01 clarke.qrk 2/28/05 4:16 PM Page 1

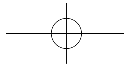
1

Introduction

You've probably picked up this book because you've heard that aspects will solve difficulties you're having with writing your object-oriented software. You've heard that aspects offer a new way to modularize your code, but you're here because you have questions like, *What are aspects? Why do I need them? Are objects obsolete?* Or perhaps you've picked up this book because you've tried programming with an aspect-oriented language and are interested in delving more deeply into the paradigm. You may have questions like, *How do I plan for aspects before design and implementation?* and *How do I design aspects so that I can better plan for implementation?* This book answers these questions and also guides you through the process of identifying and designing your aspects.

Software Development and the Object-Oriented Paradigm

Few would disagree that the object-oriented paradigm is one of the most important contributions to software development in its history. Those of us who remember developing software without objects most keenly appreciate their value. Everything to do with a "thing" is all in one place! When we want a "thing" we already have in one application to be used in another



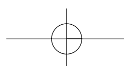
application, we can just pick it up and use it! And there are many more benefits besides.

To this day, we both remain big fans of objects. But of all the benefits associated with object-oriented development, the two of encapsulation and reuse that we hinted at above were not selected lightly. Can we really always put everything to do with a thing all in one place? Have you ever found that there was some piece of processing that did not seem to fit in any one particular class, and yet it did not feel as if it belonged in a class in its own right either? This is probably because it was too tightly coupled to behaviors in many other classes.

Think about, for example, objects that require some transaction management. It is difficult, if not impossible, to modularize all setup, communication with a transaction manager, and rolling back that may be needed to handle transactions. This is because all objects (or methods in objects) that require transaction management need to be aware that their operation is impacted by a transaction context and must behave accordingly. In other words, transaction-handling code must be placed in every object that needs it. Take a look at the code you've written in the past, and you will probably find many examples of similar (if not the same) pieces of code repeated in different places. The common, though compromised, solution to a problem like transaction management is to copy the code into the different places that need it. Code copying then results in poor modularization for much of your code and leaves you with considerable maintenance and evolution headaches. This phenomenon is also known as *scattering*, as code for a concern is scattered across multiple parts of the system.

In addition, from a reuse perspective, modules that contain code relating to many concerns are likely to be less generally useful in different situations. The phenomenon where multiple concerns are intermixed in the code is known as *tangling*.

Of course, good use of design patterns will help you encapsulate in many situations, but you will find the repetition and concern-mixing phenomena even where design patterns are well used. Ultimately, you will always encounter processing that relates to and impacts upon many portions of a system.



01 clarke.qrk 2/28/05 4:16 PM Page 3

The Case for Aspects

We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day. . . . But nothing is gained—on the contrary—by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns."

—EDSGER DIJKSTRA[1]

Aspect-oriented programming (AOP)[2] was introduced to provide a solution to the scattering and tangling described above. It is often described as liberating developers from the *hegemony of the dominant decomposition*.[3] Simply put, this means that whichever modularity you choose (objects, functions, etc.) will at some point impose unwanted constraints on your design. In the object-oriented case, the dominant decomposition is the modularity of classes, and methods. The hegemony refers to the fact that when pinned to object-orientation, developers are forced to make design decisions that lead to scattering and tangling. In some cases, developers must be able to break out of that modularity and design code that *crosscuts* an object model.

AOP allows a developer to program those crosscutting portions of a system separately from any of these structural entities. Even though in its infancy, AOP has proven to be of great use in modularizing source code and has provided a wide spectrum of benefits, from performance enhancement to more evolvable code. Aspect-oriented languages provide support for programming such crosscutting concerns, or *aspects*, in one place and then automatically propagating the behavior to the many appropriate points of execution in the code. In this way, aspects allow a developer to specify behavior that overlays an existing class model.

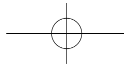
---

[1] "On the role of scientific thought," *EWD*. 477, 30 August 1974, Neuen, The Netherlands.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. "Aspect-Oriented Programming." In *ECOOP'97—Object-Oriented Programming,* 11th European Conference, LNCS 1241, pp. 220–242, 1997.

[3] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr., "N Degrees of Separation: Multidimensional Separation of Concerns," *Proc. ICSE 99*, IEEE, Los Angeles, May 1999, ACM press, pp. 107–119.

01 clarke.qrk 2/28/05 4:16 PM Page 4



However, aspects certainly should not be used as the hammer for every nail. Just as considering when to use inheritance, it is important to consider when an aspect is an appropriate choice for some functionality. Nonetheless, aspect-orientation has been shown, when used properly and appropriately, to transform necessarily hairy code into something manageable and reasonable.

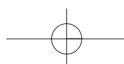
What Is an Aspect?

Simply put, an aspect is a particular kind of concern. A concern is any code related to a goal, feature, concept, or “kind” of functionality. An aspect is a concern whose functionality is triggered by other concerns, and in multiple situations. If the concern was not separated into an aspect, its functionality would have to be triggered explicitly within the code related to the other concern and so would tangle the two concerns together. Additionally, because the triggering is in multiple places, the triggers would be scattered throughout the system.

There are many examples of behavior like this—indeed, any functionality that has policies that need to be carried out in different modules of an object-oriented code base are likely candidates.

We’ve already described transaction management as an example of such code. Another typical example is logging or tracing code, since to add tracing code to a system, many locations must be modified, and every time the tracing scheme is changed, all those locations have to be altered. Another example is synchronization code, which is painful to implement, since it requires a developer to visit each method to be synchronized and add the necessary locking and unlocking functionality. Any code that may be needed in multiple places has the potential to be problematic. Having a programming model that means such code only needs to be written once and is in only one place when it requires change or deletion provides an obvious gain for the developer.

Aspects are not just a neat trick for adding logging or synchronization or other simple functionality. Such an assumption would be analogous to thinking of object-orientation as simply a means of organizing source code files. Aspects are a programmatic construct in and of themselves. Aspects



01 clarke.qrk 3/7/05 10:23 AM Page 5

provide active support, not just textual code manipulation, for separating concerns in source code. Aspects have been applied to far more complex crosscutting concerns than synchronization, logging, and tracing. For example, they have been applied in operating systems as a way to encapsulate and improve their performance.[4]

Let's look at a small example. In any banking system, almost all changes to the balance of an account affect more than one place—for example, transferring funds requires debiting one account and crediting another; interest to be credited to a customer account implies a liability to the bank's ledger (its own account); charges on a customer account imply a corresponding bonanza to the bank's ledger. These are classic examples of transactions that must complete in entirety to be valid. Figure 1–1 illustrates simplified code for these examples.

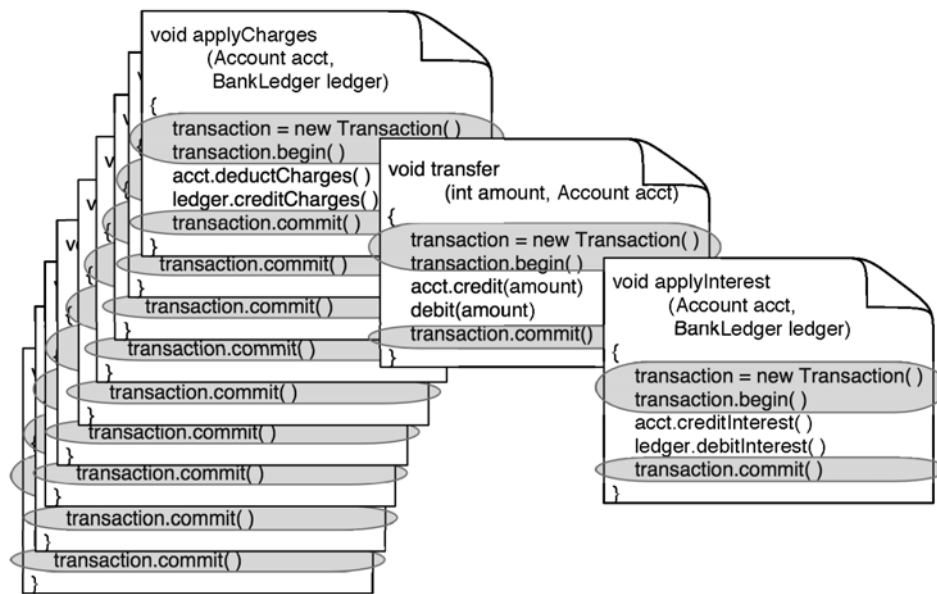
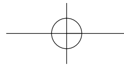


Figure 1–1 *Transaction handling occurring in multiple places.*

[4] "Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code." Yvonne Coady and Gregor Kiczales. In *Proceedings of the International Conference on Aspect-Oriented Software Development 2003.*



In Figure 1–1, each of the places where transaction handling occurs in each sequence of actions is circled. As you can see, transaction functionality is present in multiple places in the code. Because transaction handling is triggered in different situations by other concerns (such as money transfer, account updating, interest charging), transaction handling should, for this system, be implemented as an aspect.

Aspects can also be evident at earlier phases of the development lifecycle. Aspects manifest in requirements as behavior that is *described* as being triggered by many other behaviors. Requirements that described the banking system, for instance, would have mentioned that transaction handling was required for a range of activities, including managing interest, transferring money, and so on. Aspects manifest in UML designs as behavioral design elements that are triggered by other behavioral design elements in the UML models. Designs that describe the banking system would have transaction behavior tangled in the behavioral models for the banking activities.

Why Consider Aspects in Analysis and Design?

As with systems in any programming paradigm, aspect-oriented systems need to be designed with good software engineering practices in mind. The analysis and design of a system are at least as important as the implementation itself, and many professionals consider these phases to be more significant in their contribution to the success of a project as a whole.

In any development effort, it is helpful for a developer to be able to consider the structure of the final implementation at all stages of the software lifecycle. Otherwise, the developer would have to make a mental leap to get from a particular way of encoding design to another way of coding the software. In other words, developers must be able to easily map their designs to the code in order for the design to continue to make sense during the development lifecycle.

In addition to seamless traceability between the design and code, we also consider the benefits of separating aspects in the design for the design's own sake. The same benefits derived at the code level through applying aspect orientation can be derived at the design level. In the infancy of

01 clarke.qrk 2/28/05 4:16 PM Page 7

aspect orientation, developers simply used object-oriented methods and languages (such as standard UML) for designing their aspects. This proved difficult, as standard UML was not designed to provide constructs to describe aspects: Trying to design aspects using object-oriented modeling techniques proved as problematic as trying to implement aspects using objects. Without the design constructs to separate crosscutting functionality, similar difficulties in modularizing the designs occur, with similar maintenance and evolution headaches. We need special support for designing aspects, as we can then improve the design process and provide better traceability to aspect-oriented code.

A similar set of problems arises when analyzing requirements documentation to determine how to design a system. Approaches for decomposing requirements from an object-oriented perspective simply don't go far enough when trying to plan for aspect orientation. Heuristics and tools to support such an examination are helpful to the developer.

Aspects and Other Concerns

In the world of aspect-oriented language development, aspects have taken on different forms. Two are most prominent: the *asymmetric* and the *symmetric* approaches.

Asymmetric Separation

In the *asymmetric* school of thought, aspects are separate from the core functionality of a program. Aspects are encoded as events that are triggered before, after, or as a replacement for certain other events, or in certain situations are located in the core. They describe additional dynamic behavior of a system that will have an effect on the core functionality. In a distributed system, for instance, there may be a collection of domain-specific objects that need to be managed in terms of distribution, synchronization, and transaction management.

The core contains the structure and behavior relevant to the domain functionality of the system. Separate from that core are aspects like the distribution of the objects in the system, the synchronization scheme

01 clarke.qrk 2/28/05 4:16 PM Page 8

associated with the methods belonging to those objects, and the wrapping of a set of operations into a single transaction. These are described in a separate module (each in its own aspect) and are invoked at certain strategic points in the execution of the core of the program. For instance, before certain methods are executed, the synchronization aspect may be used. Or, transaction-handling processing is initiated before and after the set of operations that make up a single transaction. In our banking example, as illustrated in Figure 1–2, the *core* is the set of banking-specific classes, and the *aspect* is a separate transaction handling entity.

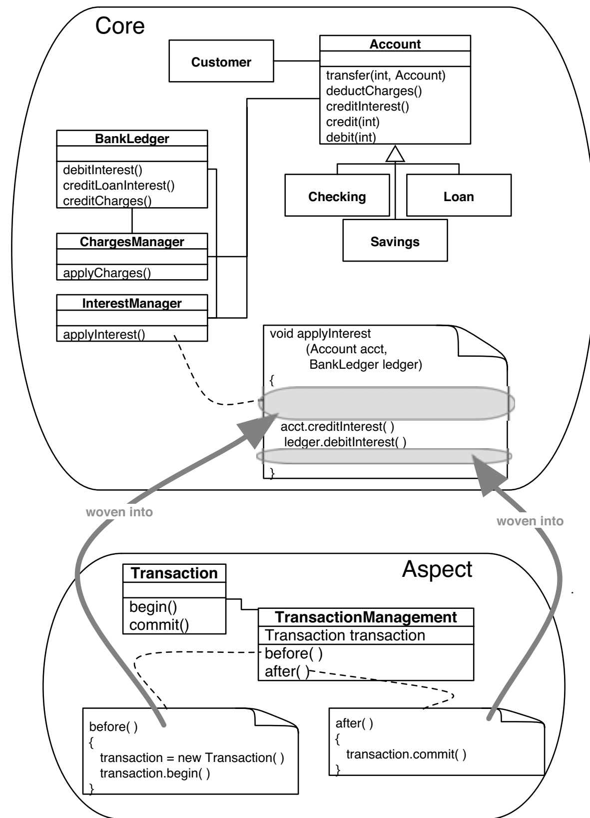


Figure 1–2 *Aspects and core in the asymmetric paradigm.*

01 clarke.qrk 2/28/05 4:16 PM Page 9

At a conceptual level, aspects have two important properties in this scheme. First, the aspect will only be triggered because of some execution in the core—for example, transaction handling is required only when changes are made to the balances of accounts. Second, the aspect is highly likely to be triggered in many parts of the system—it really is not generally all that useful to separate design/code into an aspect if it is executed in only one part of a system.

Table 1–1 provides definitions of the terms as typically used in this paradigm.

Table 1–1 *Definition of Terms in the Asymmetric Separation Paradigm*

Term	Description
Crosscutting	Concern behavior that is triggered in multiple situations.
Advice	The triggered behavior.
Aspect	The encapsulation of the advice and the specification of where the advice is triggered.
Core	The traditional object-oriented part of the system to which aspects are applied.
Joinpoint	A possible execution point that triggers advice.
Pointcut	A predicate that can determine, for a given joinpoint, whether it is matched by the predicate
Weaving	Applying the advice to the core at the joinpoints that match the pointcut statements in the aspects.

Symmetric Separation

In the symmetric separation model, in addition to the modularization of aspects, the core as described above is also analyzed for further modularization. Consider the core banking system from Figure 1–1. This example illustrates a small amount of real banking functionality with three basic features or concerns: transferring funds between two accounts, applying charges to an account, and applying interest to an account. In a real banking system, not only are there many other features, but these three features

01 clarke.qrk 2/28/05 4:16 PM Page 10

are subject to a significant number of banking rules. These rules depend on many different properties, such as the type of account, type of customer, legal and tax concerns (national/EU or state/federal, etc.), and so on. Figure 1–3 illustrates how it is possible that each feature is likely to result in many other methods that are likely to have an impact across the core set of banking classes.

Figure 1–3 necessarily depicts just a small proportion of the possible impact of each of these three features on the classes. As you can see, there are multiple methods in the account classes that handle the business rules. The `Checking`, `Savings`, and `Loan` classes all have many methods related to checking charges and checking interest. It may appear as if those could just be swept into the `Account` superclass. Unfortunately, that's not the case. Each of the accounts handles those rules very differently, and so the functionality has to be present in each of them.

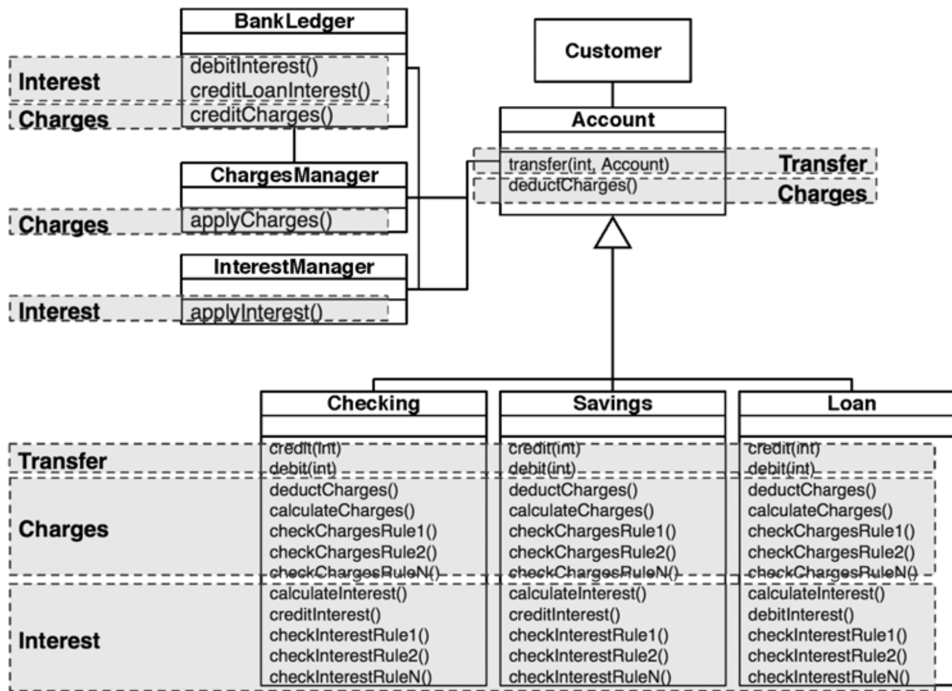


Figure 1–3 *Multiple concerns in the asymmetric core.*

01 clarke.qrk 2/28/05 4:16 PM Page 11

We hope you can visualize the potential real impact of including the full behavior not just of these three features, but also of all banking features on the core system. In the symmetric paradigm, different features of the system can be modularized into separate programs, as illustrated at the design level in Figure 1–4.

An entire system is therefore made up of bits of separate functionality that could be thought of as features or concerns. These can then be recombined in various ways to form a functioning whole. With this approach, a set of distributed objects would be formed by composing bits of basic object functionality together with bits of distribution functionality and synchronization functionality and transaction functionality.

At first glance, the duplication present in the symmetric approach looks as if it actually worsens scattering. For instance, all of the concerns except for the Transaction Management concern in Figure 1–4 have an `Account` class as well as a `Checking` class, a `Savings` class, and a `Loan` class. This duplication is required in the symmetric approach in order to provide a complete *view* of the system from the perspective of a particular concern. The completeness of the view enhances separate understandability of a particular concern in the system.[5] This understandability is achieved through increases in *locality*: Only and all relevant functionality for a concern is present within the concern module. Concern maintainability is also considered enhanced because of this functional locality. It is true that altering every method belonging to a class would require visiting many concerns, but since maintenance efforts are often performed to address particular concerns, the locality of all concern functionality within an identifiable group of modules is actually a help to system maintainability.

Of course, the symmetric approach can be applied on a continuum. It is unnecessary to keep minute concerns separate, just as it is unnecessary to bundle all the core concerns together. This spectrum is one that the developer is encouraged to explore, as each extreme has its own trade-offs and advantages.

[5] *Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition.* W. H. Harrison, H. L. Ossher, P. L. Tarr, IBM Research Division, Thomas J. Watson Research Center. RC22685 (W0212-147) December 30, 2002.

01 clarke.qrk 2/28/05 4:16 PM Page 12

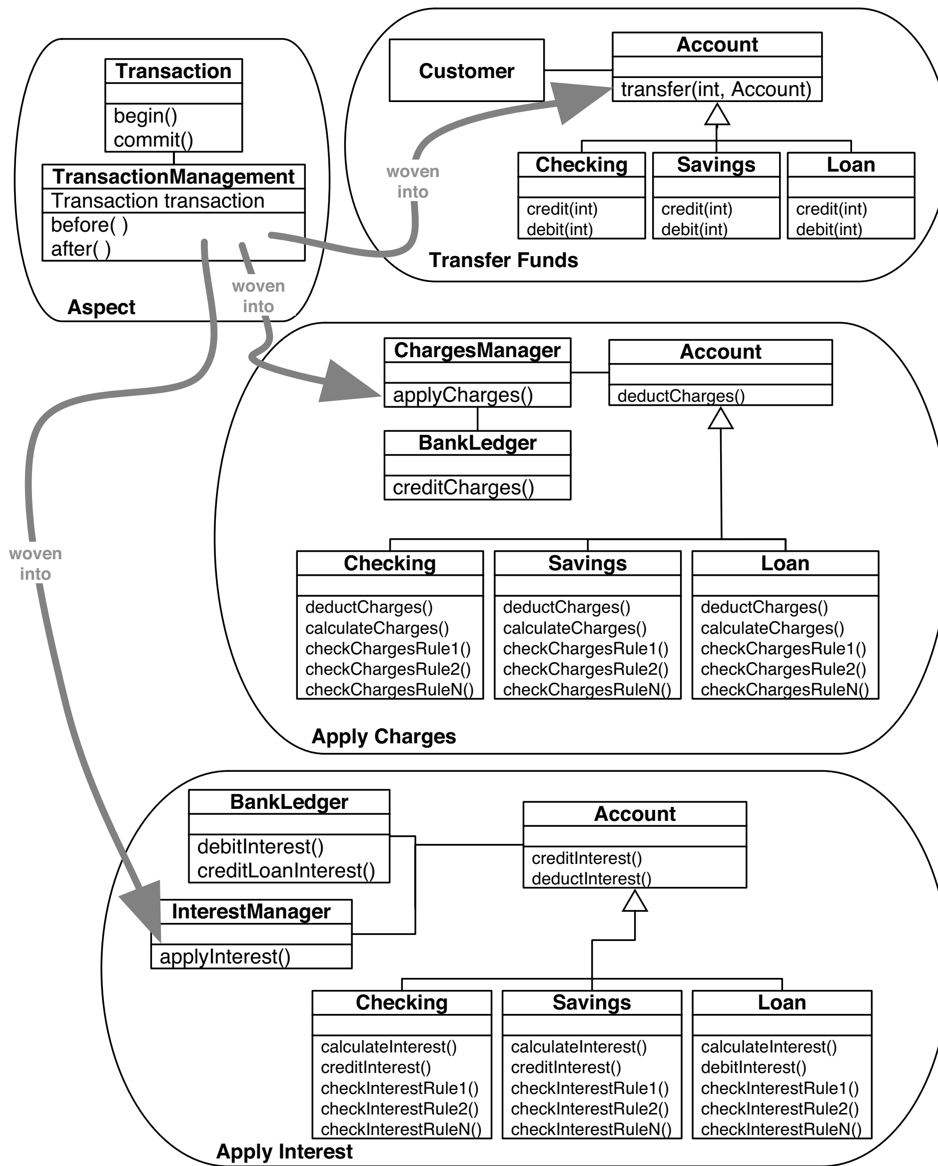


Figure 1–4 *Separation of different features in the symmetric paradigm.*

01 clarke.qrk 2/28/05 4:16 PM Page 13

The terms for the symmetric approach are given in Table 1–2. Notice that the terminology for this approach is different from the asymmetrical approach described above. Crosscutting, for instance, takes a wider stance: that it is widely triggered functionality, but also that structure and behavior (concepts) related to a particular concern are scattered throughout the system.

Table 1–2 *Definition of Terms in the Symmetric Separation Paradigm*

Term	Definition
Concern	Some "kind" of functionality in your system. This could be a feature or a type of processing.
Crosscutting	A concern triggered in multiple situations or whose structure and behavior are scattered across the code base and tangled with code related to other concerns.
Composition	Combining the separately implemented concerns to form a functioning system.

The Theme Approach

In this book, we cover how to identify aspects in a set of requirements and how to model them in UML style designs. The methodology we introduce here is the *Theme* approach to analysis and design. The terminology we use is a hybrid of the symmetrical and asymmetrical paradigms. The terminology is described in Table 1–3. Grayed-out cells in the table indicate that the term is not used in the particular paradigm.

What Is a Theme?

The word *theme* should not be considered a synonym for *aspect*. Themes are more general than aspects and more closely encompass concerns as described above for the symmetric approach. We view each piece of functionality or aspect or concern a developer might have as a separate theme to be catered to in the system. You can see in Table 1–3 that a concern is described as "Some 'kind' of functionality in your system. This could be a feature or a type of processing," and a theme is described as "An encapsulation of a concern."

01 clarke.qrk 2/28/05 4:16 PM Page 14

Table 1–3 *Definition of Terms as Used in This Book*

Term	Theme Approach Definition	Asymmetric Separation Definition	Symmetric Separation Definition
Concern	Some "kind" of functionality in your system. This could be a feature or a type of processing.		
Theme	An encapsulation of a concern.		
Crosscutting	Triggered in multiple situations.		Triggered or located in multiple places.
Concern Scattering	When the behavior related to a concern is found in more than one class		One kind of crosscutting.
Crosscutting theme	A theme that has some behavior triggered by other themes in multiple situations.		
Concept sharing theme	A theme that describes domain concepts also described in another theme, though from its own perspective. Solves difficulties associated with concern scattering.		One kind of concern.
Aspect	A crosscutting theme parameterized to handle the triggers for its behavior. Solves crosscutting.		A concern, whether triggered or not.
Base	Base theme: the theme that triggers any aspect's behavior The base: themes that trigger a particular aspect's behavior and themes that are not aspects.	The core: traditional OO design into which aspects are woven.	
Composition	Combining themes based on a composition relationship.	Weaving an aspect into the core.	Combining concerns based on a composition relationship
Merging	Merging themes that share concepts and composing base and aspect themes.		One kind of composition.
Binding	Specifying the triggers of the aspect from the base.	Weaving an aspect into the core.	

01 clarke.qrk 2/28/05 4:16 PM Page 15

For example, the three banking-related features and the transaction-handling component in Figure 1–4 are four separate themes. Themes can encapsulate aspect behavior (behavior that is triggered in multiple situations), but can also encapsulate non-aspect concern functionality.

At the requirements level, a theme is a subset of the responsibilities described in a set of requirements. At the design level, themes include the structure and behavior needed to carry out their requirements-level responsibilities.

Relationships Between Themes

Themes may be related to each other in the same way as requirements or features or aspects are related to other parts of the system. Such relationships may cause overlaps in the themes. There are two ways in which themes can relate: by sharing concepts and by crosscutting.

Concept Sharing

The first category of relationship is *concept sharing*, where different themes have design elements that represent the same core concepts in the domain (see Figure 1–5). Take, for example, the **transfer funds**, **apply charges,** and **apply interest** features in Figure 1–4. Each of these three features works with `Account`—all three work with `Checking` and `Savings` accounts, while two

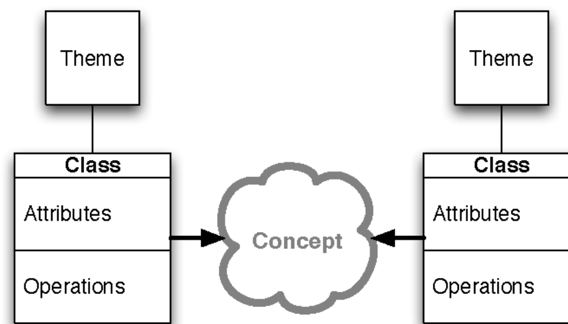
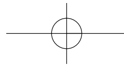


Figure 1–5 *Themes related by concept sharing.*

01 clarke.qrk 2/28/05 4:16 PM Page 16



of them also work with `Loan` accounts. Each theme contains specifications for those same concepts designed from the perspective of the theme. Another way of looking at concept sharing is to think of it, at some level, as "structure" sharing. Strictly speaking, though, we don't consider that themes actually "share" structure because each theme will have its own version as appropriate to the feature under design. Encapsulation in this manner has the benefit of locality, where only and all relevant functionality for a concern is present in a theme.

Concept sharing is one category of crosscutting in the symmetric separation paradigm, as is shown in Table 1–3. Concept sharing is not discussed in the asymmetric separation paradigm. Encapsulation in this manner has the benefit of locality, where only and all relevant functionality for a concern is present in a theme.

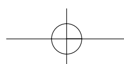
Crosscutting

The second category of relationship is the asymmetrical *crosscutting*, where behavior in one theme is triggered by behavior in other themes. Transaction handling from Figure 1–4 is an example of such a theme. Table 1–3 shows that this definition is shared with the asymmetrical separation approach and is considered "one kind of crosscutting" in the symmetrical separation approach.

Throughout the book, we use the terms *base theme*, *crosscutting theme*, and *aspect theme*. Aspect and crosscutting themes are used synonymously and are always themes that have behavior triggered in tandem with behavior in other themes. Aspects in the Theme approach are the same as aspects in the asymmetric separation approach.

Base themes are the themes that trigger aspect themes. They might be themes that share concepts with other themes, and they might be aspects themselves and have their own base. We also sometimes talk about a base that is the result of a composition of other themes to which we then apply an aspect (see Figure 1–6).

As seen in Table 1–3, we don't use the term *core* in relation to themes, since we consider a core, in the sense of the asymmetric separation paradigm, to be made up of multiple bases. In this we adhere to the symmetrical approach.



01 clarke.qrk 2/28/05 4:16 PM Page 17

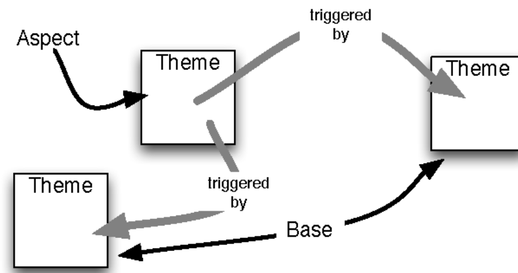


Figure 1–6 *Themes related by crosscutting: The base theme triggers the aspect theme.*

Applying the Theme Approach

The Theme approach is made up of two portions: Theme/Doc, which is a set of heuristics for analysis of software requirements documentation, and Theme/UML, which is a way to write themes (both aspects and base) as UML. In this section, we present a high-level overview of the activities involved in applying the Theme approach. These activities are depicted in Figure 1–7.

Analyzing Requirements with Theme/Doc

At the requirements level, themes are "responsible" for certain functionality described in the requirements document. Themes at this point are, essentially, named subsets of requirements.[6]

Theme/Doc (which stands for Themes in Documentation) is the part of the Theme approach that assists in identifying themes in requirements documents. It also provides heuristics for identifying which of those themes are crosscutting, or aspects.

[6] Requirements can take any form as long as they contain text. For instance, they can be entire use cases or sentences within use cases. In this book, we talk about them as though they're individual sentences in an informally written document.

01 clarke.qrk 2/28/05 4:16 PM Page 18

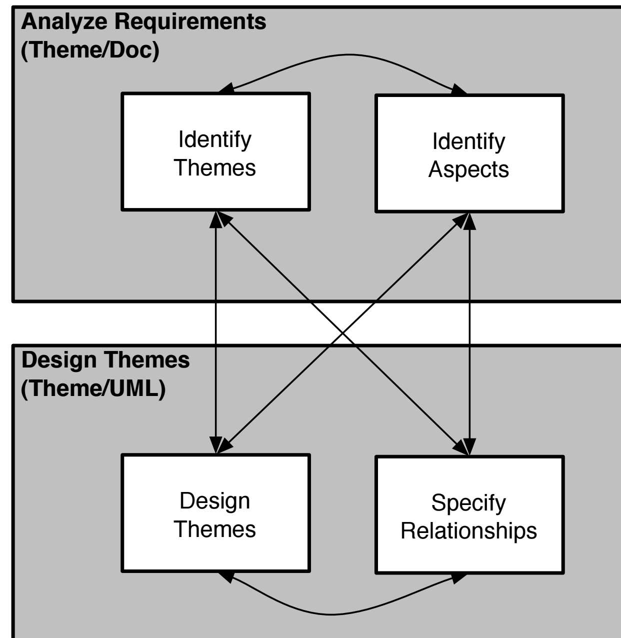


Figure 1–7 *High-level view of the Theme approach activities.*

The Theme/Doc analysis process has two main activities: (1) identifying the main themes in your system, and (2) determining whether the responsibilities of a certain theme mean that it should be modeled as an aspect. You interleave these two activities (theme identification and aspect identification) to plan for design or accommodate changes as your system evolves.

Starting Out

The process begins with determining an initial set of themes. These might be just a set of concerns you think seem important at first glance. Or, if you've already applied a requirements analysis approach and have a set of features or concerns readily in mind, then using those might make sense.

The Theme/Doc tool provides graphical depictions of relationships between requirements and themes. Figure 1–8 shows a stylized Theme/Doc view. You can see in the figure that diamonds represent themes and rounded boxes show the text of a requirement. If a requirement's text

01 clarke.qrk 2/28/05 4:16 PM Page 19

mentions a theme's name (or any term considered its synonym), it is linked to that theme. Unless at some point later in the process you sever the link, the theme is responsible for that requirement. For instance, both requirements attached to the transfer theme in Figure 1–8 mention transfer, so they are linked to it in the view.

Theme Identification

The *theme identification* activity involves iterating over the themes until you have a set you think makes sense. This process involves looking at the responsibilities of each theme to see whether, together, they represent a coherent set of behavior.

Aspect Identification

To identify aspects using the Theme approach, you look for tangling in the requirements. Two themes are tangled if they *share a requirement*. You can see an example of a shared requirement in Figure 1–8. The shared requirement mentions both **transaction handling** and **transfers**.

If two concerns are described together in a requirement, their responsibilities may be tangled. However, identifying tangling alone is not enough to

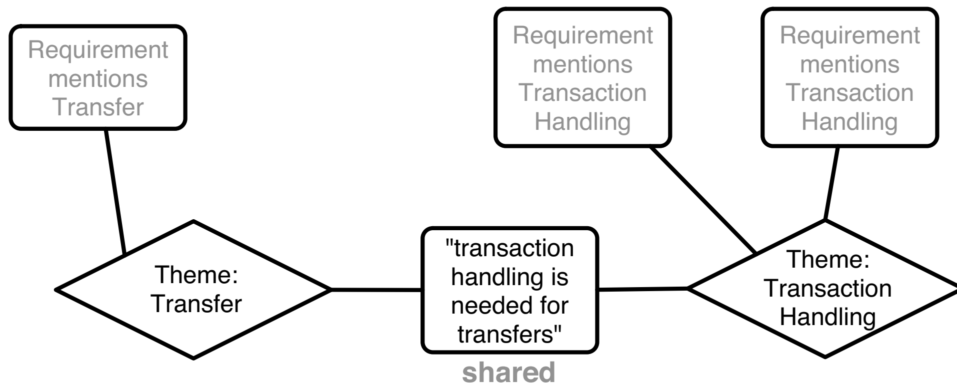


Figure 1–8 *Stylized Theme/Doc relationship view.*

identify an aspect. To locate an aspect, you ask several *aspect identification questions* about the shared requirement:

1. *Can the requirement be split up to isolate themes?* If it can, you could rewrite the requirement to better divide responsibilities between themes. The shared requirement in Figure 1–8 could not be split up and still remain an actual sentence.

2. *Is one theme dominant in the requirement?* If so, then the dominant theme should likely be responsible for that requirement rather than the requirement being shared between themes. *Transaction handling* dominates the shared requirement in Figure 1–8, since the requirement mainly talks about when transaction handling is needed.

3. *Is behavior of the dominant theme triggered by the other themes mentioned in the requirement?* If so, then you have identified a trigger relationship between two themes. **Transaction handling** behavior is triggered by the initiation of a **transfer**: When a transfer occurs, transaction handling is needed and so triggered.

4. *Is the dominant theme triggered in multiple situations?* If, across the requirements, the dominant theme is described as triggered in multiple situations, then it is crosscutting. The dominant theme becomes the aspect, and the triggering themes become the base. **Transaction handling** is needed in different situations (for transfers, for adding interest, etc.). **Transaction handling** is an aspect.

As mentioned above, theme and aspect identification activities are interleaved, as newly split requirements give way to new themes and as new themes give rise to newly shared requirements. If, as they often are, the requirements are live and changing, then new themes and responsibilities for themes will arise after you've moved on to design and implementation. Design and implementation may also cause you to revisit the choices you made about theme responsibilities, so you would come back and shift things around.

01 clarke.qrk 2/28/05 4:16 PM Page 21

Designing Themes with Theme/UML

Theme/UML allows separate design models for each of the themes identified in the requirements. It is grounded in some important steps of aspect-oriented software development: modularize, relate, and compose.

Design the Themes

From a modularization perspective, the themes that were identified using Theme/Doc can be designed separately regardless of whether one theme crosscuts another or whether there are other kinds of overlaps in the themes. Examples of overlaps other than crosscutting might be when some core domain concept (perhaps associated with particular classes), such as loan account or savings account, is relevant for multiple themes. When designing the different themes, you need not be concerned with overlaps.

You can see in Figure 1–9 that each of the banking concerns described earlier is captured in its own theme.

You will use almost entirely standard UML to design each theme from its own perspective. All the classes and methods pertinent to each of those concerns would be designed within the themes, essentially as depicted in

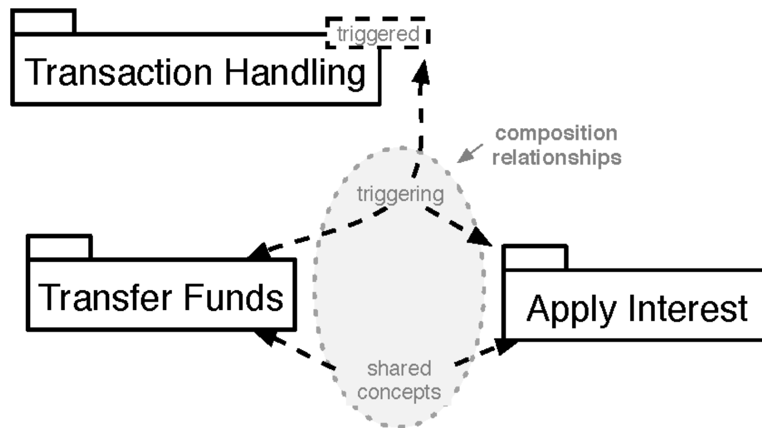


Figure 1–9 *Stylized Theme/UML composition.*

Figure 1–4. This is a considerable strength of the Theme approach. When you want to work with a single feature or concern, you have just one place to go—the theme. Within the theme, not only is every design element relevant for the concern, you can also be sure that you don't have to go anywhere else to find other relevant design elements. Aspect themes use a little nonstandard UML to capture parameterization of the behavior that is triggered by a base theme (shown as "triggered" in Figure 1–9).

Specify the Relationships

Where you will primarily notice some differences with the standard UML is in the area of specifying the relationships between the themes and in the composition capabilities that Theme/UML defines. In all aspect-oriented approaches, there must be a way to designate how the modularized concerns relate to the rest of the system. To provide this capability, Theme/UML has defined a new kind of relationship, called a *composition relationship,* that allows you to identify those parts of the theme designs that relate to each other and therefore should be *composed*. For themes that crosscut others, this means identifying when and where in those other themes the additional dynamic behavior should occur (shown as triggering in Figure 1–9). For other kinds of overlaps, this means identifying elements in the theme designs that correspond to each other and saying how they should be integrated (shown as shared concepts in Figure 1–9).

Theme/UML also provides semantics for model composition based on the composition-relationship specification. We think of this as a verification step to allow you to have a look at the overall system design, including the composition specification, to ensure that it makes sense. Of course, the composed design will have all the poor modularization that we've been trying to avoid! The developer can then take a step back and revisit the separate Theme/UML models with a view to implementation.

Theme: Symmetric or Asymmetric?

The Theme approach more closely aligns with the symmetric approach to system decomposition, since themes are individual concerns regardless of whether they are aspects or separated concerns that would be located in the core. However, the terms crosscutting and aspect are defined as in the asymmetric paradigm: as functionality that is triggered in multiple situations.

01 clarke.qrk 2/28/05 4:16 PM Page 23

As we mentioned above, however, the symmetric decomposition described in this book does not dictate unnecessarily tiny concerns. In this book, we look at several examples: Some that have many concerns and some (particularly the final case study of the book) that have a more solid base. The important thing about the Theme approach is that you should choose the degree of separation that is right for you and your situation. The heuristics for aspect identification described above are the same regardless of whether you intend to implement your system using a symmetrical or an asymmetrical decomposition approach. Similarly, Theme/UML models can encompass functionality related to either a fine-grained concern or to the entire core functionality of your system.

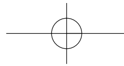
Fitting Theme into Your Existing Development Process

The analysis and design activities described in the Theme approach can be split up and molded to fit into whichever development process you are happiest using. Later, in Chapter 3, "The Theme Approach," we briefly outline how that might work. Fitting the Theme approach into processes such as the iterative and waterfall approaches is quite straightforward. Figuring out how to work them into the family of agile processes[7] deserves further discussion. In Chapter 3, we go into some of the processes in more detail. Here, however, we discuss analysis and design in the context of agile processes from a high-level point of view.

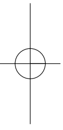
The use of Theme/UML in agile processes mirrors the relationship between standard UML and agile processes, which is a much larger question. After sifting through rhetoric from experts on both sides (UML is crucial versus UML is useless), we found words of wisdom that struck many chords with us—Martin Fowler's paper entitled "Is Design Dead?"[8] which we highly recommend.

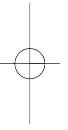
[7] Agile processes attempt to dispense with "heavyweight" development processes and focus on a lightweight approach that values immediate results over lengthy planning.

[8] Available from http//:www.martinfowler.com.



What we have taken from the "Is Design Dead?" article is that if you are the type of developer to find diagrams helpful, then you will continue to do so with agile processes, and if you are not, then you won't. In addition, it is especially important to recognize that diagrams can "actually cause harm" and therefore should be used judiciously. The use of diagrams has potential value from a number of perspectives: communication; as a means to explore a design before you start coding it; and documentation both ongoing and for handover situations. From a Theme approach perspective, we paraphrase or steal directly from "Is Design Dead?" in the following list of recommendations for managing both Theme/Doc and Theme/UML diagrams in an agile process environment:



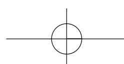


- Capture the interesting analysis and design decisions in the diagrams. Not every class in every theme may be interesting and not every attribute or method in an interesting class may be interesting. Do, however, consider capturing all the composition relationships and crosscutting behaviors during your exploration phase.

- Keep the analysis diagrams and designs only as long as they are useful. Don't be afraid to throw away diagrams that have become outdated through refactoring or other reasons—they were useful for a time, but you can let them go. Since Theme/Doc views are automatically regenerated, generate new ones when changes occur.

- When you are coding, if you think diagrams would be useful for communication outside the immediate team or for capturing a point in time, then re-create them for that purpose.

- Even if you fall into the category of people who don't really find diagrams useful, bear in mind that there are others who do, and so when you are handing over the system to other people, then diagrams that represent the current state of the code may be appreciated.



01 clarke.qrk 2/28/05 4:16 PM Page 25

What About Implementation?

The focus of this book is on aspect-oriented analysis of requirements and aspect-oriented design. Other books may be of better service if you're looking for detailed information about implementation in aspect-oriented languages. However, we spend a chapter delving into how you might follow to code from the analysis and design process described above.

There are two ways in which developers may implement a system designed using the approach presented here. As a developer, you may choose to carry through to implementation the separation of the design-level themes. Of course, you need to use a programming model that supports theme-based modularization and composition as designed using Theme/UML. Aspect-oriented languages provide such a model. Taking this approach yields the traceability benefits that we previously discussed as an advantage to using the Theme approach.

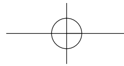
Alternatively, you may implement the composed design using an object-oriented language. It is likely that this approach would only be taken where there is a reluctance to use an aspect-oriented programming language. The resulting code will display the modularization characteristics, in which the themes are integrated into the straight object model, that aspect-oriented programming has been designed to avoid.

In either case, there is language support as well as development environments that would be helpful in implementation. Later in this book, we briefly review how to make the translation from the theme models to implementation languages, covering how to translate theme models into a selection of aspect-oriented languages.

Summary

In this chapter, we described the basic motivation for considering aspects early in the software lifecycle. We follow the symmetric approach to modularization, where features or concerns are each separated. We use the term "aspect" to refer to features or concerns that crosscut other features or concerns. We refer to all of these as themes.

01 clarke.qrk 2/28/05 4:16 PM Page 26



In this chapter we also introduced the Theme approach, which consists of two parts: Theme/Doc, which relates to consideration of themes in requirements, and Theme/UML, which allows description of themes at design. Next, we look more closely at the motivation for the use of themes by applying the best object-oriented practices to solve a design problem and seeing where those practices fall short.