

Discovering Early Aspects

Elisa Baniassad, *Chinese University of Hong Kong*

Paul C. Clements, *Carnegie Mellon University*

João Araújo and Ana Moreira, *New University of Lisbon*

Awais Rashid, *Lancaster University*

Bedir Tekinerdoğan, *University of Twente*

An integrated approach to working with early aspects lets you identify them and exploit them throughout the software development life cycle.

Traditionally, aspect-oriented software development has focused on the software life cycle's implementation phase: developers identify and capture aspects mainly in code. But aspects are evident earlier in the life cycle, such as during requirements engineering and architecture design.

Early aspects are concerns that crosscut an artifact's *dominant decomposition*, or base modules derived from the dominant separation-of-concerns

criterion, in the early stages of the software life cycle. "Early" signifies occurring before implementation in any development iteration. An aspect in requirements is a concern that crosscuts requirements artifacts; an aspect in architecture is a concern that crosscuts architectural artifacts. Identifying and managing early aspects helps to improve modularity in the requirements and architecture design and to detect conflicting concerns early, when trade-offs can be resolved more economically.

In addition, identifying aspects at one stage provides benefits downstream. Knowing requirements-level aspects helps the architect design a better system, and knowing architecture-level aspects helps produce a more robust implementation. Early aspects can span development activities, and many find their way into the code as traditional implementation aspects. More concretely, identifying and managing early aspects across phases can

- increase the consistency of requirements and architecture designs with each other and with the implementation;
- provide a rationale and traceability for aspects across life-cycle activities; and
- help ensure that crosscutting concerns evident in a system's problem domain or solution space are captured as aspects in the implementation.

In this article, we describe how to identify and capture early aspects in requirements and architecture activities and how they're carried over from one phase to another. We'll focus on requirements and architecture design activities to illustrate the points, but the same ideas apply in other phases as well, such as domain analysis or in the fine-grained design activities that lie between architecture and implementation. There are many approaches for working with early aspects in practice (for a

Dominant Decomposition in Early Software Development

Aspect-oriented software development (AOSD) emerged from a rethinking of the relationship between modularization (partitioning software into discrete, nonoverlapping implementation units) and the time-honored principle of separation of concerns. Any separation-of-concerns criterion leads to a particular partitioning, as though the software were sliced into pieces in a particular direction. But this inevitably leads to concerns that wash across the resulting modules. This is known as the *tyranny of the dominant decomposition*.¹ AOSD introduces the concept of an *aspect*—a concern that cuts across the base modules derived from the dominant separation-of-concerns criterion—plus automated support to weave the separately described aspects back into the base modules.

Work in aspects has been mostly limited to the implementation phase, dealing with concerns that implementation units have in common, factoring those out as aspects, and employing programming-language-level support to weave the aspects back at loading time, compilation time, or runtime. In fact, AOSD has become nearly synonymous with aspect-oriented programming (AOP),² and dominant decomposition usually refers to the system's decomposition into implementation units, such as subsystems, classes, and objects. Most AOSD approaches place the burden for aspect identification and man-

agement on the programmer.

But crosscutting concerns are often present well before the implementation, such as in domain models, requirements, and the architecture. Dominant decomposition, however, means something different in the early software development activities. For example, modern treatments of software architecture all embrace the concept of multiple architectural views. The dominant decomposition for the architecture is the set of views chosen to represent it. Similarly, requirements are typically organized into sections that describe features, domain objects, viewpoints, system components, subsystems, use cases, or stakeholder goals. The dominant decomposition for requirements is the organization of the requirements document into sections on the basis of stakeholders' perspectives—for example, use case, viewpoint, and goal descriptions.

References

1. P.L. Tarr et al., "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *Proc. 21st Int'l Conf. Software Eng. (ICSE 99)*, ACM Press, 1999, pp. 107–119.
2. G. Kiczales et al., "Aspect-Oriented Programming," *Proc. 11th European Conf. Object-Oriented Programming (ECCOP 97)*, LNCS 1241, Springer, 1997, pp. 220–242.

list, visit www.early-aspects.net), but they've existed mostly in isolation. We borrow practices from some of these approaches to offer an integrated view of working with early aspects, including how to link the life-cycle phases together.

Aspects and requirements

The dominant decomposition for requirements is the requirements document's organization into sections that its author has chosen (see the sidebar for more on dominant decomposition). Assume we're starting with a relatively well-organized requirements set. Where possible, we capture each concern separately, perhaps in its own section, document, or any other requirements artifact. These artifacts can capture any concern: a structural entity, a use case, a feature, a behavior, or a stakeholder goal. A *requirements aspect*, then, is a concern that cuts across other requirement-level concerns or artifacts of the author's chosen organization. It is *broadly scoped* in that it's found in and has an (implicit or explicit) impact on more than one requirement artifact. Broadly scoped properties can be quality attributes (nonfunctional requirements) as well as functional concerns

that the requirements engineer must describe with relation to other concerns.

The scenario

Consider a banking system with many requirements traditionally organized, including the following (each requirement may expand to many requirements or represent a section title in a requirements document):

Requirements A

1. *Pay interest* of a certain percent on each account making sure that the *transaction is fully completed* and an *audit history* is kept.
2. Allow customers to *withdraw* from their accounts, making sure that the *transaction is fully completed* and an *audit history* is kept.

These requirements reveal "pay interest," "withdrawal," "complete in full," and "auditing" as central concerns. Of those concerns, "pay interest" and "withdrawal" are described in separate requirements. However, "complete in full" and "auditing" are each described in both requirements 1 and 2. Figure 1a depicts this concern scattering. This arrangement is problematic: If we want to find out which trans-

actions should be fully completed or audited, we must sift through the whole requirements set for references to transactions and auditing.

Attempting to rewrite the requirements to remove the scattering would result in the following requirements:

Requirements B

- 1Δ. *Pay interest* of a certain percent on each account.
- 2Δ. Allow customers to *withdraw* from their accounts.
3. Make sure all *transactions are fully completed*.
4. *Keep an audit history* of all transactions.

This rewriting introduces implicit tangling between the newly separated concerns (“auditing” and “complete in full”) and the other concerns (“pay interest” and “withdrawal”), as figure 1b illustrates. However, it would be impossible to discuss how transactions should be audited or completed in full without at least hinting at what the transactions should be. So, this arrangement is also problematic: You can’t tell, without an exhaustive search, which transactions the “complete in full” and “auditing” properties affect.

However, an intuitive and inherently aspect-oriented solution exists (see figure 1c). In fact, the “complete in full” and “auditing” properties are aspects. They’re broadly scoped in that they impact other concerns described in the requirements (“withdrawal” and “pay interest”). The AO solution is to make the impact explicit by modularizing aspects into two sections: one that describes the requirements of the aspect concern itself (3Δ, 4Δ), and another that describes the breadth of its impact (3A, 4A). The AO solution results in the following:

Requirements C

- 1Δ. *Pay interest* of a certain percent on each account.
- 2Δ. Allow customers to *withdraw* from their accounts.
- 3Δ. To *fully complete a transaction*...
- 3A. List of transactions that must be fully completed: {1Δ, 2Δ}
- 4Δ. To *audit*...
- 4A. List of transactions that must leave an audit history: {1Δ, 2Δ}

With this organization, we capture

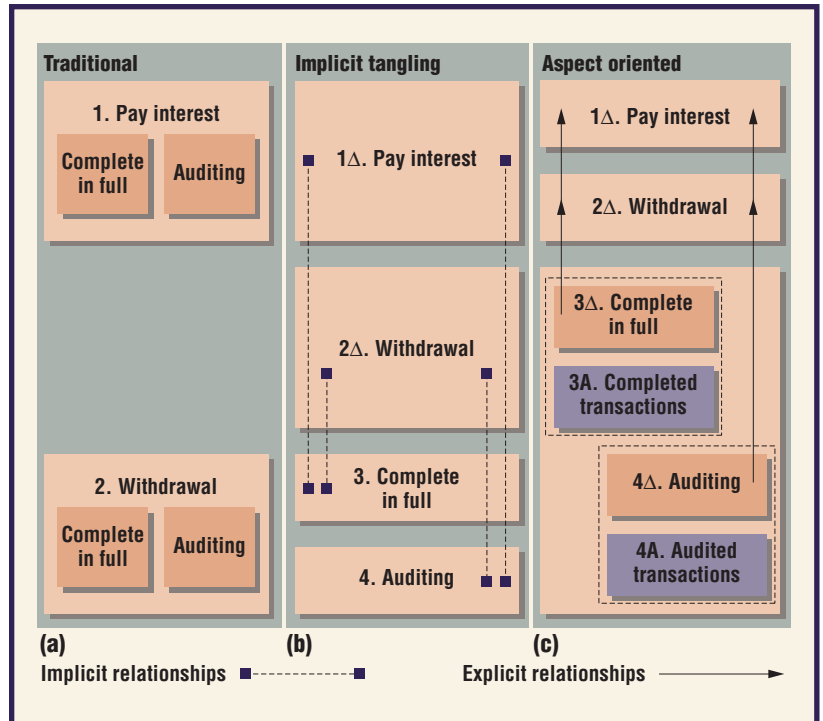


Figure 1. Three approaches (Requirements A-C) to organizing requirements: (a) Manifestation of broad impact (the same concern in multiple requirements), (b) implicit relationships (one concern implicitly mentions another), and (c) explicit relationships (impact requirements (in purple) describe one concern’s impact on another).

- Core or base concerns (“withdrawal” and “pay interest”): 1Δ, 2Δ
- Aspect descriptions: 3Δ, 4Δ
- *Impact requirements*: a requirement describing the influence of one concern over other concerns: 3A, 4A (purple in figure 1c)

Activities

Now that we have an idea of what a requirements-level aspect looks like, we can consider a less ad hoc approach for working with them. This involves four basic steps: identify, capture, compose, and analyze.

Identify. In an existing set of requirements, it isn’t likely to be clear what aspects are present. In the banking example, we used two typical aspect examples: transaction management and auditing. In a way, we used domain knowledge about banking to recall that transaction management is typically a crosscutting concern in requirements. (A domain model endowed with early aspects of its own would have brought this to our attention explicitly.) But a requirements

The multiview architectural approach moves away from the idea that only one structure determines a system's decomposition.

document might describe many atypical aspects. This step involves spotting these, so that we can manage them effectively and consistently.

When identifying aspects in requirements, look for:

- *Aspect terms.* These are quality attributes (such as security, data consistency, or reliability). “Complete in full” is such a property. Requirements engineers can identify these terms using simple searching techniques or tools made especially for aspect-oriented requirements analysis, such as AORE¹ and EA-Miner.²
- *Impact requirements.* These describe one concern’s influence over another. In our example, requirements 3A and 4A indicate how requirements engineers must impose “complete in full” and “auditing” on “pay interest” (1Δ) and “withdrawal” (2Δ). Visualization techniques such as Theme/Doc,³ EA-Miner,² and XML-based techniques such as ARCADE⁴ can point out such requirements because they show which requirements describe which concerns, revealing concern overlap.
- *Scattered concerns.* These are terms, concepts, or behaviors that appear in multiple (well organized) requirements. Such concerns may have broad impact. If a series of use cases describe the requirements in the banking system, the concept of auditing might appear in both the “pay interest” and “withdrawal” use cases. Since auditing is scattered over multiple artifacts, it’s likely an aspect. It’s possible that not all scattered concerns should be captured as aspects, however. Some scattered concerns might be too trivial or heterogeneous to capture separately. For instance, if a different type of auditing is required for every transaction, it’s unhelpful to decouple auditing’s description from its transaction description. However, if some core concepts related to auditing crosscut the other concerns, those should be collected and their points of impact recorded.

Capture. In this step, requirements engineers reorganize the requirements so that each requirement artifact describes only one concern. In the previous example, we did this by transforming the requirements from the model shown in figure 1a to that shown in figure 1b,

removing the “auditing” and “complete in full” properties from the “withdrawal” and “pay interest” descriptions and creating new artifacts describing those two concerns.

Compose. In this step, we formally state the impact requirements (3A and 4A in figure 1c) to specify how concerns should be composed. Compositions in requirements are analogous to *pointcut specifications*—a selection of a set of *join points*, or well-defined points in the base modules where the aspect should be applied—in AOP implementation. But not surprisingly, composition in requirements looks different from composition in code. For instance, to capture the influence of “complete in full” (3Δ) and “auditing” (4Δ) on “withdrawal” (2Δ) and “pay interest” (1Δ), we would write these specifications:

- Constrain {1Δ, 2Δ} by 3Δ or Constrain *withdrawal and pay interest* by “complete in full”
- Constrain {1Δ, 2Δ} by 4Δ or Constrain *withdrawal and pay interest* by “audit”

Drawing out the analogy with join-point models for AOP, the italicized elements in this composition specification are the pointcuts, the underlined elements are the type of advice, and the elements in quotes are the advice behavior.

To ensure that we capture all the relationships, we must fully assess each aspect’s breadth of impact. Once again, visualization techniques such as Theme/Doc and EA-Miner may help assess which concerns a broadly scoped property affects.

Requirements engineers can compose requirements using techniques such as those Rashid and his colleagues discuss.⁴ Some composition techniques result in scenarios or state machines,⁵ or projections of the aspect influences on other requirements.⁶

Analyze. This step involves analysis of the modularized aspect concerns to understand their trade-offs with respect to other requirements and to identify conflicts and inconsistencies. The analysis is facilitated by composing the requirements using the techniques we just described, which has the benefit of revealing potentially conflicting aspects.^{4,6} For instance, if we add a new requirement regarding response time for a completed transaction, a

Table 1**Software architecture views**

	Module view	Component-and-connector view	Allocation view
Elements	Units of implementation	Units of run-time behavior	Software and structures in the software's environment
Relationships	How the system is constructed (such as "specializes," "is a part of," "inherits from," or "is allowed to use")	How the elements interact with each other to do the system's work (such as "sends data to," "invokes," or "synchronizes with")	How the software elements map to environmental elements (such as "deployed on" or "assigned to")

conflict might arise if auditing is carried out within a transaction, due to the time it would take. We can then consider and resolve such conflicts.

Aspects and architecture design

A software architecture is a software system's structure or structures; these structures comprise elements, those elements' externally visible properties, and the relationships among them.⁷

Views

Modern treatments of architecture represent structures and other architectural information as a set of *views*, each of which shows a particular kind of element and the corresponding relations among those elements. The three basic kinds of views are module, component-and-connector, and allocation (see table 1).⁸

Some views are hybrid, conveying information belonging to more than one of these families. For instance, a UML class diagram used to illustrate an architectural view can show both structural and execution-time information. Views are often thought of as a documentation mechanism, but equally important is that they're also an architecture creation method (a by-product of which is architectural aspect discovery). An architect engineers the structures native to each view to achieve the system's quality and behavioral goals. A view-based architecting process can and will reveal intra-view or cross-view commonalities that the architect can capture as aspects.

Hence, we consider that the views themselves, rather than the software elements and relationships they show, form the dominant decomposition for an architecture. The whole point of the multiview architectural approach is to move away from the idea that systems have only one structure or style determining the decomposition. The decomposition is hier-

archical, as views have substructure. The architect can decompose a view into "view packets" that show small parts of the system at great depth, or broader parts of the system at shallow depth. Aspects in architecture, then, manifest as discovered concerns that crosscut views or parts of views. We can use aspects to naturally and conveniently capture issues that often arise during architectural design, such as

- making sure all elements share common behaviors, such as a start-up protocol, recovery in case of failure, reporting in case of error, and so on (for example, all elements in our banking system might need to behave the same if asked to execute a transaction on a nonexistent account);
- specifying a common substructure—for example, requiring every application module to include three submodules to monitor the module's health, periodically send a heartbeat to the system-wide health monitor, and check the module's inputs for validity, respectively; and
- giving elements interfaces that have parts in common, such as common sets of methods by which the elements' functionality can be invoked.

Activities

As in requirements, the main activities concerning architectural aspects are identify, capture, compose, and analyze.

Identify. Although no automated technique exists for identifying architectural aspects, the architect can use utilitarian, manual approaches for finding crosscutting architectural concerns during the normal activities of architecture-based development.

Analyze the system's business case. The architect uses the business case to identify stake-

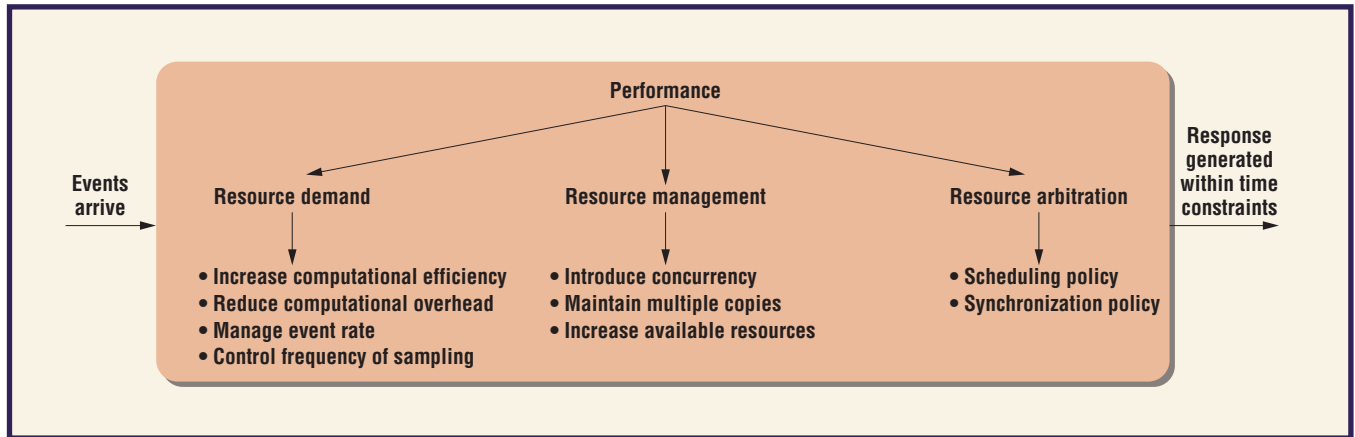


Figure 2. Tactics for increasing system performance.

holders and the driving goals behind the system’s construction. Beyond what a requirements specification usually captures, these goals might represent the developing organization’s ambitions underlying the system. An example is an organization’s desire to reuse a subsystem being developed for the current project in future systems. Another example is a desire to use Java 2 Enterprise Edition because the staff is already trained in it. These goals might eventually be refined into quality attribute goals (such as for performance, security, or reusability) that will shape the architecture, or they might simply translate to design constraints. Analyzing the business case can identify overriding concerns that might cut across the architectural elements of the design and hence be candidates for aspects.

Understand the architecturally significant requirements. Some requirements are more influential than others in the architecture. To identify these requirements, architects look for quality attributes that reflect the system’s highest-priority business goals and those that have the most impact on the architecture, especially decomposition decisions. In this step, the architect can use requirements aspects identified earlier. In the banking example, we’re required to keep an audit history of all transactions; this was captured as a requirements aspect. This aspect provides a cue that every architectural element whose function is to carry out a transaction must record its actions in an audit history.

Select patterns and tactics. Most prescriptive approaches to architecture design start with choosing appropriate architectural patterns. Architectural patterns represent concerns satisfied using a group of architectural elements.

Work on code-level design patterns has shown that capturing patterns using aspects is promising. The same is true for architectural patterns (for example, those that Frank Buschmann and his colleagues discuss⁹), where they form the basis for prepackaged architectural design decisions. Architectural tactics represent a finer-grained approach to imbuing the architecture with quality attributes.⁷ For example, redundancy is a tactic for increasing a system’s availability. Figure 2 shows tactics for increasing system performance.⁷ Choosing patterns and tactics to solve a design problem helps identify architectural aspects in two ways:

- A pattern or tactic used in several places in the architecture is by definition “crosscutting” (an aspect) and need be specified only once.
- Some architectural patterns might call directly for the use of aspects in their definition. These aspects are simply added to the architectural aspects list.

Finally, we can consider texture. Architectural *texture* refers to a set of fine-grained design decisions that apply to a broad set of elements.¹⁰ For example, our banking system elements that perform transactions on a shared database will be required to follow certain protocols to allow rollback in the case of failure. Texture decisions such as these are candidates for aspects because a particular choice will (somewhat) uniformly affect the design of many architectural elements, possibly in different views.

Capture. The architect captures aspects in the architecture document, which we can structure

as a set of views plus information that ties the views together.¹¹ If architectural aspects apply across a set of elements, whether in a single view or across views, we can factor them out and document them separately—in an *aspect view*. We document aspects in the aspect view using the same languages and notations that are used to describe the architecture’s corresponding nonaspectual parts—for example:

- If an aspect captures behavior common to a set of elements, we can represent the behavior using whatever language the architect is using to capture elements’ nonaspectual behavior: statecharts, message sequence charts, Z specifications, and so on.
- If an aspect captures common substructure, the architect represents the substructure in the same way that other structural information is captured: UML class diagrams, for example.
- If an aspect captures parts of an interface common to a set of elements, the architect represents that in the same way that other interfaces are captured: IDL, for example.

Conventional views have places (sections in a documentation template) to show structure, behavior, interfaces, and so on. An aspect view can use exactly the same organization. The only difference is that aspects will be abstract with respect to specific architectural elements. For example, in an aspect view, behavior is specified not for an element but for a set of elements to be named later. Each aspect should be documented with a rationale that identifies its origins (which of the identification activities outlined previously led to its discovery).

In the banking example, requirements aspects emphasized to the architect that every transaction must run to completion and maintain an audit history. The architect, when designing architectural elements to carry out transactions, may choose to impose a common internal structure and common behavior on each one to achieve those requirements in a consistent and conceptually coherent fashion. These become the architectural aspects. The aspect view would show an abstract element (standing for any transaction element), that element’s substructure, those subelements’ interfaces, and how they behave, all to assure transaction completeness and capture audit history.

Compose. In addition to producing views, an architect must produce documentation that explains how views are related. Suppose our banking application’s architecture is built as a client-server system. A server might represent a single implementation unit—we code it once—but several execution units if the banking system is to have multiple servers running. Somewhere in the documentation, the architect must map the (single) server in a module view to the multiple servers in a component-and-connector view.

Adding aspects to the mix requires establishing a mapping between the elements of an aspect view (the architectural aspects) and the elements or substructures in the different views to which they apply. This is the conceptual analog to join points. In the banking example, the mapping would establish the correspondence between the transaction elements and the aspects showing their common substructure and behavior.

Analyze. Architectural analysis in the early-aspect context, in addition to its conventional purpose of assessing the architecture’s fitness for a prescribed purpose, includes evaluating the suitability of the aspects that the architect has identified. The Architecture Tradeoff Analysis Method can readily accommodate architectural aspects into its analysis flow.¹² To the ATAM, an aspect’s creation is simply an architectural decision like any other to be evaluated in the context of prevailing functional and quality attribute goals. The Aspectual Software Architecture Analysis Method (ASAAM) includes steps to help identify aspects after the architecture has been designed.¹³

The flow of early aspects: Later use

Up to this point, we’ve looked at why and how to apply aspects in requirements and architecture. However, early aspects also have relevance and importance when passed between stages and on to later stages of the life cycle—for example, for broader vision, improved traceability, and improved trade-off handling and negotiation.

Broader vision

Early aspects from both requirements and architecture offer valuable insight in the implementation phase. They provide an improved localized description of stakeholder

Early aspects also have relevance and importance when passed between stages and on to later stages of the life cycle.

Table 2

Identifying and capturing early aspects

Life-cycle phases	Early-aspects activities				
	Getting aspects from requirements	Getting aspects from architecture	Identifying aspects	Capturing and composing aspects	Passing aspects on
Overall	—	—	Start with understanding the artifacts' basic decomposition. Then, look for broadly scoped properties and signs of scattered concerns and identify artifacts that tangle concerns.	Capture aspects in requirements by specifying and capturing them separately and by specifying their composition. If possible, rewrite documentation to better align with aspects. Otherwise, capture aspects and their impact separately.	Use identified and captured aspects to drive implementation, maintenance, and incremental development.
Requirements engineering	—	Aspects that arise in architecture might be evident earlier. Feed them back to the requirements activity for possible incorporation. For instance, use them to guide keyword or property searching among requirements. Or they might suggest the need for a new requirements artifact.	Look for common terms, requirements that influence other requirements, and requirements that tangle two or more concern terms. Some requirements describe the impact of one concern on another.	Untangle all requirements that you can. Reorganize them so that each concern is described in its own section or location. For the rest (the aspects), separate the aspect descriptions from their corresponding impact requirements. Impact requirements are used to more formally specify how the aspect should be composed with other concerns.	Concerns and impact requirements are given to architecture design and motivate the formation of views. Use concerns and impact requirements to guide implementation: requirements aspects may (after passing through an architecture/design phase) become code-level aspects, and requirement compositions become join points. Composition and analysis of requirements aspects provide early insights into architectural trade-offs.
Architecture design	Examine the concerns, impact requirements, and trade-off decisions made to resolve concern conflict. Treat a requirements aspect as a candidate for an architectural element whose responsibility will address that concern.	—	Look for concerns in the business case; identify architecturally significant requirements; choose patterns and tactics (design patterns describe how a particular design choice affects many architectural elements); and identify common policies, substructures, interfaces, and behaviors. For instance, we discussed examining multiple use cases or architectural elements for commonalities.	Extract aspects from views to form a view of their own. Use the languages and notations used to capture conventional views to capture the aspect view. Map the aspects and the views and elements to which they apply to one another.	Architectural views guide implementation, system testing, integration, incremental development, and maintenance. Architectural aspects become candidates for implementation aspects. Use the mappings between views when implementing pointcuts using an AOP language.

needs by addressing the crosscutting concerns, as gathered by domain experts, requirements engineers, and architects. These are roles whose scope, vision, and experience extend beyond implementation. These people, more than programmers, are guided by broader life-cycle and organizational concerns. They're in a good position to learn from the past and reuse the ac-

quired knowledge to design better applications. They respond to an organization's business goals, from which programmers might be largely insulated. They glean concerns directly from stakeholders, with whom programmers seldom interact. They have insights into the entire application domain, whereas implementers tend to focus on only the system at hand.

Improved traceability

Aspects in requirements often give rise to aspects in architecture, and then in implementation. For instance, aspects at the requirements level can be mapped onto design aspects, a function or component, or an architectural decision.⁴ By identifying and managing aspects throughout the life cycle, every aspect that ends up in the implementation has a firm pedigree. Either the implementers created it, or the architecture imposed it. Architectural aspects, in turn, are documented showing their origins, one of which could be requirements aspects. Hence, we can trace every aspect throughout the system's life cycle back to its origins, thus providing the insight necessary to effectively manage the aspect as the system evolves.

Improved trade-off handling and negotiation

Identifying aspects at each phase automatically reveals the scope of the concern, whereas a concern scattered throughout might be overlooked or have its impact underestimated or overestimated. This enables trade-off handling and negotiation between phases on an informed, rather than a speculative, basis. For example, an architect who wishes to "push back" on a difficult requirement will know the scope of that requirement exactly.

Several approaches are available for incorporating aspects in the requirements and architecture phases of software development. Some existing approaches explicitly include aspects; for others, we can straightforwardly add them in. An all-encompassing integrated approach doesn't yet exist. Given the ever-increasing role that aspects play in software development and their new and compelling role in the early design phases, we can hope for such a methodology in the future. But the lack of such a packaged solution need not deter us from gaining the advantages of working with early aspects. At each stage, as table 2 illustrates, we can identify and capture them and pass them on.

The full potential of early aspects is only now being glimpsed, but indications are that it could become a valuable resource for software system designers. Toward this end, a series of Early Aspects workshops (www.early-aspects.net) began in 2002. They're now a primary

About the Authors



Elisa Baniassad is an assistant professor at the Chinese University of Hong Kong. Her research interests include innovating and conducting empirical studies on visualization techniques related to aspect-oriented requirements engineering and establishing rationale relationships between high-level documentation and source code. She is coauthor of *Aspect Oriented Analysis and Design: The Theme Approach* (Addison Wesley, 2005). She received her PhD in computer science from the University of British Columbia. Contact her at the Dept. of Computer Science and Eng., Ho Sin Hang Bldg., Chinese Univ. of Hong Kong, Shatin, New Territories, Hong Kong, SAR, China; elisa@cse.cuhk.edu.hk.

Paul C. Clements is a technical staff senior member at Carnegie Mellon University's Software Engineering Institute, where he has led projects in software architecture and software product line engineering. He received his PhD in computer sciences from the University of Texas at Austin and is coauthor of *Software Architecture in Practice* (Addison-Wesley, 2003), *Documenting Software Architectures* (Addison-Wesley, 2002), and *Evaluating Software Architectures* (Addison-Wesley, 2001). Contact him at Software Eng. Inst., Carnegie Mellon Univ., Pittsburgh, PA 15213; clements@sei.cmu.edu.



João Araújo is an assistant professor at the New University of Lisbon. His research interests include aspect-oriented software development, requirements engineering, and model-driven development. He received his PhD in computer science from Lancaster University. He is a co-organizer of the Early Aspects workshop series. Contact him at Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Quinta da Torre, 2829-516 Caparica, Portugal; ja@di.fct.unl.pt.

Ana Moreira is an assistant professor at the New University of Lisbon. Her research interests include aspect-oriented requirements engineering and design, model-driven engineering, and object-oriented modeling. She is an editorial board member of *Software and Systems Modeling and Transactions on Aspect-Oriented Software Development* and is the steering committee chair for the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006). She is also a co-organizer of the Early Aspects workshops series. Contact her at Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal; amm@di.fct.unl.pt.



Awais Rashid is a senior lecturer in Lancaster University's Computing Department, where he leads research in aspect-oriented software engineering. His research interests include aspect-oriented requirements engineering, aspect-oriented databases, and object data management. He is the author of *Aspect-Oriented Database Systems* (Springer, 2004) and a founding co-editor in chief of *Transactions on Aspect-Oriented Software Development*. He also coordinates the European Network of Excellence on AOSD, funded by the European Commission. Contact him at Computing Dept., Infolab21, South Dr., Lancaster Univ., Lancaster LA1 4WA, UK; awais@comp.lancs.ac.uk.

Bedir Tekinerdoğan is an assistant professor in the Software Engineering Group at the University of Twente, where he leads the software architecture design group. His research interests include aspect-oriented software development, software architecture design, model-driven engineering, and software product line engineering. He's also a co-organizer of the Early Aspects workshop series. Contact him at Univ. of Twente, Dept. of Computer Science, Software Eng., PO Box 217 7500 AE, Enschede, NL; bedir@cs.utwente.nl.



gathering place for researchers and practitioners interested in this novel concept. Seven workshops have been held and two more are scheduled for major conferences in 2006. Work on individual pieces of this integrated approach has shown that early aspects work in practice in both industrial and pedagogical settings.^{3,5,6,13} We hope that this article has



<http://dsonline.computer.org>

IEEE Distributed Systems Online


brings you peer-reviewed articles, detailed tutorials, expert-managed topic areas, and diverse departments covering the latest news and developments in this fast-growing field.

Log on for **free access**

to such topic areas as

- **Grid Computing**
- **Mobile & Pervasive**
- **Distributed Agents**
- **Security**
- **Middleware**
- **Parallel Processing**
- **Web Systems**
- **Real Time & Embedded**
- **Dependable Systems**
- **Cluster Computing**
- **Distributed Multimedia**
- **Distributed Databases**
- **Collaborative Computing**
- **Operating Systems**
- **Peer to Peer**
- **Software Engineering**

To receive regular updates, email
dsonline@computer.org

shown that early aspects can play an important and integrated role throughout the life cycle. 

References

1. I. Brito and A. Moreira, "Integrating the NFR Framework in a RE Model," presented at Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, 2004, <http://trese.cs.utwente.nl/workshops/early-aspects-2004/Papers/BritoMoreira.pdf>.
2. A. Sampaio et al., "EA-Miner: A Tool for Automating Aspect-Oriented Requirements Identification," *Proc. Int'l Conf. Automated Software Eng. (ASE 05)*, ACM Press, 2005, pp. 352-355.
3. E. Baniassad and S. Clarke, "Theme: An Approach for Aspect-Oriented Analysis and Design," *Proc. 26th Int'l Conf. Software Eng. (ICSE 04)*, IEEE CS Press, 2004, pp. 158-167.
4. A. Rashid, A. Moreira, and J. Araújo, "Modularisation and Composition of Aspectual Requirements," *Proc. 2nd Int'l Conf. Aspect-Oriented Software Development (AOSD 03)*, ACM Press, 2003, pp. 11-20.
5. J. Araújo, J. Whittle, and D. Kim, "Modeling and Composing Scenario-Based Requirements with Aspects," *Proc. 12th IEEE Int'l Requirements Eng. Conf. (RE 04)*, IEEE CS Press, 2004, pp. 58-67.
6. A. Moreira, A. Rashid, and J. Araújo, "Multi-Dimensional Separation of Concerns in Requirements Engineering," *Proc. 13th IEEE Int'l Requirements Eng. Conf. (RE 05)*, IEEE CS Press, 2005, pp. 285-296.
7. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed., Addison-Wesley, 2003.
8. P. Clements et al., *Documenting Software Architectures: Views and Beyond*, Addison-Wesley, 2002.
9. F. Buschmann et al., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.
10. M. Jazayeri, A. Ran, and F. Van Der Linden, *Software Architecture for Product Families: Principles and Practice*, Addison-Wesley, 2000.
11. P. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, vol. 12, no. 6, 1995, pp. 42-50.
12. P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2001.
13. B. Tekinerdogan, "ASAAM: Aspectual Software Architecture Analysis Method," *Proc. 4th Working IEEE/IFIP Conf. Software Architecture (WICSA 04)*, IEEE CS Press, 2004, pp. 5-14.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.