feature

aspect-oriented programming

Aspect-Oriented Development with Stratified Frameworks

Colin Atkinson and Thomas Kühne, *University of Kaiserslautern*

Separation-of-concerns technologies are key to improving the maintainability and adaptability of software artifacts.[1] These technologies identify loosely coupled, modular, and reusable units of description from which developers can generate tailored software systems with minimal effort. Aspect-oriented programming provides perhaps the most explicit application of the separation-of-concerns tenet.

Aspect-oriented development is a "separation-of-concerns" technology that promises to improve productivity and product quality in software development projects. Architecture stratification combines the strengths of component-based frameworks and model-driven architectures to support aspect-oriented development.

AOP aims to separate concerns that crosscut a particular decomposition strategy, such as object orientation, and traditionally uses weaving techniques[2] to recombine them. It is based on the recognition that no matter what criteria are used to modularize a system, there will always be system properties that "cut across" module boundaries. An *aspect* is a software artifact that addresses one system concern and must be combined (ideally through an automated process like weaving) with the base modules and other aspects to yield a complete application.

Other leading development paradigms can also be understood as separation-of-concerns technologies. Component-based development, for example, separates core functionality from connectivity and uses an assembly metaphor to create complete systems from prefabricated parts, while framework-based development separates domain commonalities from application variabilities and uses an instantiation metaphor to create individual applications. The Object Management Group's (www.omg.org) more recent model-driven architecture (MDA)[3] approach, on the other hand, separates platform-independent architectural abstractions from technology-specific realizations of these abstractions, and uses a refinement or mapping metaphor to generate concrete, executable systems.

These technologies might appear to use incompatible criteria for modularizing the software artifact descriptions. However, closer examination reveals that this is not true. Although some of the specific mechanisms used in these paradigms might be incompatible, the underlying separation-of-concerns strategies they employ are not. On the contrary, because their modularization criteria are essentially orthogonal, there is a

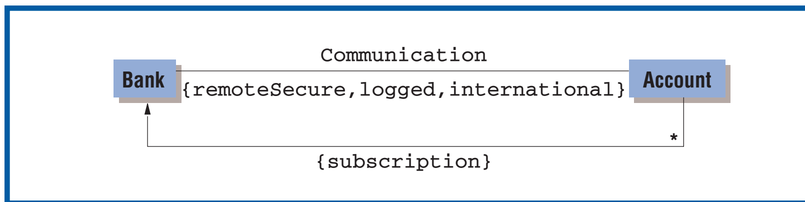


Figure 1. A simple architecture for a banking system. A secure communication protocol and a subscription scheme connect the two components, `bank` and `account`.

high potential for synergy from their integrated application. The synergies between components, frameworks, and model-driven development have in fact been recognized for some time in leading-edge framework technologies such as San Francisco[4] and the newest generation of development methods such as Catalysis[5] and KobrA.[6]

Despite its promise, AOP's separation of cross-cutting concerns has yet to be fully integrated with the other approaches. As a result, many developers view aspect-oriented development as an "academic" software-development technology less suited for enterprise software development than components frameworks and MDA. We describe a strategy that exploits the advantages of aspect-oriented development in model-driven and component-based frameworks. Key to our strategy is the use of architecture stratification,[7] rather than weaving technology, to address cross-cutting concerns. This not only addresses the limitations of the individual technologies but leverages the synergies between them.

Structural view of system architectures

In the software engineering literature, you'll often see phrases like "The architecture of a software system is …," implying that a single architecture uniquely captures a given system's important high-level properties. Although developers typically view an architecture from various perspectives[8] or using a range of decomposition techniques emphasizing different concerns (such as multiple tops[9]), these views are usually regarded as providing different windows onto the same basic underlying architecture. For example, structural views describe the underlying structural organization of the system, dynamic views describe system behavior, and physical views describe system deployment, including the mapping of software onto hardware.

Of the various possible views, developers generally regard the structural view as the most important, because it provides the foundation for organizing the other views in terms of the system's components and connectors. It also provides the basis for classifying common architecture styles.[10] In practice, however, you can define several structural architectures—depending on the level of abstraction at which you would like to see the system—each with different sets of connectors and components.

Consider Figure 1, a UML class diagram describing a simple banking system that uses remote and logged communication to access accounts holding international currencies. This representation provides a high-level view of the component and connector types making up the system's architecture. A secure communication protocol and a subscription scheme implying implicit invocation connect the `bank` and `account` component types.

Figure 2a describes the same system, again in terms of components and connectors, but at a lower level of abstraction. In the diagram, we've reified the notion of international currencies into new `Dollar` and `Euro` components. Even this more detailed scenario abstracts from other realization details—in this case the particulars of remote and secure communication. Figure 2b illustrates how we realize both the "logged" property of the communication between `bank` and `account` in terms of lower-level interactions with a `logger` object, and the communication's "remoteSecure" property in terms of lower-level interactions involving object request broker objects (`clientOrb` and `serviceOrb`). Figure 2c shows an even more detailed architecture in which the encrypted property of the communication between the object request brokers is realized in terms of interactions with a lower-level `encryption` object.

Figure 2 demonstrates that there are many valid ways to capture a system's structural architecture (that is, its component and connector types). The prevailing practice has been to pick one architecture (usually the least abstract) as the "real" or "best" architecture. Developers often see the others as temporary by-products to be discarded once they have determined the final version. With an appropriate organization scheme, however, you can use these architectures to systematically access views detailing important system concerns. Moreover, these concerns are typically the type of cross-cutting concerns AOP addresses.

We use architecture stratification to organize the various architectural representations so that they expose and relate the system's cross-

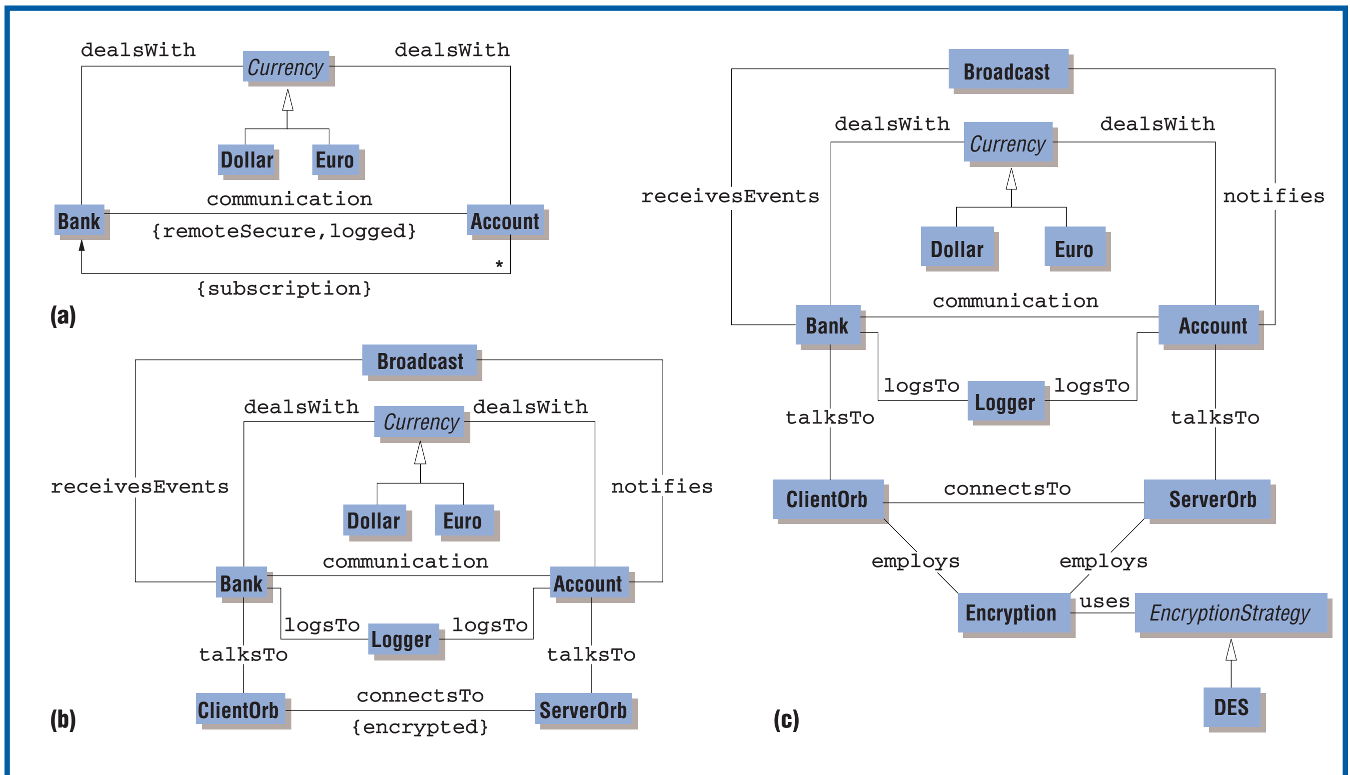


Figure 2. Different levels of abstraction in a banking system's structural architecture: (a) low detail, (b) medium detail, and (c) high detail.

cutting concerns. This lets us integrate an AOP-style separation of concerns with component, framework, and pattern technologies.

Architecture stratification

Architecture stratification's basic goal is to identify, elaborate, and relate different architectural views so that they best represent a system's cross-cutting concerns. Instead of ignoring the different architectural levels, we regard them as individual strata in a transcending architecture hierarchy. The development of these views is analogous to the well-established principle of stepwise refinement,[11] with the connectors between components as the entities driving the refinement.[12]

Figure 3 shows how this approach refines an interaction between two components, *X* and *Y*, and reifies it in a lower stratum—that is, a lower level of abstraction. Assuming *X* is `Bank` and *Y* is `Account`, the refinement explains how the two components communicate via the object request brokers *A* and *B*. In general, the semantics of an interaction between two components *X* and *Y* is given by a set of lower-level components and interactions with less rich semantics.

In the higher stratum in Figure 3, *Y* is *X*'s communication partner, whereas in the lower stratum *A* is. Thus *X* turns into *X*′, reflecting the communication partner change. A refinement transformation not only replaces complex interactions with components and simpler interactions but also adapts and modifies affected components accordingly. The refinement doesn't affect *Y* because the asymmetry of the client-server component interaction model means that a client needs to know its server's identity, but a server doesn't need to know its clients' identities.

Note that the fundamental characteristics of an architectural stratum or its relationships to neighboring strata are independent of the notation used to capture the components and connectors involved. Thus, we can use the stratification concept with any description language, including a high-level modeling language such as UML, a domain-specific architecture-description language, or a low-level, general-purpose programming language. When used with a high-level description language, architecture stratification provides an excellent basis for the MDA approach because it directly addresses the central question of how to organize and relate different kinds of architectural models.

Refinement transformations

Refinement transformations capture the relationship between a connector in one stra-

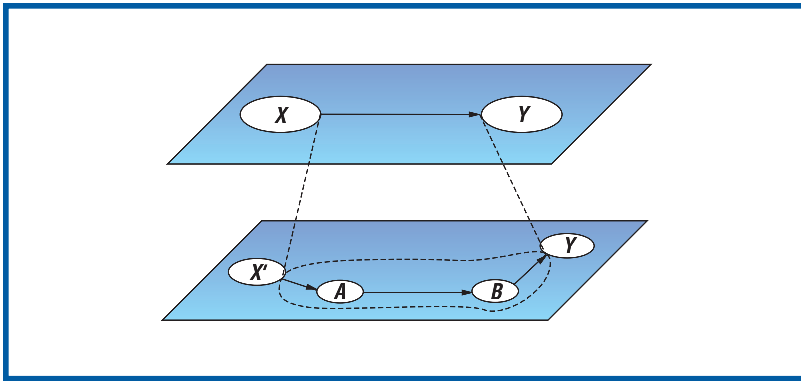


Figure 3. Interaction refinement. The lower stratum shows the interaction between *X* and *Y* in the top stratum, but at a lower level of abstraction. The communication between *X* and *Y* occurs through object request brokers *A* and *B*.

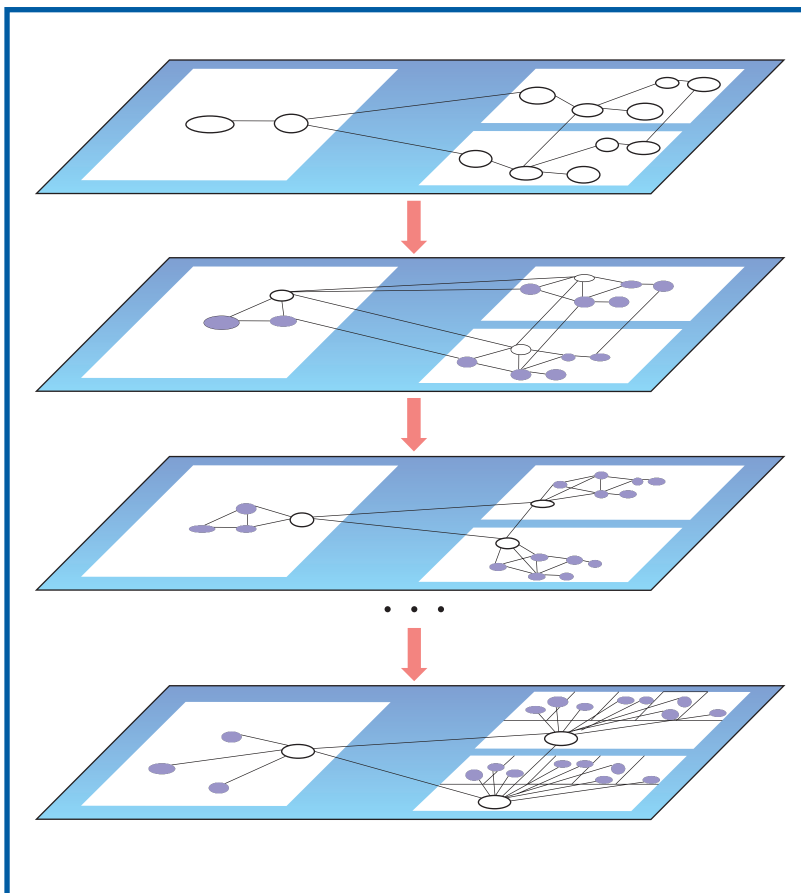


Figure 4. Stratified architecture. Working downward, each subsequent stratum is a refinement of the one above. White ovals represent the objects or components that result from refining an interaction in the adjacent higher-level stratum.

tum and connectors and components in the stratum below. Such transformations define a relationship between the originating and the resulting scenarios, detailing the components that must be changed or added. We deliberately use the "relation" concept because relations are inherently undirected, and thus independent of system development direction. We could just as easily elaborate an architectural strata's hierarchy using a bottom-up approach as a top-down approach.

Our use of annotations to distinguish the types of interactions in Figure 2 lets us tie a set of refinement transformations to an annotation language in which labels can refer to concerns (`subscription`, for example). Although the set of refinement transformations and associated interaction labels (the concern designators such as `remoteSecure`, `logged`, and `encrypted`) depend on the domain under consideration, we can generally reuse them in architectures in similar domains. Moreover, the lower the level a concern (encrypted, for example), the more likely it is to recur in multiple domains. Relating interaction labels to refinement transformations in this way provides a basis for a domain-specific description language capable of expressing typical domain concerns.

Other methods for organizing architectural representations also use explicit-interaction refinement to help relate concepts at different levels of abstraction and to drive development. However, these methods typically refine interactions independently of one another.[5] To obtain a meaningful strata hierarchy like the one Figure 4 illustrates, you must systematically select a specific set of interactions to refine in each stratum. For instance, if you start from a stratum capturing mostly business-level interactions (Figure 2a) and select all interactions pertaining to remote communications, you'll get a communication stratum. If you then select all interactions relying on some form of persistence capability, you'll get a persistence stratum, and so on. In Figure 4, white ovals represent the objects or components introduced at one level by refining selected interactions from the level above.

Strata versus layers

Layering is a common technique for making the parts of a large, complex system more independent and reusable. At first sight, strata like those we've introduced might appear to be merely a recasting of layers as used in traditional architectural approaches. This is not the case, however. Strata, although motivated by the same basic goal of attaining a hierarchical structure, differ fundamentally from layers.

A layer represents a slice through a system

Table 1

Classification of concerns

Concern	Subconcern	Subsubconcern
Infrastructure	Coordination	Scheduling Synchronization
	Distribution	Fault tolerance Communication (for example, serialization) Replication
	Persistence	
Services	Security	Encryption Authorization
	Recovery (for example, exceptions)	
	Undo	
Paradigms	Functions (such as visitor pattern)	
	Events (publish-subscribe pattern, for example)	

that packages a coherent subset of the system's functionality, such as platform, infrastructure, domain, and application layers. Thus, a single layer can't convey the whole system's functionality. Rather, you must consider all layers simultaneously to understand what the whole system does. Each architecture stratum, however, describes the entire system, but with a different degree of abstractness. Each stratum therefore describes what the whole system does, but not how it does it. Different strata elaborate on how the system realizes a certain part of the overall functionality. Thus, strata are more like representations of a complete program at different levels of abstraction (for example, source, assembler, and binary code).

Transparency and abstraction goals can also highlight the difference between layers and strata. When moving upward in a layered architecture, the layers become not only more application-specific, but also—like strata—more concise in the sense that top layers use concepts with richer semantics than lower levels. In systems with strict layering,[12] however, lower-layer functionality is completely transparent to clients several layers above. In other words, code in higher layers can't refer to details present several layers deeper. This is an advantage because it lets you replace complete layers without affecting layers above, but it is also a disadvantage, because such transparency can be perceived as restrictive. Often, for reasons of efficiency or flexibility, a layer must bypass adjacent lower layers to directly access low-level services. Architectures allowing such bypasses are generally referred to as having nonstrict layering.[12]

Rather than completely remove detail, it is usually preferable to merely abstract from detail. Abstraction doesn't make details inaccessible; rather, it makes certain details irrelevant at a particular level of abstraction. This is what architecture stratification accomplishes. Because each stratum describes the full system, a client can explicitly access any low-level functionality, but only in the corresponding low-level stratum.

The ability to abstract from details when possible, but provide access where necessary, is particularly important for cross-cutting concerns. A major problem of weaving-based systems, especially when dealing with multiple interfering concerns, is that they aim for transparency between the target code and all aspects respectively. (See the "Architecture Stratification vs. Other Aspect-Oriented Approaches" sidebar for a discussion of weaving-based approaches.) The aspect-oriented community has so far been unable to fully solve the resulting superimposition problems.

Because the concepts of layers and architectural strata are orthogonal, they are fully compatible and can be combined to organize information in a model-driven architecture. We can organize the components within a given stratum in terms of layers.

Strata, concerns, and patterns

Although stratification is a simple and natural idea, to apply it effectively you must refine appropriate sets of interactions in one stratum to realize a well-defined concern in the stratum below. If done correctly, the resulting abstraction levels disentangle different cross-cutting concerns and present them at their natural detail level. Different concerns will only make themselves visible when stratification exposes certain detail levels.

Suppose, for example, we use the classification of concerns in Table 1 as the basis for stratifying an architecture. As the strata re-

Architecture Stratification vs. Other Aspect-Oriented Programming Approaches

Weaver-based approaches operating on code artifacts completely separate aspect code from base code. This strength, however, is also a weakness: when several aspects and the base code interfere at the same join points, issues of priority, nesting, and mutual exclusion arise. All separation-of-concerns approaches that use a low-level code structure have difficulty defining join points in terms of higher-level system abstractions because they must recreate these abstractions from the flat code structure. Furthermore, weaver-based technologies have only just begun to define aspects based on other aspects rather than the base code, whereas the hierarchical organization of concerns is a built-in feature of stratification.

Our architecture stratification approach currently lacks dedicated tool support, but it can benefit the development process even if applied manually. Although manually applying stratification requires developers to explicitly work with tangled specifications, this is a natural consequence of abstraction. Some aspect interferences must be dealt with manually, and the lower strata are the best place to do this effectively.

Another approach for making a framework aspect-aware is to provide aspect functionality (such as an `AspectModerator` class) within the framework.[1] This requires neither an aspect nor an annotation language. Furthermore, by deferring weaving to runtime, even the dynamic addition and reconfiguration of aspects becomes possible. Unlike stratified frameworks, however, this approach does not provide any leverage in terms of abstraction levels and can suffer from more efficiency problems than compile-time weaving.

Reference

1. C. Constantinides et al., "Designing an Aspect-Oriented Framework in an Object-Oriented Environment," *Proc. ACM Computing Surveys Symp. Application Frameworks*, ACM Press, New York, 2000; http://portal.acm.org/citation.cfm?doid=351936.351978.

veal more detail—proceeding from top to bottom—more concerns become relevant. The strata hierarchy will thus mirror the classification of concerns shown in Table 1. As one concern (distribution, for example) is elaborated in one stratum, it raises the need for a dependent concern (such as security) at a lower level. As a result, concerns are separated from each other until a certain level of detail (that is, the stratum in which they are first introduced), from which they spread into aspects and other dependent concerns as the architecture unfolds into a certain design (typically a decomposition into pure components or objects).

A stratified architecture therefore provides two invaluable pieces of information about a concern:

- The abstraction level at which a concern is first introduced (top-down view)
- The degree of influence a concern has on the overall system structure (bottom-up view)

The latter explains why choosing a certain implementation technology, such as a particular middleware solution, can affect higher-level system structures. In fact, these bottom-up concern implications often make it hard to match a single interface to a particular middleware solution, and then simply swap it for a competing implementation technology.

Note also that a stratified architecture reifies interactions into components (see Figure 3). In other words, stratification refines a concern associated with an interaction into components. High-level interaction attributes, such as secure data transmission, will eventually be described in the form of components (encryption interfaces and so on).

Abstractions as aspect anchors

Because concerns are irrelevant until the developer requires a certain level of detail, we can define a concern's corresponding aspect in terms of the system abstractions inhabiting the stratum in which the concern first appears.

One of the main challenges for weaver-based aspect-oriented approaches operating at the code level[3] is expressing where in the code to apply an aspect. Developers currently use declarative patterns to match locations in the base code to determine the join points between base code and aspect. The problem with this approach is that the patterns can only use abstractions from the implementation domain—typically methods, classes, and packages. Any higher-level system abstractions must be recreated, for example by the systematic use of naming conventions.

In a stratified architecture, you can anchor an aspect to system abstractions at the relevant level. In general, the abstraction level at which a concern becomes relevant usually offers semantically richer abstrac-

tions than those in the implementation domain. You can think of these components and interactions at higher strata as reified abstractions of the lower strata. Hence, it is easier to attach the aspect to these abstractions and let the respective refinement transformations distribute it to the scattered locations in the low-level realization.

Organizing patterns

Stratification's ability to untangle information is also valuable when software architecture descriptions include patterns.[13] Although patterns typically address different levels of abstraction (for example, architectural styles, microframeworks, and language idioms), they are usually applied in a single, flat level of abstraction. Without further documentation, it is difficult to reconstruct why the patterns were applied and how they are related.

A stratified architecture alleviates this problem by letting a user express the application of particular patterns at their natural levels of abstraction. Thus, in a typical stratified architecture, patterns will be distributed among the various strata according to the concerns they address and the problems they solve, making it much easier to comprehend where patterns are being used and why. For example, Figure 5 illustrates how using the blackboard architecture pattern in the top stratum leads to the use of the publisher–subscriber pattern in the stratum below, which in turn leads to the use of the double-dispatch idiom in the next stratum to determine the correct action depending on both agent and event type.

The fact that patterns and concerns are distributed similarly across strata should not be surprising because, as mentioned previously, design patterns usually address specific concerns such as coordination (reactor pattern), events (publisher–subscriber pattern), or functions (visitor pattern). They therefore naturally appear in the stratum that focuses on the concern they are intended to address.

You can use this clarified presentation of pattern application levels and relationships not only in system construction but also to discover pattern languages—that is, reoccurring pattern configurations that define each other's context in a dependency tree structure.

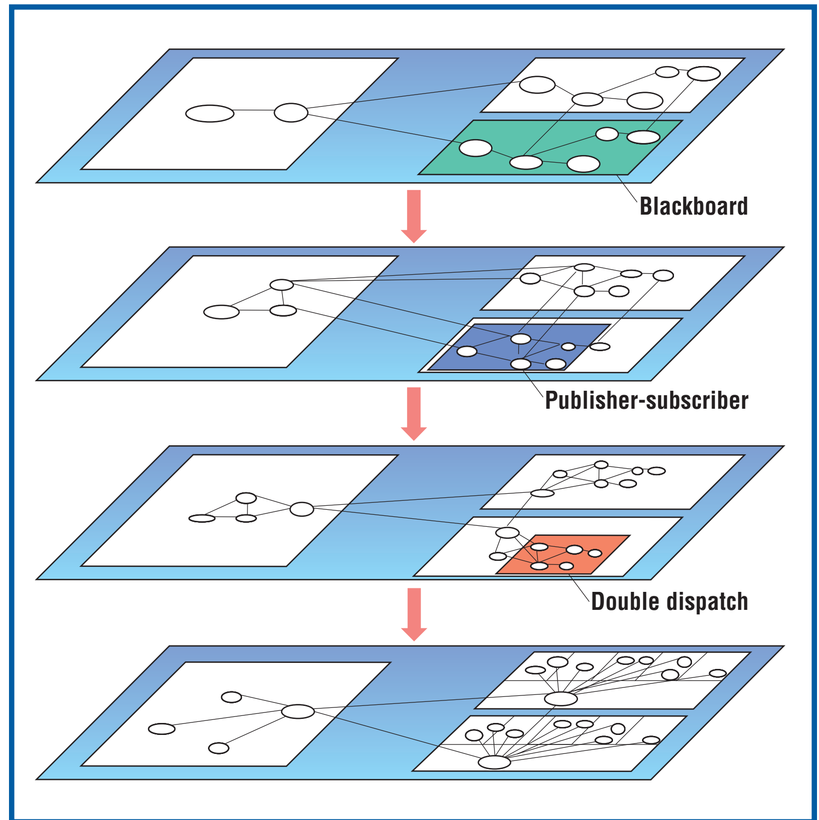


Figure 5. Stratified patterns. Stratified architectures typically distribute patterns according to the concerns they address and the problems they solve.

Stratified frameworks

Although stratification has many advantages over other strategies for modularizing and describing cross-cutting concerns, if you must manually create and maintain strata and use traditional implementation technologies to realize the information captured in a stratified architecture (essentially by implementing the lowest-level stratum), the approach essentially becomes a sophisticated but work-intensive design approach. In contrast, AOP allows separately defined aspects to be weaved automatically into an executable system.

Applying the architecture stratification approach in a framework context alleviates this problem. Essentially, frameworks are collections of software development artifacts (including executable code) that have been parameterized so that users can instantiate them into user-specific applications by defining minimal additional (user-specific) information. Thus, if you apply architecture stratification within the context of a parameterized, semicomplete framework, you can exploit the effort put into the separate

Stratification's ability to untangle information is also valuable when software architecture descriptions include patterns.

description of concerns through stratification every time you instantiate the framework. In other words, whereas weaver technologies automate the exploitation of aspects by "weaving" them together, frameworks automate their exploitation by facilitating their straightforward instantiation. Frameworks are usually long-lived and have variants, so each time you instantiate the framework, the initial effort involved in stratifying it is amortized.

Organizing hot spots

Using architecture stratification principles to organize artifacts in a framework also addresses some significant problems with traditional framework technologies. First, the disentanglement of information provided by stratification is invaluable for users needing to understand the framework at both the code and design levels. Because users often have difficulty comprehending a framework's extension and parameterization points, they rarely use them effectively. Stratification distributes these *hot spots* (points of variation) among the strata according to their natural level of abstraction, just as it distributes concerns and patterns. The strata therefore naturally clarify and expose the framework's extension and parameterization points.

Parameterized interactions

Stratification also lets developers parameterize or extend more of a framework's properties in an object-oriented way. Frameworks have traditionally supported the adaptation and exchange of data types and some pluggable or redefinable behavior, but the interactions within the framework are typically predefined and fixed. In a stratified framework, you can modify or adapt an interaction simply by changing the components responsible for realizing it in the stratum below. While this is also possible in a flat framework without abstraction levels, such frameworks lack the structure to facilitate simple identification of the interaction requiring the change.

Traceability

Finally, stratification lets developers trace corrective or evolutionary changes made to the framework through the refinement transformations, facilitating evaluation of the changes' impact. For example, you can trace a change to a rather high-level system abstraction downward along the refinement transformations to validate its effect on the more detailed strata. Likewise, you can trace a low-level change upward to check which higher-level interactions will be affected. Starting from this higher strata, you can again use the refinement transformations to navigate back down to the detailed strata, evaluating the effect of the modifications. Through this process, you can systematically evaluate the system-wide impact of a local change.

As the software industry struggles to come to terms with the OMG's new MDA paradigm, stratifications' support of the MDA approach might prove to be its most important advantage. A key challenge in effective model-driven development is the concise and modular description of key architectural concerns and their traceable mapping to concrete executable representations. Architecture stratification provides an ideal conceptual foundation for creating and organizing such models.

Our current research focuses on integrating the architecture stratification approach with our work on model-driven development, particularly object-oriented metamodeling and KobrA.[6]

About the Authors

Colin Atkinson is a professor at the University of Mannheim. His research focuses on object and component technology and its use in the systematic development of software systems. He received a BSc in mathematical physics from the University of Nottingham and an MSc and a PhD in computer science from Imperial College, London. He is a member of the IEEE and the British Computer Society. Contact him at colin.atkinson@ieee.org.

Thomas Kühne is a junior professor at the Darmstadt University of Technology. His research interests are object technology, programming languages, software architectures, and metamodeling. He received an MSc and a PhD from the Darmstadt University of Technology, Germany. Contact him at FG Metamodeling, TU Darmstadt, Wilhelminenstr. 7, 64283 Darmstadt, Germany; kuehne@informatik.tu-darmstadt.de.

Acknowledgments

We carried out initial conceptual work on stratification at the University of Kaiserslautern within the German Special Research Project SFB 501.

References

1. D. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Comm. ACM*, vol. 15, no. 12, 1972, pp. 1053–1058.
2. G. Kiczales et al., "Aspect-Oriented Programming," *Proc. European Conf. Object-Oriented Programming*, Lecture Notes in Computer Science, no. 1241, Springer-Verlag, Berlin, June 1997, pp. 220–242.
3. Object Management Group (OMG), "Executive Overview: Model-Driven Architecture," www.omg.org/mda/executive_overview.htm, 2001.
4. P. Monday, J. Carey, and M. Dangler, *San Francisco Component Framework,* Addison-Wesley, Boston, 1999.
5. D.F. D'Souza and A.C. Wills, *Objects, Components and Frameworks with UML: The Catalysis Approach,* Addison-Wesley, Boston, 1998.
6. C. Atkinson et al., *Component-Based Product Line Eng. with UML,* Addison-Wesley, Boston, 2001.
7. C. Atkinson and T. Kühne, "Stratified Frameworks," *Int'l J. Computing Science and Informatics, Informatica,* vol. 25, no. 3, 2001, pp. 393–401.
8. P. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, vol. 12, no. 6, Nov./Dec. 1995, pp. 42–50.
9. J.O. Coplien, *Multiparadigm Design for C++*, Addison-Wesley, Boston, 1998.
10. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, Upper Saddle River, N.J., 1996.
11. N. Wirth, "Program Development by Stepwise Refinement," *Comm. ACM,* vol. 14, no. 4, Apr. 1971, pp. 221–227.
12. C. Szyperski, *Component Software: Beyond Object-Oriented Programming,* Addison-Wesley, Boston, 1999.
13. M. Broy, *Towards a Mathematical Concept of a Component and Its Use*, tech. report I9746, Technische Universität München, Munich, Germany, 1997.
14. E. Gamma et al., *Design Patterns: Elements of Object-Oriented Software Architecture*, Addison-Wesley, Boston, 1994.

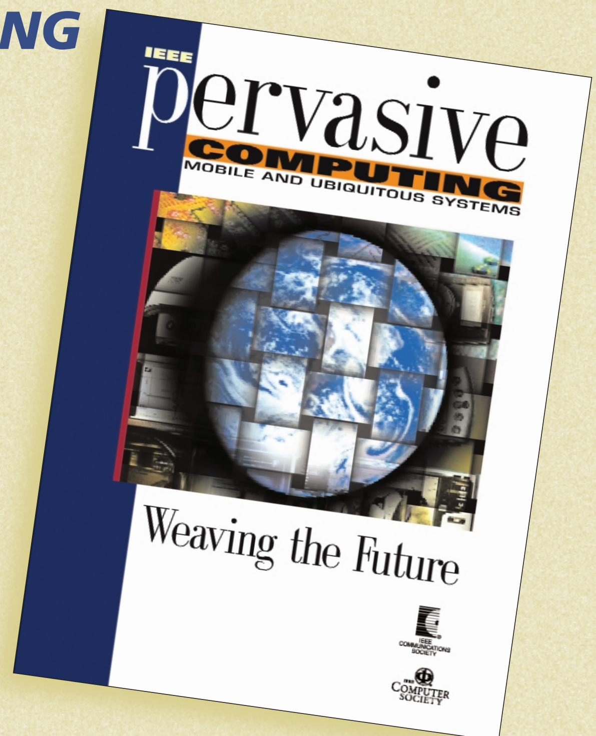
For more on this or any other computing topic, see our Digital Library at http://computer.org/publications/dlib.

IEEE PERVASIVE COMPUTING

The exploding popularity of mobile Internet access, third-generation wireless communication, and wearable and handheld devices have made pervasive computing a reality. New mobile computing architectures, algorithms, environments, support services, hardware, and applications are coming online faster than ever. To help you keep pace, the IEEE Computer Society and IEEE Communications Society are proud to announce ***IEEE Pervasive Computing***.

This new quarterly magazine aims to advance mobile and ubiquitous computing by bringing together its various disciplines, including peer-reviewed articles on

- **Hardware technologies**
- **Software infrastructure**
- **Real-world sensing and interaction**
- **Human–computer interaction**
- **Security, scalability, and privacy**



Editor in Chief
M. Satyanarayanan
Carnegie Mellon Univ. and Intel Research Pittsburgh

Associate EICs
Roy Want, Intel Research; Tim Kindberg, HP Labs; Deborah Estrin, UCLA; Gregory Abowd, GeorgiaTech.; Nigel Davies, Lancaster University and Arizona University

SUBSCRIBE NOW!

http://computer.org/pervasive