Chapter 1
Introduction

Idioms, design patterns, pattern languages, architectural patterns, integration patterns. It's where we've been and the procession of software patterns marches on. I here present a higher order of software pattern—the Enterprise Business Pattern, or EBP. EBPs consume design, architecture, and integration patterns in large quantities. They define the essense of large, complex, industry-standard, product-based solutions. EBPs are pattern systems, but they are above those concerned only with class design and application architecture. Here I introduce the concepts that I have noted among the patterns found in enterprise business solutions today.

Patterns used in the development of software have helped many engineers make better architecture and design decisions and thereby produce better software. I expect that trend to continue. As new computing platform architectures, components, policies, tools, and standards appear, software patterns will be published to the greater good of the software engineering community. In this work I capture patterns that help software architects and developers identify large, complex enterprise elements that address specific electronic business problem domains.

Software patterns are certainly nothing new. Patterns in general have been used for several decades. What building architects and engineers learned about reusable designs was long ago refitted to benefit their software counterparts. Since that time patterns have been applied to every layer of software systems as well as to the organizations that develop them. Extensive and exhaustive work has been done already to produce patterns, familiarize us with, and demonstrate to us how patterns are used.

Based on these facts, how do I plan to introduce a new theme on patterns?

In one sense I plan to offer nothing very different from what is already known and used. A pattern is the definition of a solution that can be reused time and again, but without ever applying it in exactly the same way twice. It's a best practice for solving a certain kind of problem in a certain way under a certain set of circumstances. From that standpoint I cannot really improve on anything at all. A pattern, is a pattern, is a pattern.

On the other hand, I will introduce you to a new *kind* of pattern. It's really been the introduction of new *kinds* of patterns that has advanced pattern development and consumption. While various pattern authors have touched on patterns about large business solutions in the enterprise, there have not yet been extensive and exhaustive product-neutral patterns developed. I have put together a full complement of patterns about business solutions in enterprise computing.

Compared to historical works, my patterns are quite different. Previously much of the focus in pattern development has been applicable to the lower levels of software construction. Much work has been done to address design, architecture, and integration solutions through the use of best practices discovered from experience. Naturally as lower-level patterns are discovered, it leads the way to the recognition and definition of higher-level patterns. Design patterns have led to architecture patterns, and architecture patterns to integration patterns.

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

Enterprise Business Patterns, or *EBP*, is a next logical step. That is naturally so because the older patterns serve as a foundation and are consumed in large quantities—used extensively—inside the EBPs. Furthermore, because my patterns are so much larger than their predecessors, I take advantage of using predefined mechanisms for presenting an EBP. I use *pattern systems* [POSA1] or pattern frameworks, as well as a pattern language approach as I decompose the fine-grained details of the large, overarching patterns. These approaches help me organize and manage the complexity involved. What I am doing can be compared to the construction team building the floors of a sky-rise building after the architecture, design, as well as the foundation have already been completed.

When you have been around software development for a decade or two, you see new technologies, programming languages, methodologies, and development processes come and go. You likely have observed, though, that those of especially good quality stick around. Call them the *technology shakers*. The pattern movement took serious hold around the early 1990s. The first software pattern work to take the industry by storm, as you likely know, was *Design Patterns* [GoF], which appeared in book form in 1995. It's been nearly a decade since that work was published, and newer works have enhanced that offering and continue to be delivered down to this time. Clearly, software patterns are one of those *technology shakers*. With such a strong background and the proven extensibility of the original idea, new, higher-level patterns will have even more impact on the industry. Will *Enterprise Business Patterns* be the next to march in the pattern procession? The ideas presented here are certainly helpful. But I believe they may also help establish a new widespread trend toward large, complex patterns.

Pattern Development—The Procession

Since my patterns make extensive use of design, architectural, and integration patterns, let's briefly review some of the predecessors and consider how work on patterns has progressed to date. As we do so, I build on each lower-level pattern *type* and establish the next higher-level pattern type, until I establish a sound basis for the EBP level.

Design Patterns

Design patterns, the most widely used software pattern to date, are low-level, nuts and bolts guidelines to creating better software. They are language independent, but are based on object orientation. Residing above language idioms in their level of abstraction, they are focused on capturing simplistic *creational*, *structural*, *behavioral* perspectives of software. Stating that they are low-level in no way minimizes their value to or impact on a broad range of engineers. Design patterns have helped many a developer by crystallizing the best approaches to software design and implementation regardless of the kinds of software to which they are applied. Nonetheless, design patterns are basically limited to influencing how classes are defined, how groups of classes are composed and arranged, how instances of classes access other instances, and how a few objects interact and collaborate. Design patterns are concerned with the finer-grained aspects of software construction.

Note: Design patterns, and software patterns in general, are not limited in use to object-oriented systems and applications. However, my work here assumes that an

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

object-oriented approach is being used. Certainly patterns are most widely applied to object-oriented development today.

Let's review a few design patterns, a sampling from each of the categories found in the timeless book *Design Patterns* [GoF]. A synopsis of each of the patterns discussed here is found in **Table 1.1**.

Category	Sample Pattern	Intent
Creational	*Singleton*	Ensures a class has only one instance, and provides a global point of access to it.
Structural	*Facade*	Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Behavioral	*Mediator*	Defines an object that encapsulates how a set of objects interacts. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and lets you vary their interaction independently.

Table 1.1: The categories of classic design patterns and a sample pattern within each category.

1. Creational: Singleton. Likely many reading this is familiar with the *Singleton* pattern. Its use is very common. If you need to ensure that there is only one instance of a given class, you need to implement a singleton. You protect against a client's ability to create multiple instances of the class by preventing the new operation (message) from being invoked (sent) to the class. In Java and C# terms, you hide the class' constructor from public access. A factory method acts as the single point of access to the class' instance. When the factory method is invoked, it checks whether the single instance already exists. If it does not exist, the method creates one and only one instance and returns it to the client. If the single instance has been created on a prior invocation, the factory method simply returns (answers) the pre-existing instance. Here's a sample of the *Singleton* pattern implemented in Java followed by a code snippet of a client accessing and using it:

```
public class AppConfig
{
    private static AppConfig instance;

    public static synchronized AppConfig getInstance()
    {
        if (AppConfig.instance == null)
        {
            AppConfig.instance = new AppConfig();
        }

        return AppConfig.instance;
    }

    public String getDatabaseUrlAsString()
    {
        // . . .
    }

    protected AppConfig()
    {
        super();

        this.init();
```


© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

```
    }

    protected void init()
    {
        // . . .
    }
}
```

```
AppConfig aConfig = AppConfig.getInstance(); // only one instance ever

String aDatabaseUrl = aConfig.getDatabaseUrlAsString();
```

Clearly the *Singleton* pattern's concerns are limited in scope to just one encapsulated object every time it occurs in every application around the globe. It's concerned with the creation of a single object. Beyond the object's creation it only ensures that any client having access to the singleton class may obtain the single object and never effect the creation of more than one instance. It's a fairly limited nuts and bolts type of pattern.

You may sometimes be able to judge a lower-level pattern from a higher-level pattern by examining how many times a particular pattern occurs in a medium- to very-complex application. That's because a lower-level pattern will not be able to be applied one time across a broad range of application concerns. Rather, you tend to see a lower-level pattern applied each time a particular applicable application concern must be addressed. It's likely that in a medium- to very-complex application there will be dozens or even hundreds of singletons.

2. Structural: Facade. When you have a complex object model that must be accessed by a client, it's much easier for the client to do so if a *Facade* is provided for them. If a complex object model were a high mountain range, then a facade would be a flat, smooth road that made the enormous peaks and deep gorges and valleys disappear. A facade presents a limited and specialized API that facilitates much simpler access to the more complex object model behind it. As a design pattern, a single facade will generally be implemented in front of, say, a database domain model, to allow the client to access some special aspect of the model without having to understand the details of the model. The facade "understands" the complexities so the client doesn't have to.

The *Facade* pattern is much more versatile than *Singleton*. While in many cases it will be used at a design level with limited overall platform scope, it can also take on a more architectural role. For example, whenever many unrelated and highly distributed clients must access an object model (such as an enterprise database domain model), the use of a facade in front of the data model has much more architectural significance.

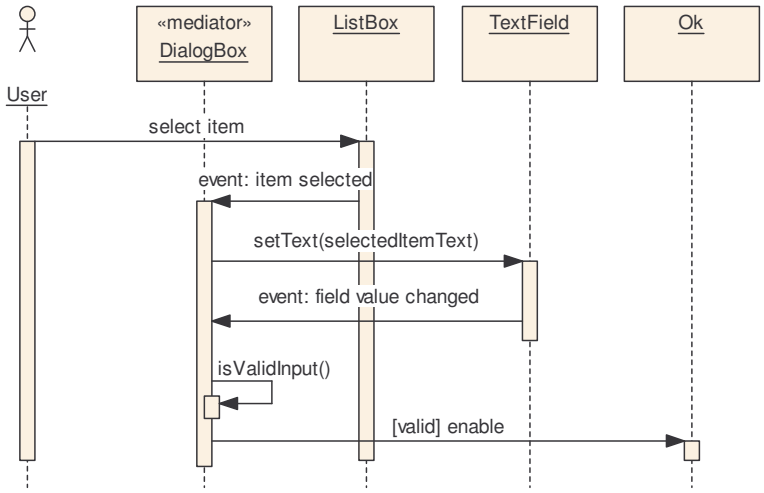
3. Behavioral: Mediator. Probably fewer developers have used the *Mediator* pattern than those who have used *Singleton* and *Facade*. At face value it appears to have rare usage. But if you have done modern GUI development, where frame windows own child windows that must interact with one another, you likely have used the *Mediator* pattern. How so?

Imagine a dialog box that contains four controls, an entry field, a listbox, and two command buttons, namely Ok and Cancel. The requirements for the dialog box are as follows: The entry field must contain a valid text value before the Ok button is enabled. This listbox contains valid values for the text field, and when a listbox item is selected its value is placed in the text field, which in turn enables the Ok command button. There are two ways to design the solution to this problem domain. You can create special versions

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

of all the controls such that each control knows a lot about its interaction with its sibling controls. For example, when the user selects an item in the listbox, the listbox would have to inform the text field to accept the item as its input value. This produces a very tight coupling between control components, and every dialog box that uses this technique will have to contain custom implementations of each control. Or, you can use the *Mediator* pattern to decouple the lot.

I assume you'd prefer the mediator approach. In that case the dialog box frame window, the parent window of all the child control windows, acts as the mediator. Each individual control reports to its parent dialog component all significant events that occur within them. For example, if the user selects an item in the listbox, the listbox tells the parent dialog frame that an item was selected. It's then the responsibility of the parent component to decide what to do with the selected item. In the case of our dialog box, the text field's input value will be set to the selected listbox item. The text field will then report back to its parent that its input has changed. The dialog box then validates whether the input is correct. If it is, the dialog box enables the Ok button:



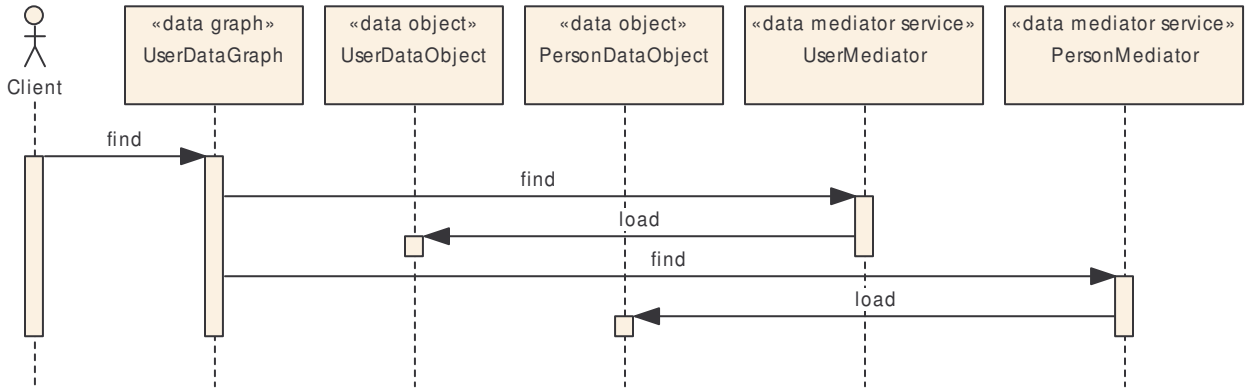
As it turns out, you've used the *Mediator* pattern quite a bit, or at least experienced its strengths often. Now if only our business systems could make better use of them! Otherwise the *Mediator* pattern, being quaint and barely understood, may not make a huge impact on the overall system.

Think about how the *Mediator* pattern alone could be used to make enterprise "components" *truly* components. If, for example, an EJB is to be *truly* reusable, its designers should learn from GUI framework design. Concerns for such things as how a third-party component gets its data could be completely eliminated if the EJB's designers specified an interface based on a *Mediator*. A server-side integration mediator object provided by the EJB consumer by implementing the interface defined by the component developers would act like the dialog box. The integration mediator would coordinate interactions between components that know nothing about the existence of others, such as entity beans used to access database data and the EJB business component purchased "off the shelf."

Interestingly the *Service Data Object* (SDO) specification, a new Java Specification Request (JSR), number 235, calls for the use of a special Mediator pattern. SDO uses a

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

Data Mediator Service, or DMS, to manage the retrieval and updating of persistent data. The SDO spec uses the Mediator pattern in much the same way that I just described:



To be sure, a design pattern is much like a leaf or two on a twig, on a limb, on a branch, on a trunk of a tree in a forest, when viewed from the perspective of an entire system or even an enterprise of systems. If our interest in patterns went no further than design patterns we would be '*engineering in the small*.' [Plauger]

Pattern Languages

A pattern language is the definition of how two or more patterns work together to define the properties of a solution that is larger and more complex than any of the individual patterns alone. The textual links between patterns are as important as the individual patterns, because the integration of all the patterns in specific ways is essential to the overall solution. While pattern languages are not necessarily confined to design issues, many pattern languages tend to deal with design-level domains.

The CHECKS Pattern Language of Information Integrity

Any program that accepts user input will need to separate good input from bad, and to make sure little of the latter gets recorded. This pattern language tells how to make these checks without complicating the program and compromising future flexibility.

The language has eleven patterns presented in three sections. The first section describes values as they should be captured by the user-interface and used within the domain model. The second and third sections discuss detecting and correcting mistakes, first during data entry and then after posting or publication. The patterns draw from the author's experience developing in financial software in Smalltalk. They are written as if part of a larger language and therefor may seem sketchy or incomplete. This paper is as much an experiment in the selection and linking of patterns as an attempt to communicate practical knowledge.

Section 1. First consider quantities used by the domain model…

Patterns:
1. Whole Value
2. Exceptional Value
3. Meaningless Behavior

Section 2. A person reaches through a program's interface to manipulate the domain model…

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

Patterns:
4. Echo Back
5. Visible Implication
6. Deferred Validation
7. Instant Projection
8. Hypothetical Publication

Section 3. Now consider mechanisms that address the long-term integrity of information…

Patterns:
9. Forecast Confirmation
10. Diagnostic Query

Figure 1.1: A pattern language expressing the interactions between multiple patterns to make up a complete and uniform solution.

A classic example of a pattern language is *CHECKS* [C2], developed by Ward Cunningham. **Figure 1.1** contains as an example the skeleton of CHECKS. The CHECKS pattern language is concerned with user data entry and its validation. There are 10 total patterns in CHECKS, but the links between the patterns are important as they collectively form a language. As Christopher Alexander states in his book *The Timeless Way of Building* [ALEX]: "In this network, the links between the patterns are almost as much a part of the language as the patterns themselves." For complete documentation on the CHECKS pattern language, see http://c2.com/ppr/checks.html, which is part of Ward Cunningham's *Portland Pattern Repository*.

Architecture Patterns

On a higher plane than design patterns reside architectural patterns. Architecture has been defined in different terms. Fowler [P of EAA] says that many engineers consider architecture to mean in general 'anything really important about a software system.' The "three amigos" [UML] define *architecturally significant* use cases as requirements that if removed would significantly alter the way the system would look and behave. Schmidt, et al, leading architecture patterns authorities [POSA1/2], define an architectural pattern as follows:

> "An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them." – POSA1, page 12.

To be sure, software architecture is important stuff. One thing that can be said about it is that software architecture is a higher-level construct than class and component design. Architecture supports the execution and collaboration of any number of software components on one or a group of heterogeneous computing platforms. Architecture deals more with how components access shared system resources and how they interact with other components. The interacting components may or may not be intimately aware of the existence of peer components, but they allow the platform architecture to care for such details. An architecture pattern deals with solutions to such problem domains.

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

While both the Fowler [P of EAA] and Schmidt [POSA1/2] teams expertly address architecture patterns in their respective works, the Fowler team provides the most recent and most widely applicable architecture patterns to date. Fowler's *Patterns of Enterprise Application Architecture* fills the needs of developers who target the two most popular and influential enterprise computing platforms currently at hand, J2EE and Microsoft .NET.

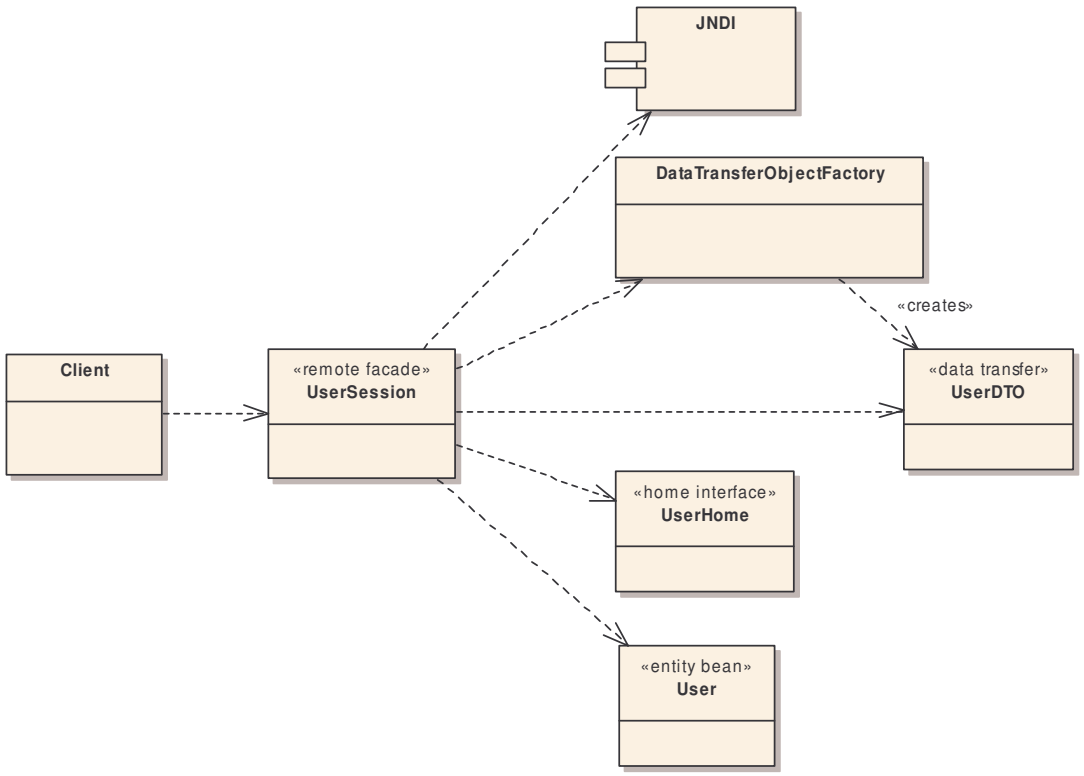
Some patterns have turned out to be somewhat hybrid entities. They tend to blend design concerns with architecture concerns. One such work is *Core J2EE Patterns* by Crupi et al [CoreJ2EE]. This work is, as noted by its developers, a blend of design and architecture patterns. However, while these patterns could be referred to as *platform patterns*, I regard them more as architecture patterns because they address solutions to developing components for the J2EE platform architecture. You will see a lot of overlap between the older [CoreJ2EE] work and Fowler's [P of EAA] when Fowler's presents J2EE solutions.

1. Remote Facade. Continuing the discussion that began with the *Facade* design pattern above, let's now expand on that to demonstrate how it can be extended to take on a more architecturally significant role. Anyone who develops .NET web services or enterprise Java components (specifically EJB Session Beans), has used the *Remote Facade* [P of EAA] pattern.

There's generally a lot going on behind the EJB session bean, for example. There's the coordination of access to data sources and usually the entity beans or other domain object mechanisms that access the database. Many complex objects must be created and managed behind the scenes. In the case of J2EE, it also generally manages database transactions. The *Remote Facade* pattern hides the details of all those complex interactions and provides the remote client with a fairly flat view of the business objects, rules, and logic used to manipulate them. It provides a service layer interface that hides the mayhem happening on the server side. And the pattern stipulates that course-grained method access is a superior approach to fine-grained. Clearly, the particular facade created for EJB access—more commonly called *Session Facade* [CoreJ2EE] in the J2EE world—is a bit more complex than the *Facade* pattern defined in *Design Patterns* [GoF].

After reviewing the following logical model, the developer of class `Client` would certainly rather use the *Remote Facade* defined by `UserSession`, than to directly use the object model living behind it. And this facade doesn't front a particularly complex object model:

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.



Further, this specific kind of facade does much more than to simplify data model access. It completely hides how the data model is implemented. As far as the client is concerned, the data model could be implemented by EJB Entity Beans or via plain old Java objects and O-R mapping. It is also responsible for transaction management and possibly for determining the data sources accessed by the specific domain objects. It is never practical to use J2EE-based domain objects directly from a remote client. You would pay an unbearable performance and data integrity penalty to do so.

Therefore, *Remote Facade* is significant far beyond design. It supplies invaluable architectural strengths and the specific enterprise platform is inadequate without it. It builds on the strengths of the original *Facade* design pattern by extending its scope and sphere of control to the subsystem level and multiple tiers of the entire enterprise application.

Integration Patterns

In the past five to seven years another *type* of pattern has emerged. Numerous solutions have been developed that allow two or more heterogeneous applications, which know nothing about the other(s), to work together. The general term used to describe such integration components is *Enterprise Application Integration*, or *EAI*. Products produced by companies such as *webMethods* and *SeeBeyond* have enabled integrators to step over large, jagged, and slippery obstacles common to integration initiatives. The underlying technologies, however, are generally available in enterprise platforms such as J2EE and .NET without purchasing additional products. Nonetheless, products such as the *webMethods Integration Platform* and *SeeBeyond's eGate Integrator* help to smooth out the way by providing the pre-built frameworks optimized for known integration problem

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

domains. Adapters for popular applications and architectures are often supported out of the box.

There are several ways for application integration to be accomplished. Until recently no definitive work had been done to capture the EAI patterns. Now the book *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* [EIP] discusses the primary ways to approach integration efforts. It names *File Transfer*, *Shared Database*, *Remote Procedure Invocation*, and *Messaging* as the four main approaches. As stated in its title, it focuses on messaging solutions as the best and most flexible and enduring approach to application integration available today. Most of its catalog of 65 patterns focuses on the various techniques useful to integrators for providing messaging solutions. Two such patterns, nearly universally useful and available to practically all integrators, are the *Point-to-Point Channel* and the *Message Bus* patterns.

1. Point-to-Point Channel. In messaging systems the best way to affect a remote procedure call from one application to another is via a point-to-point channel. Essentially, a sender object in one application communicates with a receiver object in another application via a message queue. The idea is to allow the receiver to appear to the sender as if it were an object receiving the stimulus of a simple local method invocation within the same process address space. While this could be accomplished using an actual remote method invocation technology such as DCOM or RMI, messaging is favored for its various advantages. A message queue can be durable, meaning that the "invocation" is guaranteed to occur at some future time. DCOM and RMI cannot guarantee that the intended receiver will both receive the method invocation and respond to it. (For example, even if an invocation is properly received, network problems may prevent the receiver from answering the sender, causing the entire invocation to fail.)

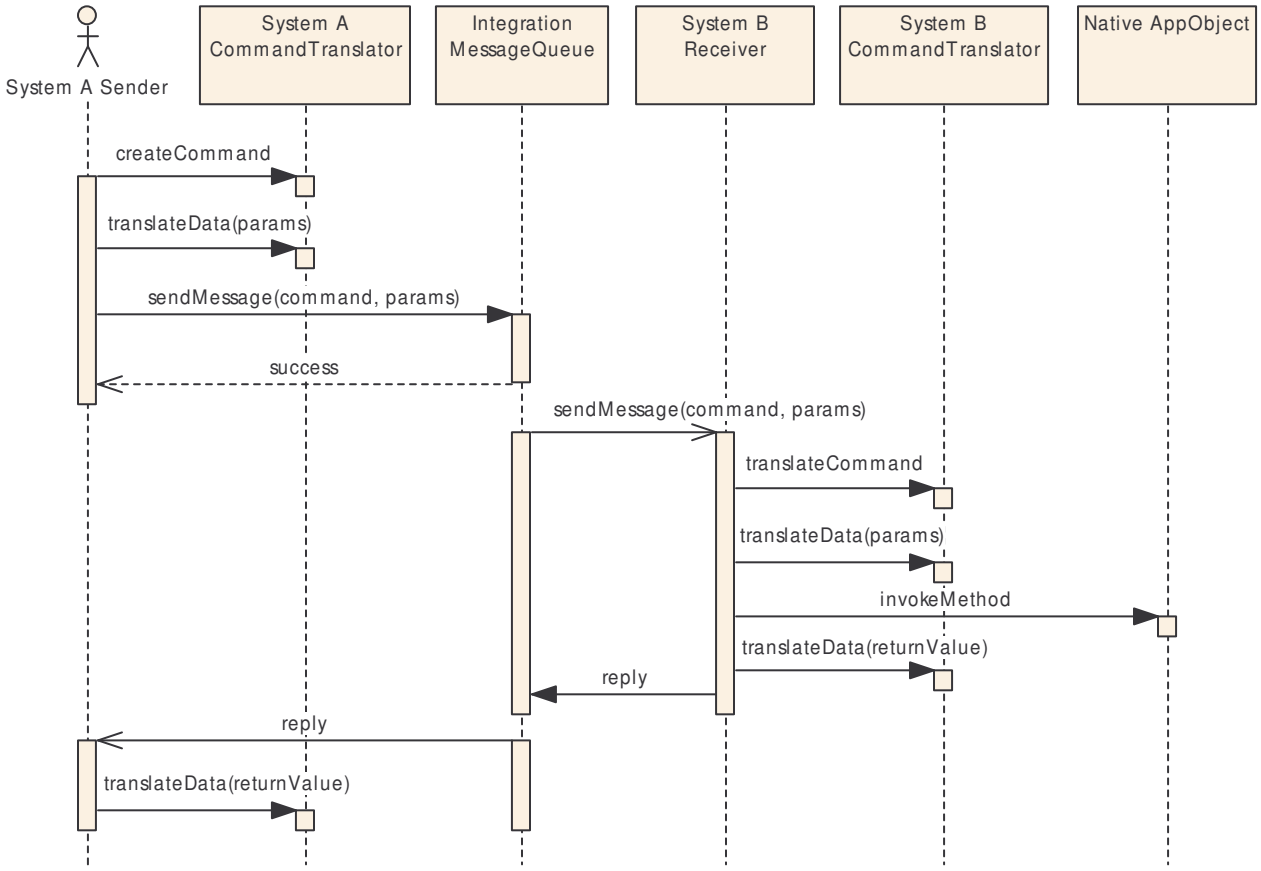
Point-to-point itself is important to the RPC nature of the pattern because it stipulates that only one receiver will be sent the message. In other words, unlike the Publish-Subscribe Channel pattern, point-to-point messages will not be broadcast to multiple listeners. Therefore, one message received and replied to effectively mimics a method invocation and return response from the receiver.

Further, the use of a messaging pattern allows for the existence of completely heterogeneous senders and receivers. The queue acts as a mechanism for commonality between all participating integrated systems. However, the message queue itself is not enough to ensure cooperative translation. Enter the *Message Bus* pattern.

2. Message Bus. Since the systems participating in the integration all use different data formats and storage mechanisms, and because the all define unique APIs, it is important that the point of integration provide the definition of commonality between the collaborators. The *Message Bus* pattern establishes a common data model and a common API for all participants in the integration to share. Therefore, the messages themselves sent on the point-to-point queue shall have agreed-upon characteristics. Some characteristics are definitive of the command set in use. In essence a command defines what each "method" name is. Further, the order and types of the "method's" parameters and how they are passed is also defined. The definition of these concerns will be suitably translated so that each and every participating system will be able to understand the commands and accompanying data.

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

All totaled, the *Point-to-Point Channel* and *Message Bus* patterns collaborate as follows:



Messaging systems, while often having the appearance of working synchronously, actually work asynchronously. Hence, darkened arrowheads in the above sequence diagram represent synchronous invocations while open arrowheads represent asynchronous message sends between the queue and the message receiver.

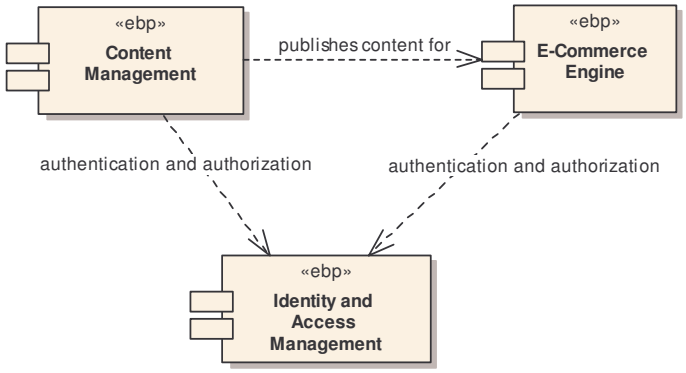
Granted, Integration Patterns definitely have an architectural influence, but they go somewhat beyond architecture as well. They actually extend the architecture of multiple enterprise applications and provide the illusion that all participating systems are one, or at least unitedly support a common business goal. In essence, integration patterns define a super architecture unto themselves, stitching together large applications, as a developer would do with smaller reusable components such as .NET/COM or EJB to make a single application.

Moving Along

This brief review of the history of software patterns provides perspective for the idea that pattern developers continue to strive for higher ground by standing on the shoulders of preceding works. I have recognized recurring solutions to huge enterprise business problem domains as suitable candidates for defining important software patterns. As a

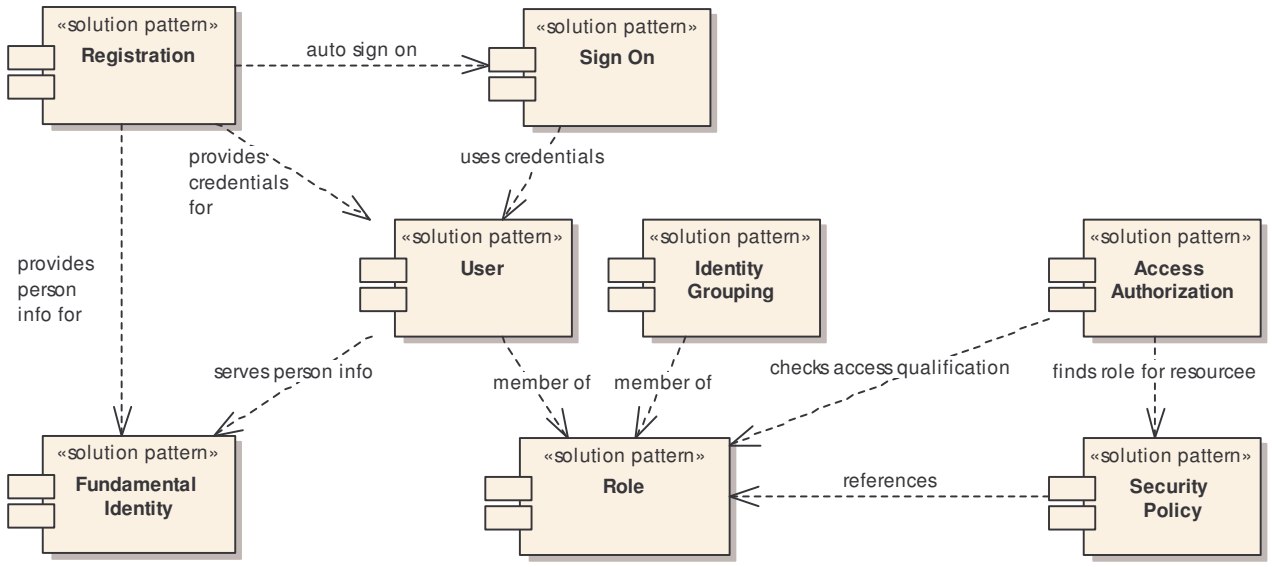
© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

reference point, the following EBP diagram illustrates a few of the patterns I have recognized and how they are used in conjunction with one another:



Here the *Content Management* EBP uses the *Identity and Access Management* EBP to authenticate users of the content management system and authorize their manipulation and publication of various kinds of content. Once the content design and development steps are completed, the content is published out to a web site. In this example, the site is one providing e-commerce behavior. This is the third EBP in the example, the *E-Commerce Engine* pattern. Like the *Content Management* EBP, the *E-Commerce Engine* uses the *Identity and Access Management* EBP to authenticate users and authorize access. However, in this case the authentication is for customers shopping on the site and authorization is for resources such as access to user accounts and payment information.

Within a given EBP we find smaller patterns. When captured and documented as a pattern language, they play together to define an integrated solution to a large business problem domain. For example, note the smaller *solution patterns* (explained below) within the overarching *Identity and Access Management* business pattern. If we drill down on the *Identity and Access Management* EBP component, we see, in part, the following:



© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

Within the *solution patterns* we can identify the use of several design and architecture patterns. The *Identity and Access Management* solution patterns use the following design and architecture patterns, to name just a few:

Design Patterns	Architecture Patterns
Abstract Factory	Business Delegate
Adapter	Domain Model
Builder	Domain Object Finder
Command	Front Controller
Decorator	Model-View-Controller/Model-2
Double Dispatch (object-oriented callback)	Remote Facade
Factory Method	Remote Procedure Invocation
Iterator	Service Controller
Singleton	Service Locator
	Value Object/Data Transfer Object
	View Helper

Clearly, design patterns, architecture patterns, and integration patterns form the foundation for EBPs. With this overview of the pattern procession in general and EBPs specifically, I now have my footing to look at the foundation for the rest of this work. Next I provide an introduction to *Enterprise Business Patterns*.

Enterprise Business Patterns Language

I now provide an explanation of how EBPs are captured into a concise document format. This presents the format of my pattern templates, both for the overarching patterns and for their collections of solution patterns. The pattern templates are essential to the precise hosting of the pattern language of each EBP. I consider an understanding of the template and language of my patterns to be essential before we set out on an analysis of how EBPs were identified.

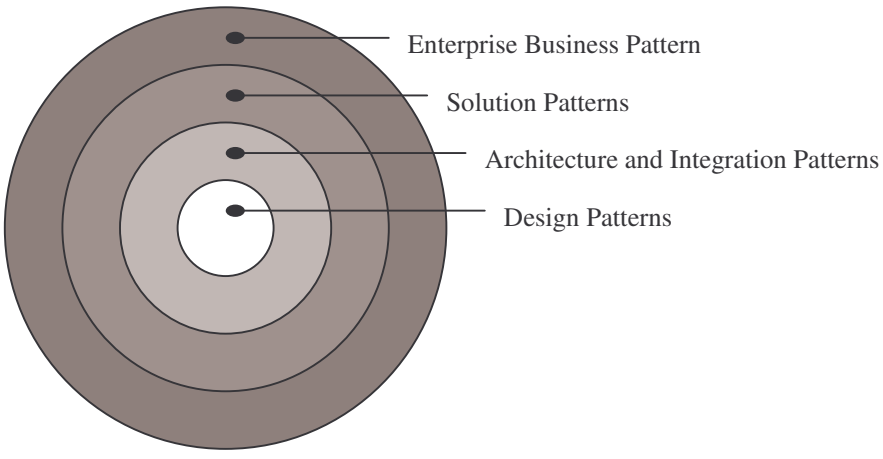


Figure 1.2: The layers of an Enterprise Business Pattern. The outer-most layer is the overarching business pattern, with successive inner layers representing solution patterns, architecture and integration patterns, down to design patterns.

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

Each EBP is a set of pattern layers. As you peel away the outer layers of an EBP you see more and more detail about the represented enterprise solution. The outermost layer of an EBP is its overarching pattern definition. This is a definition of the overall pattern—the big picture—with pattern language links to the inner layers. The links in the overarching pattern invite you to peel away the outer layers until you reach the specific solution layer of interest. Together all the layers for a pattern language that describes the EBP has a whole.

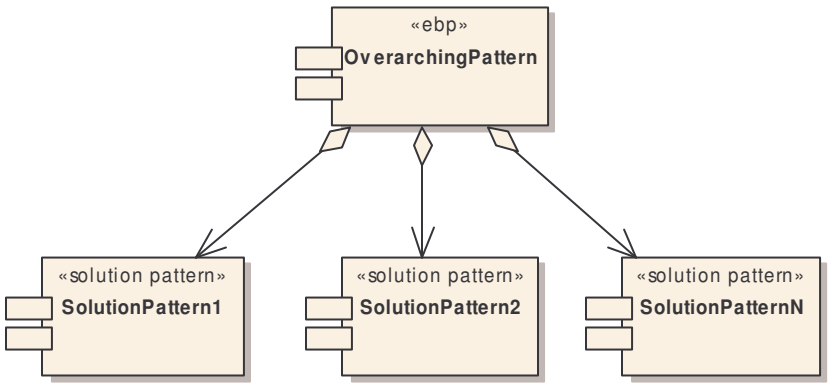
Since an EBP consumes lots of lower-level patterns, at some point the EBP fades into architecture and design. My pattern language may provide a link to the lower-level patterns, but it will not document them directly. I leave it to you to reference the architecture, integration, and design patterns in their respective works.

Figure 1.2 provides a look inside an EBP as the layers of patterns move from business solutions to component architecture and finally to class design details. The outer layer is the business solution black box, while inner layers become clearer as they reside closer to core implementation details.

The outermost layer and the one just below it are of greatest interest to us in this work. Therefore, I will concentrate on the two outermost layers. Let's look at the pattern templates for both the Enterprise Business Pattern layer and the Solution Pattern layer. I start with the outermost layer and following this I peel away to the second layer. The EBP pattern language automatically falls out from this discussion as the important template features are highlighted.

Enterprise Business Pattern Layer

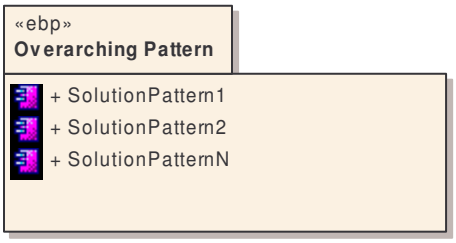
The black box Enterprise Business Pattern lies at the outermost layer of my patterns. This pattern captures the essence of the overall pattern, or its "big picture." I refer to it as the overarching pattern. An overarching pattern forms an umbrella over solution patterns giving them enterprise context, the parent of the solution patterns if you will. Consider each overarching pattern as both a pattern in its own right, as well as a pattern container. The container holds a collection of Solution Patterns (explained in the next subsection), which are the patterns that can be mixed and matched to solve the larger business problem. The Solution Patterns are referenced by links in the language of the overarching pattern:



© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

This is very much like Ward Cunningham's CHECKS pattern referenced above (*Pattern Languages*). However, while not a fundamentally different concept, my patterns are much larger than CHECKS, and provide more extensive examples too.

The main purpose of the overarching pattern is to gather together all high-level concepts of the business problem domain and organize them into one area of concise notation. This notation starts at the highest of concepts and steadily introduces the mid-level concepts, where the overarching pattern finally delegates to the corresponding solution patterns. The same abstract EBP above may also be expressed as follows, which is a shorthand version of the same overarching pattern with contained solution patterns:



Because patterns deal heavily in notation, it is appropriate that it has a consistent template. **Table 1.2** presents the layout and section descriptions of the Enterprise Business Pattern Template. You should familiarize yourself somewhat with these sections, as it will make reading the patterns a more productive experience. Next are a few additional pertinent comments about certain sections of the template.

The *Pattern Synopsis* is an implicit section; that is, the text "*Pattern Synopsis*" will not be found anywhere in the pattern. Rather, each pattern leads off with its boldface name, its brief statement, and an appropriate diagram. Architecture and design patterns will often use a static structure (class or component) diagram, or an interaction (sequence) diagram for its synopsis. These cannot generally capture the intent of an overarching enterprise business pattern because such patterns deal more with functional requirements than with simple object relationships and behaviors. Use case diagrams work best for my purpose.

Section	Description
Pattern Synopsis (implied)	The boldface pattern name, a few brief statements, and simple diagram summarizing the functional qualities of the Enterprise Business Pattern. Generally a use case diagram best captures a fairly complete functional intent of the pattern.
Background	Familiarizes the reader with the pattern's problem domain and provides some common arguments for considering the use of the pattern.
Value and Benefits	Fantails on the statements made in the *Background* section by explaining why the pattern should be used. Introduces how the pattern addresses the enterprise business problem domain. It also discusses tradeoffs in implementing certain solution strategies within the pattern.
Putting It to Work	Moves the reader from the abstract toward the concrete, making reference to solutions. Discusses integration points with existing information resources or those that must be provided. Technology options, such as those centric to a given enterprise platform, may also be highlighted.
Solution Pattern Strategies	Names the Solution Patterns employed by the overarching pattern and expresses how these interact with one another. The pattern language kicks

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

	into high gear as this section links to the associated contained patterns.
Examples	If applicable, provides some information on high-level design decisions of the reference implementation and reused components. Implementation details will be found in the same section within the Solution Patterns.
Consequences	States technical or non-technical competing forces when implementing certain solution strategies or a combination of them. Takes a risk-based approach and provides mitigation plans.
Related Patterns	Lists closely related overarching and/or Solution Patterns and how they may be used in conjunction.
Frameworks and Tools	Lists any outstanding frameworks, tools, etc., available for implementing the solution. These may be turnkey, Commercial Off The Shelf (COTS), packages that implement the entire solution or those that I used in my reference implementation.

Table 1.2: Layout and section descriptions of the overarching Enterprise Business Pattern Template.

You will notice some overlap between the *Value and Benefits* and *Putting It to Work* sections and the *Consequences* section. However, you will also note that the *Consequences* section clearly draws out tradeoffs as risks with mitigation plans, compared to those stated as prose in the other two sections.

The *Solution Pattern Strategies* section lists each Solution Pattern used in logical order of usage. In other words, as much as possible I attempt to list patterns in the order the user would interact with the pattern or in which the pattern would be architecturally executed relative to other patterns in the actual system. Appropriately, let's now look at Solution Patterns and how they are presented in more detail.

Solution Patterns Layer

The Solution Pattern layer lies one layer below the black box Enterprise Business Pattern. The Solution Patterns are each contained by a single Enterprise Business Pattern. As their name suggests, these are finer-grained patterns holding the descriptions to solving a key aspect of the overarching pattern. Here is where you will find the real reference implementations as a mapping of solution descriptions into real computing resources. You will note that Solution Patterns are more similar to architecture patterns than are the overarching patterns. They are still more course-grained that architecture patterns since they utilize at least several such patterns. But they are rubber-meeting-road patterns rather than abstract.

The pattern must be somewhat abstract at the description level because the pattern cannot be tied to a given enterprise platform. It must work equally well if implemented on J2EE, .NET, or any future concrete enterprise platform.

Section	Description
Pattern Synopsis (implied)	Same as corresponding section in overarching pattern.
Background	Same as corresponding section in overarching pattern.
Value and Benefits	Same as corresponding section in overarching pattern.
Putting It to Work	Same as corresponding section in overarching pattern.
Examples	An implementation of the Solution Pattern. An implementation will be provided for either J2EE or .NET, or both. There will generally be several useful code snippets that are embedded with text. The full source for reference implementations is available online [provide details].
Consequences	Same as corresponding section in overarching pattern.

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

Related Patterns	Same as corresponding section in overarching pattern.
Frameworks and Tools	Same as corresponding section in overarching pattern.

Table 1.3: Layout and section descriptions of the Solution Pattern Template.

The Solution Pattern Template is almost identical to that of the Enterprise Business Pattern Template. However, this one does not have a section corresponding to the overarching pattern's *Solution Patterns Strategies* section (what I am here discussing). Rather, significant architectural, integration, and design patterns are called out in the *Putting It to Work* and *Reference Implementation* sections. Therefore, lower-level patterns are made part of my pattern language by the links to them. It is your responsibility to obtain the linked-to patterns yourself.

The contents of the *Reference Implementation* section are also different. As noted in **Table 1.3**, this section contains source code with descriptive text, not design information as is found in the corresponding Enterprise Business Patterns section.

Now that you've been introduced to the pattern templates for both of my patterns of primary concern, let's see how they are presented within the overall book layout.

Using the Pattern Catalog

A pattern catalog is much like any other kind of catalog. If you want to order some new hiking boots you might browse through the "Sierra Trading Post" catalog, select some boots you like, and have them shipped to your home. In the case of a pattern catalog on Enterprise Business Patterns, you'd browse through the pages observing the running page headers. Once you saw the business pattern name and the solution pattern addressing your current problem domain, you'd read the pattern and perhaps the family of patterns in detail.

My pattern catalog is hosted in its own section of the book. This chapter and other narrative chapters are in Section I, while Section II houses the entire pattern catalog.

Each Enterprise Business Pattern starts with a "pattern home page." This page has the name of the black box overarching pattern, followed by a list of contained Solution Patterns. Take for example one of the most fundamental EBPs in the book, namely *Identity and Access Management*. **Figure 1.3** displays its pattern home page. The chapter number is first, as each EBP has its own chapter. The name of the overarching EBP is listed next. Then there is the heading *Solution Patterns* followed by a bulleted list of the names of each supporting pattern. Together these patterns form a language that describes the solution to a large problem domain in the enterprise-computing arena.

Chapter 6

Identity and Access Management

Solution Patterns

- **Access Authorization**
- **Fundamental Identity**
- **Identity Grouping**
- **Registration**
- **Role**

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

- **Security Policy**
- **Sign On**
- **User**

Figure 1.3: The pattern home page for the Identity and Access Management Enterprise Business Pattern.

In the real world you may be dealing with the implementation of an Identity and Access Management system. It may be realized via a commercial product, or you may be specifying your own. In either case you would peruse the Identity and Access Management EBP, as well as its pertinent solution patterns. Doing so would help you learn the space as a systems integrator, or it may even give you a jumpstart on analysis, design, and implementation of a custom solition.

I suggest that you familiarize yourself with the range of overarching EBPs in the catalog. Once you have done so, the catalog will become a reference guide to problem domains it addresses.

Three Huge Extraprise Business Patterns

My analysis began with an attempt to narrow down the world of enterprise computing into a set of patterns common to most electronic businesses. Since even the common business computing enterprises are vast and complex, I would not take the time to address patterns that reside on the fringes of the industry. It's impossible to address more than the core business patterns and still deliver a useful catalog in a timely manner. I, of course, welcome a more expanded treatment of patterns in the business enterprise than I am able to provide. Perhaps part of that job will fall upon you.

With that said, I in no way think that you will be bored by the patterns presented in the catalog. Few if any individual developers posses so much expertise as to understand the nooks and crannies of all of even the most common Enterprise Business Patterns. I anticipate that some readers may have an intimate understanding of one, two, or perhaps three of my EBPs, but likely not all of them. Therefore, I expect that every reader can take away an array of fresh knowledge and best practices from the pattern catalog.

I've concluded that there are three common enterprise-computing environments, which I consider to be the largest of all patterns in existence today. I call these three environments *Extraprise Business Patterns*, or XBPs. Further, I believe that the vast majority of EBPs reside within the three XBPs, and I have harvested all my patterns for the catalog from them.

Extraprise Business Pattern	Acronym	Synopsis
Business to Consumer	B2C	Also called "Self Service," this pattern defines how a consumer interacts with a business system to obtain product or information, or both. Many times this represents a Web-based storefront. By far this is the most popular enterprise business pattern, the one that is most readily thought of when analyzing the e-business technology solution space.

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

Business to Business	B2B	Sometimes call "Extended Enterprise," this pattern captures how organizations interact with one another electronically to conduct business with each other. Business partners use this pattern to exchange information in such a way that it may appear that the opposite business entity is just an extension of their own business.
Business to Enterprise	B2E	This pattern captures the resources and behavior that support the core business operations of a single, potentially large, business organization. Even if a single business organization does not have even one business partner with which it interacts electronically, it will still support at least some of the basic enterprise business patterns that exist within B2E pattern framework.

Table 1.4: A list of the three "huge" XBPs with brief synopses. Collectively these three patterns represent the entirety of the B2* Extraprise Business Pattern.

As summarized in **Table 1.4**, the three huge XBPs are as follows: The *Business-to-Consumer* or *B2C* pattern, the *Business-to-Business* or *B2B* pattern, and the *Business-to-Enterprise* or *B2E* pattern. **Figure 1.4** shows the same patterns relative to one another in an "extraprise" topography. In essence I have drilled down into a group of three patterns, which collectively represent the *B2** pattern, or *Business to Star* (or *Business to Everything*). These three huge patterns represent the modern electronic business soup to nuts. Here's why I have concluded so.

First of all, offering products or intelligent business data and information to consumers via a Web site is an essential part of modern business. This is the case whether a business is directly generating revenue from its Web offerings or not. Even if an organization is completely service oriented and it has no practical way of generating revenue from an e-commerce Web site, it will probably still offer customers and potential customers some sort of value via a Web site. The value might be through white papers that are available for download on the site, and as a list of services and how to obtain a request for proposal from the organization. Of course many companies provide demonstration software and/or open source software as free downloads from their Web sites. And there are the classic shop-for-product sites with their product catalogs and shopping carts. All of these represent the standard *Business to Consumer* or *B2C* XBP. This pattern is sometimes referred to as the "Self Service" pattern because users go to a Web site and manage the requisition of various resources of interest completely on their own. However, I believe that the B2C nomenclature is much more common and therefore expresses more clearly the intent of the pattern.

Behind the e-commerce or e-business Web site, or behind the traditional brink-and-mortar organization, lies the means of conducting your business on a day-to-day basis. There's software around employee communication and collaboration, project planning and scheduling, product inventorying, customer relationship management, accounting and financials controls, human resources, workstation, server, and network administration, just to name a few aspects of the *Business-to-Enterprise* or *B2E* XBP. Of course if the company also supports the *B2C* pattern, then their business operations likely include some form of Web content management, application delivery, and other mechanisms to manage the B2C site. The B2E pattern could well be called the "Business

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

Operations" or BOP pattern. However, *Business-to-Enterprise* or *B2E* is the common industry term.

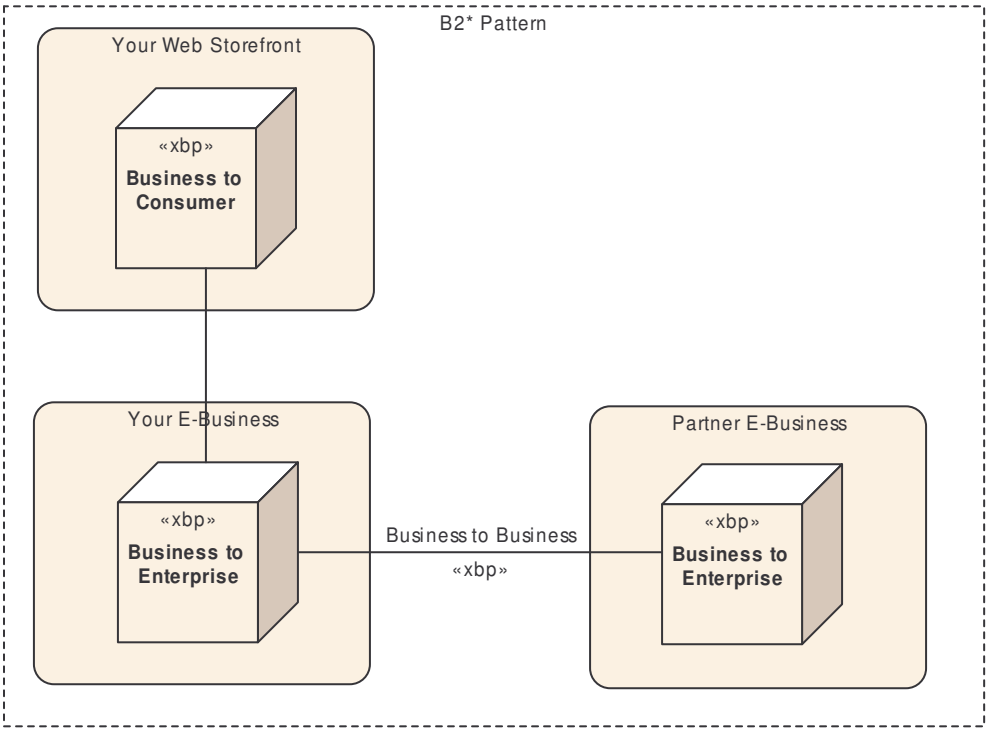


Figure 1.4: Example "extraprise" topography of the three "huge" enterprise business patterns. All three of these huge patterns live within the B2* pattern.

Whether or not a business owns and maintains the B2C pattern, it might still engage in e-business operations with an outside organization such as a partner, OEM, reseller, or supplier. In this case the Business to Business or B2B XBP would facilitate the business' interactions with the outside company entity. The interactions might include the exchange of product and sales information, contracts, software license keys, sales leads, all of which likely include highly sensitive data that must be secured. Such enabling technologies are an indispensable part of many modern e-business environments. Because this pattern may give the illusion that an organization's enterprise is larger and more extensive than their employed IT professionals actually administer, this pattern is sometimes called "Extended Enterprise."

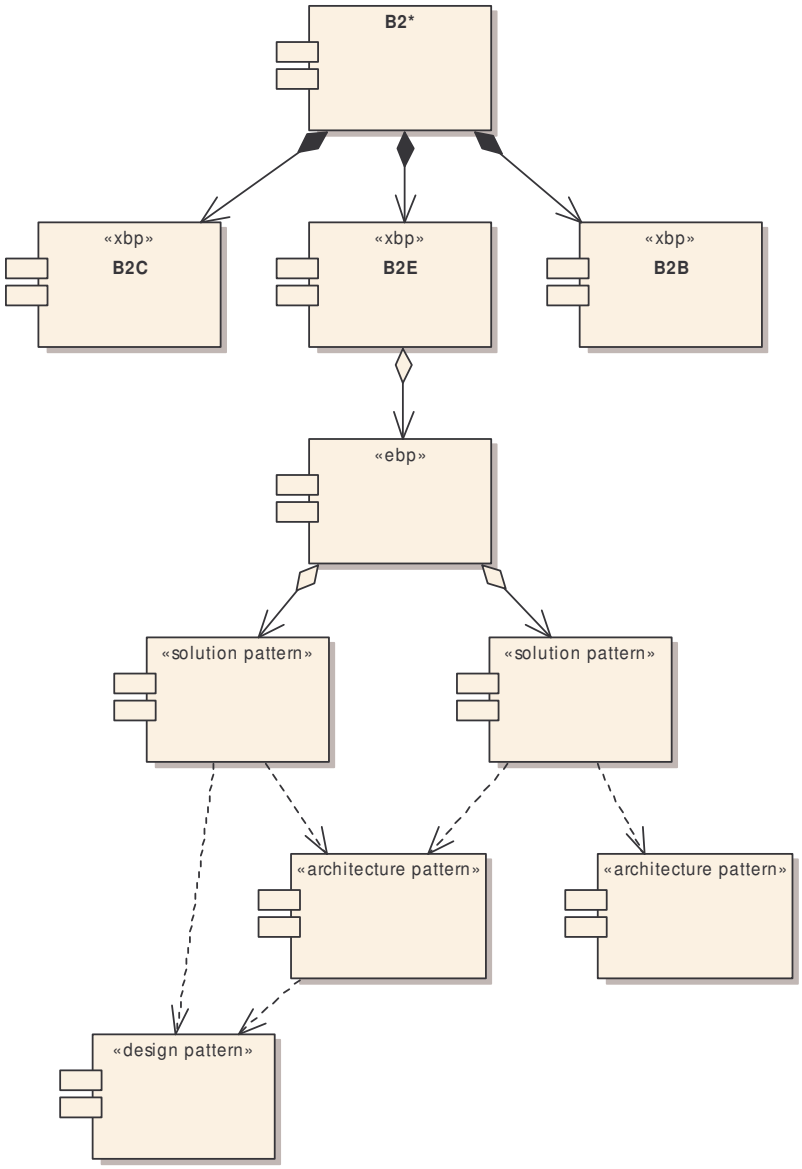
Interestingly the stereotyped nodes in **Figure 1.4** only represent two of the three XBPs, namely B2C and B2E. The diagram is a stereotyped UML deployment diagram, which represents pattern concepts as computing nodes. While there will obviously be more than one computing node in most Web sites and back office business data centers, this deployment diagram represents as many nodes as necessary within each box, but there may be more. So where's the node for B2B? There is none. I have chosen to express the B2B pattern as a stereotyped UML association between the two B2E nodes; that is, between your enterprise business and your partner's. I chose association because there is an absolute "instance" of each business on opposite sides of the relationship. Again, assume that there is in reality any number of partner businesses represented by the

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

node at the right side of the diagram. For simplicity I have limited the partners to as many nodes as necessary: one.

Within each of these three huge patterns reside the real, concrete EBPs of interest to us. A few of patterns contained inside one huge XBP also reside in one or more other the other two XBPs. I have discovered the EBPs naturally as I analyzed each XBP space. So wherever I find an EBP that has overlapping application, I simply say so in the discovery text. The discovery text thus makes a forward reference to the other applicable XBP(s), and the subsequent references are made back to the original discovery text and to the EBP itself.

I could correctly add another ring to **Figure 1.2**. It would be a darker outer ring and it would be named Extraprise Business Pattern, as it would representatively encapsulate any number of EBPs found within. Here's a static structure diagram that shows the relationships among the patterns described in this introduction:



© Copyright 2003,2004 Vaughn Vernon. All rights reserved.

In the next three chapters I drill down into the three XBPs, B2C, B2E, and B2B, to find and identify the Enterprise Business Patterns contained within them. The patterns discovered within one or more XBPs are those that comprise my pattern catalog in Section II of this book. The next three chapters are divided by XBP.

Summary

In this introduction I have reviewed the history of software patterns. I have introduced you to my analysis of the common enterprise computing environments as three huge patterns in one: B2C, B2E, and B2B all encapsulated in B2*. Within each of the three huge patterns, or XBPs, we find the real, concrete EBPs, which themselves are made up of many of the more traditional architecture, integration, and design patterns.

© Copyright 2003,2004 Vaughn Vernon. All rights reserved.