameworks

After the structure of the solution has been defined, the next architectural effort that must take place is defining the frameworks within that structure. In most complex solutions, a number of frameworks are required to fulfill the functional requirements.

The first step in defining the domain frameworks is to understand the objects and relationships within the context of the problem domain. A UML analysis model is a good place to start defining the objects and relationships for a solution. If an analysis model for the solution becomes too large, it can be broken up into subdomains. Each subdomain can lend itself to a different framework.

Design Patterns [GOF] divides patterns into three categories: structural, behavioral, and creational. Looking at the solution requirements, the solution's architectural structure, and the analysis models in the context of these categories is a good way to begin defining the domain frameworks.

Structural Patterns

Structural patterns are used to define the composition of objects and control access to subsystems of objects. Using the analysis model in the context of solution's structure, the architect can begin defining the frameworks in the context of the structural patterns.

The following examples describe the trade-offs of system requirements and how they apply to different structural-design patterns. In contrast to the architectural patterns which define an entire solution or subsystem, there are usually a number of structural-design patterns in a single framework.

The Composite [GOF] structural pattern might be applicable, if the domain contains the following relationships:

- Objects of the same type (subclass) are contained within objects of that same type in a hierarchical structure.
- We want to treat parts of the hierarchy independently in some scenarios, or combined in other scenarios.

The Facade [GOF] structural pattern might be applicable:

- If a subsystem is significantly complex and the details should be abstracted to the consumer.
- If the system requirements indicate the subsystem can be deployed on a remote server, giving an interface to expose remotely.

Behavioral Patterns

Behavioral patterns are concerned with algorithms and communications between objects. Behavioral patterns address issues like which behaviors are applicable in which state of an object, and which algorithm is applicable for a particular object for a given context.

The State [GOF] behavioral pattern would be a good addition to a framework, if:

- The behavior of an object changes, based on the state of the object.
- Consumers are accessing the object through a stateless facade, and the applicability of a method exposed through the interface must be determined at run time.

The Strategy [GOF] behavioral pattern would be a good addition to a framework, if:

- The object wants to expose a static interface and determine which algorithm to run, based on a criterion determined at run time.

Creational Patterns

Creational patterns are concerned with the instantiation of objects. The creational patterns focus on the composition of complex objects and the encapsulation of creational behavior.

The Builder [GOF] creational pattern might be applicable, if:

- An object requires different representations in different contexts.

The Singleton [GOF] creational pattern would be a good addition to a framework, if:

- A single instance of an object is needed.

Example for Using Patterns to Define a Software Solution

The example for this article is a simple payment-verification system for an e-commerce business.

Solution Boundaries

The requirements for the solution say that it must accept an order from the order-fulfillment system and verify that the payment is legitimate in either the Bank Card Verification system or the Charge Card Verification system. Figure 1 shows the external processes and events on which the payment-verification system relies.

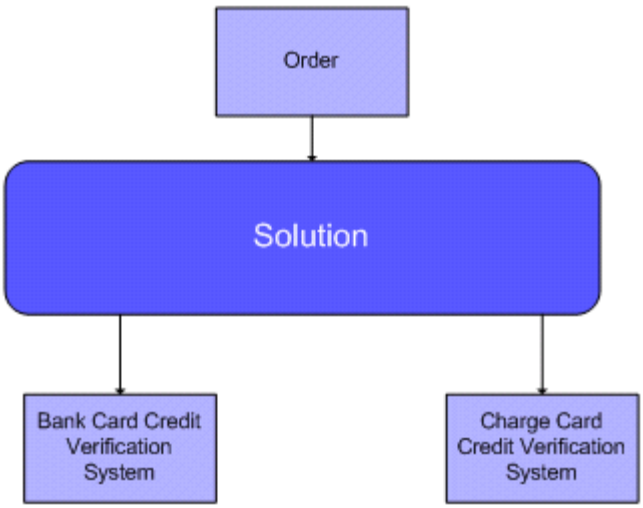

Figure 1. The solution's external events and dependencies

Solution Structure

The requirements also state that the solution requires a secure public interface. Because the external payment-verification systems run on different platforms, we know that we need technology adapters to access these systems. Taking these requirements into consideration, the structure lends itself to the Layers [POSA1] architecture pattern with a public layer, a domain layer, and a technology-adapter layer. Figure 2 shows the structure of the solution.

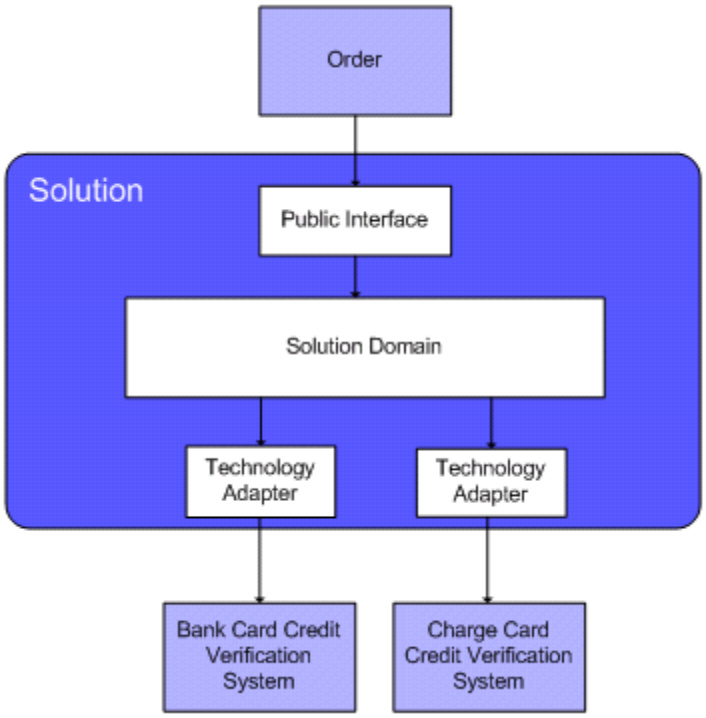

Figure 2. The solution's layered structure

Analysis Model

The solution requirements state that an external event passes an account number, the total amount that must be verified, and a payment method to the solution. Based on the requirements, we conclude that there are two domain objects required to fulfill the solution requirements: Account and Payment. Figure 3 shows the objects and relationships in the analysis model.

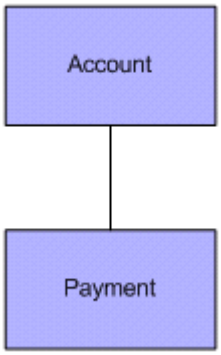

Figure 3. The solution's analysis model

Structural Patterns in the Solution

After defining the analysis model, it is understood that the solution needs to be structured based on the account and the payment objects. The requirements also say that in certain circumstances

an account can have child accounts related to it—like a company that wants centralized billing for all orders, but wants to track the individuals who are placing the orders. For this requirement, we implement the Composite pattern [GOF]. Also, we anticipate this system growing in size and complexity and would like to address that in the design by using the Facade pattern [GOF] to abstract the complexities. Figure 4 depicts the structural patterns in the solution.

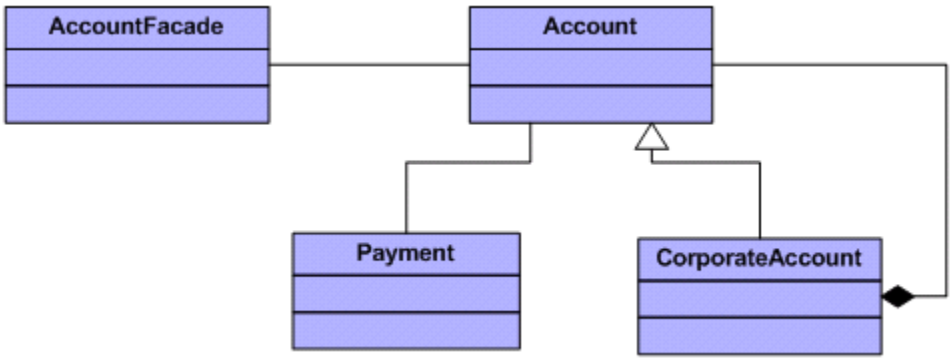

Figure 4. The Composite and Facade structural patterns

Behavioral Patterns in the Solution

Drilling down on the solution requirements, we understand that the state of the account might require that credit requests be rejected, for example if the account is in default. The State Pattern [GOF] allows us to place that constraint on the account object.

There is also a requirement that the solution must interface with at least two different credit-verification systems. This requirement lends the design to present a consistent interface to access the payment-verification systems, which in turn lends the solution to implementing the Strategy pattern [GOF]. Figure 5 shows the solution with the structural and behavioral patterns.

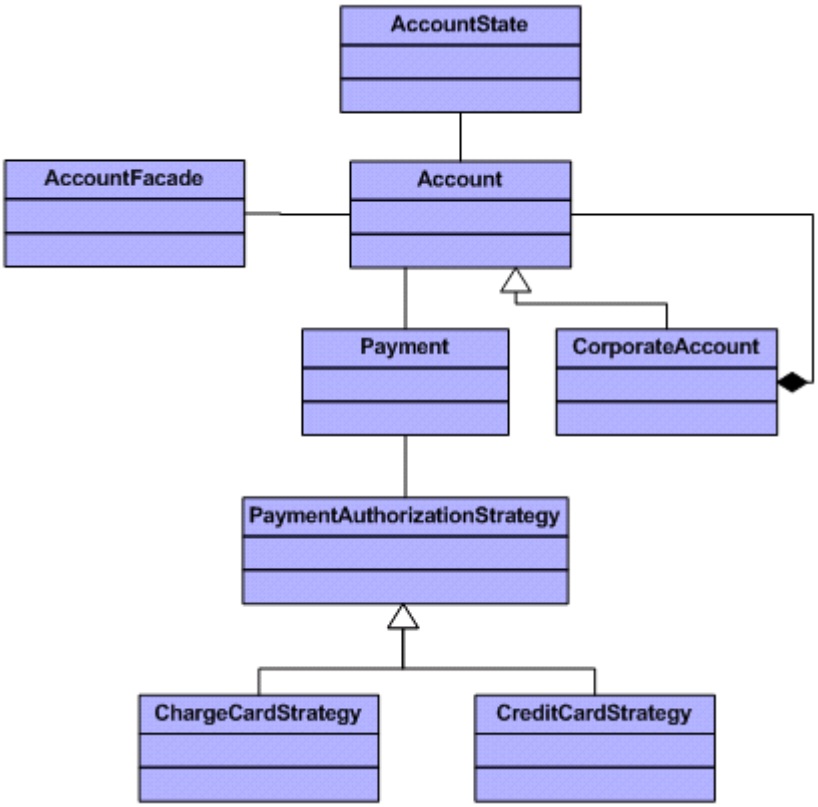

Figure 5. The solution containing structural and behavioral patterns

Creational Patterns in the Solution

The last requirement the solution must fulfill is that all authorizations for an account must happen sequentially, requiring a single instance of the account to be in the solution at a given time. The Singleton pattern [GOF] fulfills this requirement. We decide to use the Builder pattern [GOF] to encapsulate the singleton logic with the other logic required to build the account object. Figure 6 shows the payment-authorization framework containing Creational, Structural, and Behavioral patterns.

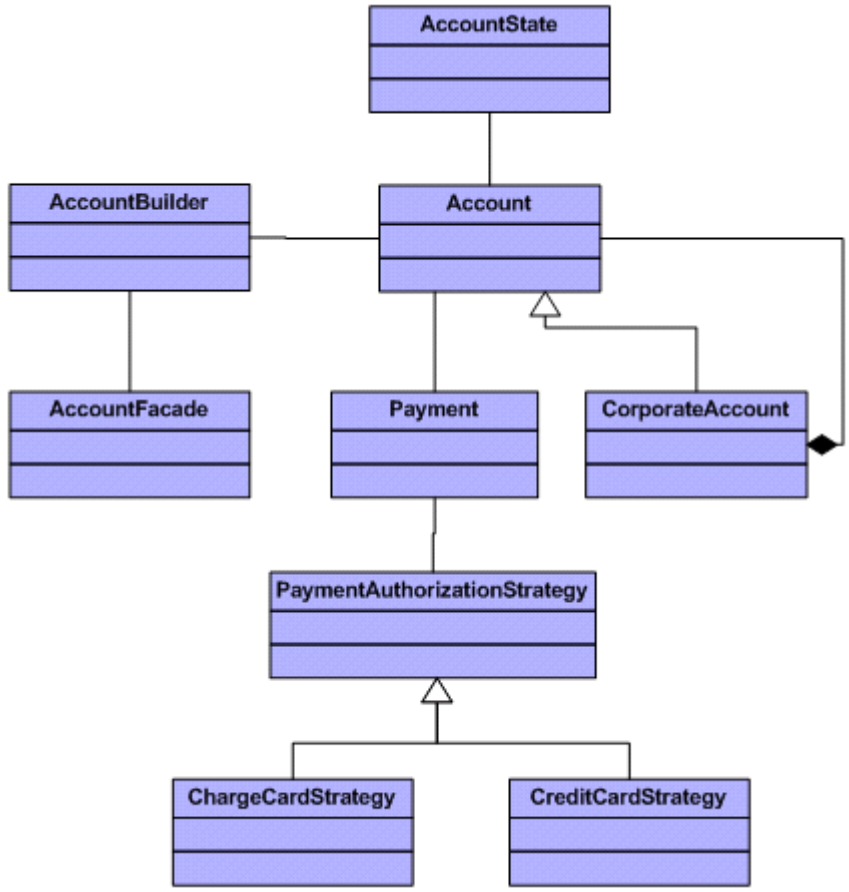

Figure 6. The payment-authorization framework

Conclusion

Patterns are an important tool in the analysis and design of a software solution. Patterns give the architect the ability to conceptualize a solution at different levels. Patterns are also a valuable way of communicating concepts between software professionals.

All solution architecture and design should map to functional requirements. By implementing patterns in the design of a solution, an architect can create domain-specific frameworks to provide consistency throughout a software solution. Because system architecture and design are highly subjective, the best assessment of a framework is whether it satisfies the defined functional requirements, and whether it is flexible, maintainable, and extensible.

A formal process is not necessary to define a patterns-based solution. An agile approach of coding functional requirements and refactoring to patterns-based frameworks might make sense if the domain is not well-defined initially. Conversely, with a well-defined domain, you can begin solution design by using patterns and modify the design to fulfill other requirements, such as performance.

Learning to architect a software solution by using patterns takes effort. Knowing the vocabulary of patterns is not enough to be proficient in design. The ability to apply patterns to create domain-specific frameworks is what is most important. The best way to gain proficiency in applying patterns is by actually designing and developing software solutions. With experience and constant reinforcement, designing a patterns-based software solution will become second nature.

References

[POSA1]
Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. West Sussex, England: John Wiley & Sons Ltd., 1996.

[GOF]
Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison Wesley Professional, 1995.

About the author

Joseph Hofstader is a Systems Architect at Avaya, with the title of Distinguished Member of Technical Staff. Joe has spent the last 12 years in the architecture, design, and development of solutions in the telecommunications industry. Joe holds a master's degree from George Mason University in Fairfax, VA.