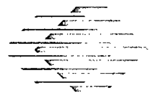


Carnegie Mellon
Software Engineering Institute

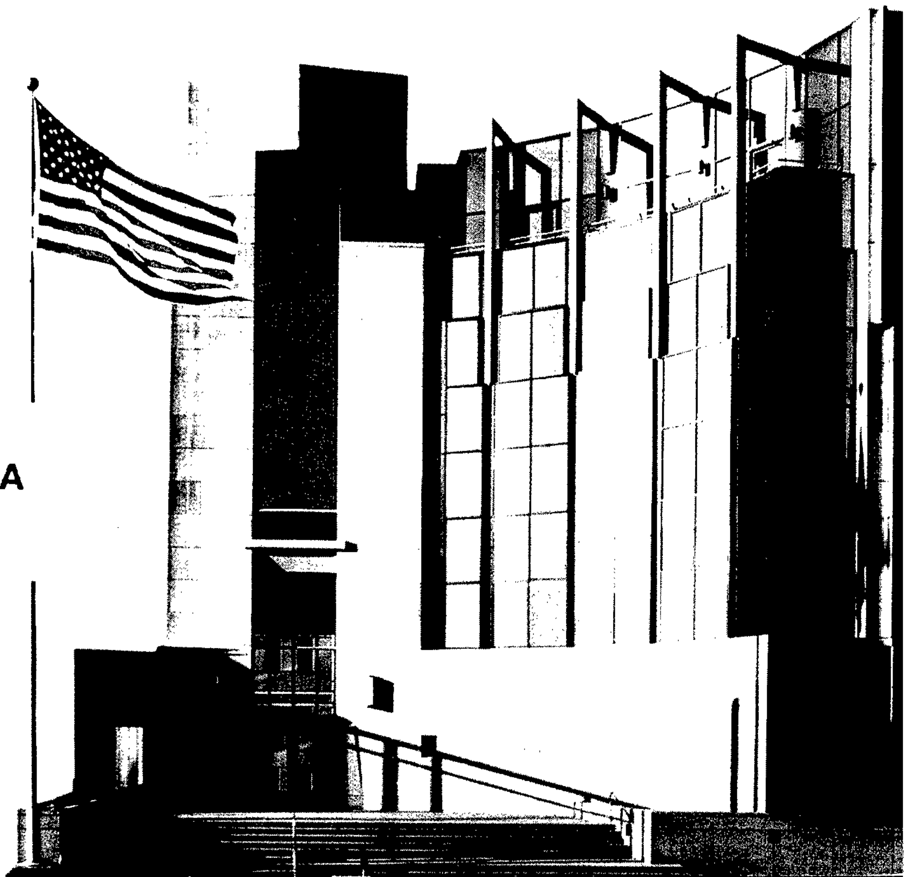
Integrating Software-Architecture-Centric Methods into Extreme Programming (XP)

Robert L. Nord
James E. Tomayko
Rob Wojcik

September 2004

Software Architecture Technology Initiative

Unlimited distribution subject to the copyright.



Technical Note
CMU/SEI-2004-TN-036

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

Integrating Software-Architecture-Centric Methods into Extreme Programming (XP)

Robert L. Nord
James E. Tomayko
Rob Wojcik

September 2004

Software Architecture Technology Initiative

Unlimited distribution subject to the copyright.

Technical Note
CMU/SEI-2004-TN-036

20050323 035

This work is sponsored by the U.S. Department of Defense.

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2004 Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Contents

Acknowledgements vii

Abstract ix

1 Introduction 1

2 Software Architecture and XP 5
- 2.1 Example: An ATM System 5
  - 2.1.1 Eliciting Requirements 5
  - 2.1.2 Creating a Design 6
- 2.2 The ATM Example Revisited 8
  - 2.2.1 Eliciting Quality Attribute Scenarios 8
  - 2.2.2 Creating and Evaluating a Design 10
- 2.3 Summary 12

3 Identifying Requirements: User Stories and the QAW 14
- 3.1 The QAW 14
- 3.2 The QAW and XP 15
- 3.3 Reflections 16

4 Using the ADD Method to Form an Architecture Design 18
- 4.1 The ADD Method 18
- 4.2 ADD and XP 19
- 4.3 Reflections 20

5 Evaluating Architecture with the ATAM and CBAM 22
- 5.1 The Integrated ATAM/CBAM 22
- 5.2 The ATAM/CBAM and XP 23
- 5.3 Reflections 24

6 Evaluating Intermediate Designs with ARID 25
- 6.1 ARID 25
- 6.2 ARID and XP 25
- 6.3 Reflections 26

7 Summary 27

References 29

List of Figures

Figure 1: Software Architecture Axioms 1

Figure 2: The Manifesto for Agile Software Development 2

Figure 3: Life-Cycle Activities of Architecture-Centric Development 4

Figure 4: A Simple Design (Notation: UML) 7

Figure 5: A Candidate Architecture—Deployment View (Notation: UML) 8

Figure 6: A Candidate Architecture Revised Using ADD—Deployment View (Notation: UML) 11

Figure 7: QAW Inputs, Outputs, and Participants 14

Figure 8: ADD Inputs, Outputs, and Participants 18

Figure 9: The Combined ATAM/CBAM Inputs, Outputs, and Participants 22

Figure 10: ARID Inputs, Outputs, and Participants 25

List of Tables

Table 1: QAW and XP Practices 16

Table 2: The ADD Method and XP Practices 20

Table 3: ATAM/CBAM and XP Practices 24

Table 4: ARID and XP Practices 26

Table 5: The Architecture-Centric Methods and XP Values 27

Table 6: The Architecture-Centric Methods and XP Activities 28

Acknowledgements

We would like to thank Felix Bachmann, Tony Lattanze, Paulo Merson, and Linda Northrop of the Carnegie Mellon® Software Engineering Institute, Orit Hazzan of the Technion-Israel Institute of Technology, and Jennifer Engleson of Carnegie Mellon's School of Computer Science for their review.

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

Abstract

This technical note fits the architecture-centric methods of the Carnegie Mellon® Software Engineering Institute (SEI) into the framework of Extreme Programming (XP). These methods include the Architecture Tradeoff Analysis Method®, the SEI Quality Attribute Workshop, the SEI Attribute-Driven Design method, the SEI Cost Benefit Analysis Method, and SEI Active Reviews for Intermediate Design. This report presents a summary of XP and examines the potential uses of the SEI's architecture-centric methods.

1 Introduction

For the past 10 years, the Software Architecture Technology Initiative[1] at the Carnegie Mellon® Software Engineering Institute (SEI) has developed and promulgated a series of architecture-centric methods, starting with the Software Architecture Analysis Method (SAAM) [Kazman 96] and continuing with the Architecture Tradeoff Analysis Method® (ATAM®) [Clements 02], the Quality Attribute Workshop (QAW) [Barbacci 03], the Attribute-Driven Design (ADD) method [Bass 03], the Cost-Benefit Analysis Method (CBAM) [Kazman 02], and Active Reviews for Intermediate Designs (ARID) [Clements 02]. These methods are predicated on the axioms shown in Figure 1.

1. Quality attribute requirements drive the software architecture.
   - Quality attribute requirements stem from business/mission goals.
   - Scenarios are a powerful way to characterize quality attributes and represent stakeholder views.
2. Architecture-centric activities drive the software system life cycle.
   - These activities must have an explicit focus on quality attributes.
   - These activities must involve stakeholders directly.

Figure 1: Software Architecture Axioms

At the same time, the SEI has disseminated a wealth of architectural knowledge and practical expertise via its books [Bass 03, Clements 03, Clements 02] and papers. These efforts have now culminated at the point where the SEI is pursuing their integration by (1) combining related methods [Kazman 03, Nord 03, Nord 04] so they work more synergistically and (2) fitting the architecture-centric methods into popular processes of software development [Kazman 04].

One category of these popular development processes is agile software development. Agile refers to a paradigm of software development that emphasizes rapid and flexible development and de-emphasizes project and process infrastructure for their own sake. In 2001, 17 leaders of the Agile Movement signed the manifesto shown in Figure 2 [Agile 01].

[1] Formerly called the Architecture Tradeoff Analysis Initiative.

® Carnegie Mellon, Architecture Tradeoff Analysis Method, and ATAM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

1. **Individuals and interactions** over processes and tools
2. **Working software** over comprehensive documentation
3. **Customer collaboration** over contract negotiation
4. **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler
James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick
Robert C. Martin
Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas

© 2001, the above authors
This declaration may be freely copied in any form, but only in its entirety through this notice.

Figure 2: The Manifesto for Agile Software Development

As an example of agile software development, customers and developers are free to interact constantly, rather than only at times set by processes or during the silence caused by some tools. This freedom is needed because one of the basic principles of agile methods is frequent and early delivery of something valuable to customers.

Projects get built by self-motivated coders with the best tools, and progress is measured by the amount of code written by developers that contributes to delivered functionality. Change is saved for face-to-face conversation. The goal is for all persons to maintain a sustainable pace while providing input to any parts of the project, as needed.

Agility is enhanced by technical excellence and good, simple design. Simplicity is a means of developing those functions of the software that are needed today and putting off those needed for later iterations. The project team is self-organized. When the team notices that it is losing quality, it reorganizes itself.

A full treatment of agile methods is beyond the scope of this report. Examples of methods in use include Scrum [Schwaber 02], Extreme Programming (XP) [Beck 04], Crystal [Cockburn 01], Adaptive Software Development (ASD) [Highsmith 00], and Feature-Driven Development (FDD) [Palmer 02].

The XP method has quite a following. We chose it as the focus of this report because it is one of the most mature (dating to the mid-1990s) and best known of the agile processes. It got the name "extreme" from its tendency to "turn up the volume" of its practices. Even a cursory

look at the practices listed below shows that most are part of many methods, but, in XP, they are taken to the extreme.

XP practices are based on four values: (1) communication, (2) simplicity, (3) feedback, and (4) courage:

1. Communication emphasizes person-to-person talking, rather than documents that explain the software. Also, several of the XP practices call for an on-site customer, so a lot of time is spent communicating to that customer.
2. Simplicity is an XP value that calls for the solution of the customer's problem to be unpretentious. Both developers and customers easily understand the software solution. If it is not obvious, that solution is a candidate for another practice called Refactoring (one of the core XP practices) [Fowler 99]. Certainly, Simple Design (one of the core XP practices) is a practice that contributes to the value of simplicity.
3. Feedback means that everything done is evaluated with respect to how well it works. How well it works is indicated through the feedback gained from exercising every working part of the solution. Feedback, which is the result of Nyquist's work at Bell Laboratories in the early 1930s, is used to determine if the solution is correct. Since engineers began studying feedback, it has been one of the prime ways of guiding solutions.
4. Courage means that developers are prepared to make important decisions that support XP practices while building and releasing something of value to the customer each iteration. This could mean discarding code or taking the time to refactor the design when it turns out that certain decisions prove inadequate.

The 4 core values are implemented with 12 core practices: (1) Planning Game, (2) Pair Programming, (3) System Metaphor, (4) Simple Design, (5) Coding Standards, (6) Collective Code Ownership, (7) Refactoring, (8) On-Site Customer, (9) Test-Driven Development, (10) Continuous Integration, (11) Small Releases, and (12) Sustainable Pace.[2]

This report concentrates on integrating architecture-centric methods into agile processes like XP. There is no definition of an architecture-centric development life cycle in the XP literature, so we use the list of architecture-centric activities developed by Bass, Clements, and Kazman shown in Figure 3 [Bass 03].

[2] Sustainable Pace was formerly referred to as the "40-hour week" until someone realized that many Europeans think of 35-hour weeks as a "sustainable pace." Even Carnegie Mellon University uses 37.5 hours for staff, though the authors know of no one that works such short hours. At any rate, specific numbers did not communicate the idea well [McBreen 02].

Architecture-centric development involves iteratively

- creating the business case for the system
- understanding the requirements
- creating or selecting the software architecture
- documenting and communicating the software architecture
- analyzing or evaluating the software architecture
- implementing the system based on the software architecture
- ensuring that the implementation conforms to the software architecture

Figure 3: Life-Cycle Activities of Architecture-Centric Development

In Section 2 of this report, we look at how XP develops software and at some of XP's shortcomings related to software architecture. In this section, we also introduce a case study for an automated teller machine as a means of exemplifying XP. Section 3 looks at requirements identification and suggests how the QAW might play a role in XP. In Section 4, we look at analysis and design and examine the potential place of the ADD method. Section 5 describes the potential role of the ATAM and CBAM, and Section 6 describes the use of ARID. Section 7 concludes this report with some reflections on the usefulness of augmenting XP in this fashion.

2 Software Architecture and XP

In this report, we first use an example of an automated teller machine (ATM) that is part of a bank automation system to understand how XP develops software and to describe some of XP's shortcomings related to software architecture. Then, we introduce the SEI's architecture-centric methods using the same ATM example to illustrate how they complement XP and address those shortcomings.

2.1 Example: An ATM System

Let us first examine this example purely from the perspective of XP. In XP, the customer records the required system functionality at the beginning of each development iteration. The requirements are embodied in *user stories*.[3] The client and the developers together determine the functionality that will be developed for the current iteration during the Planning Game practice.

2.1.1 Eliciting Requirements

The customer generally develops the user stories—stories about how the software is to be used or how it will work. User stories also contain descriptions of test situations. For example, user stories from the perspective of the "ATM customer" would include activities such as using the ATM to withdraw money, deposit money, transfer money between accounts, or check the balance of a bank account. These stories tell us what the user wants, but avoid saying how to do it. Figuring out "how" is left to the engineer. Stories may also show up later as changes affect the system.

The customer and developers share information to determine which stories to implement during the Planning Game practice and to produce a schedule. The customer usually goes first; he or she determines the next increment of functionally that will be of value to the user. The stories for this iteration, which usually appear on cards, are selected and given to the developers. The developers then figure out how long each story will take to build, adjusting this estimate with a concept called *velocity*.

Velocity is idiosyncratic to organizations and people; it means how much work a person can accomplish in a day. Time for things such as all-hands meetings and bathroom stops are subtracted from the working day, yielding the true average working day—the velocity. For

[3] These stories were once called *customer stories*. We think they were renamed after some practitioner of agile methods realized that customers aren't necessarily users [McBreen 02].

instance, if you can count on an average of two hours of non-software production activities, your velocity would be approximately six hours per day.

Each card is marked with how long the story will take to build. The cards are collated and examined against the length of this iteration (usually about two weeks). The stories that can be done in the time allowed are returned to the customer, who chooses the highest priority ones for implementation. The ones that are too long are returned as well, to be further broken up, possibly for the next round. This is step two of the Planning Game practice. The third step is the return of the highest value items to the developers. In the final step, the developers deliver a schedule to the customer.

Because XP emphasizes verbal communication, user stories may not fully capture the requirements; much of the latter will live in the heads of the developers.

2.1.2 Creating a Design

The first XP practice that influences the architecture is System Metaphor. The original purpose of this practice was to provide both the customers and developers a simple description of what is in the design and a point on which they could agree to talk.

System Metaphor is the least used practice of XP. Using a system metaphor is not a mandatory step of the XP process. If no agreement can be reached with the customers about the metaphor, it is frequently replaced as a practice with the stand-up meeting normally associated with Scrum [Schwaber 02].

Many have reported on the difficulty of coming up with a good system metaphor. Recently, at a major conference, XP originator Kent Beck left it up to a vote as to whether to continue with that task. Jim Tomayko, one of the authors of this report, also shared this experience and reported on the difficulty finding metaphors [Herbsleb 03]. He was about to join those who gave up on metaphors until he found considerable literature on metaphors in other fields and an article in the XP literature that made it easier to define metaphors. David West offers a metaphor for the system architecture as an interpretation of the Mandala architecture, which is of Tibetan Buddhist origin [West 02].

When it is difficult to come up with a meaningful system metaphor, a minimalist or naïve metaphor can be used; for example, a human bank teller for the ATM example. Such a person follows devised procedures and is aware of constraints. Whether naïve or detailed, the metaphor is the first notion of an architecture. One limitation of the naïve metaphor is that it represents what the Rational Unified Process (RUP) calls the *logical view* of the architecture [Kruchten 04]—the key objects of the systems and their relationships; this metaphor does not address other important views such as concurrency and deployment.

The second XP practice that has some influence on architecture is Simple Design, which means that each component will be easy to understand. Design patterns may be used to create a simple design. The candidates for refactoring are here in the design as well. The design is usually drawn on a whiteboard, so it's easily changeable.

Achieving simplicity in the design is a challenge. Simplicity is not limited to the Simple Design practice: it is one of the core values of XP and, as such, is important across all the practices. Thus, simplicity is the motivation for the Simple Design practice and a challenge to many people, especially those who, in complexity, find both comfort and a place to hide things.

Returning to our ATM problem while applying these practices, we settle on the story that includes "balance inquiry" as the first release. This story offers some value to users in that they can check their account balance even when they're far from home. Following the naïve metaphor of a teller, users can get their account balances easily, but looking up balance after balance becomes crushingly boring for the tellers. Boredom is not a problem for machines. These requirements and constraints begin to drive the architecture. Depositing, transferring, and withdrawing funds from the account are well-known capabilities that will be added later via special components that will all need access to the account balance. The relationships between those components are shown in Figure 4.

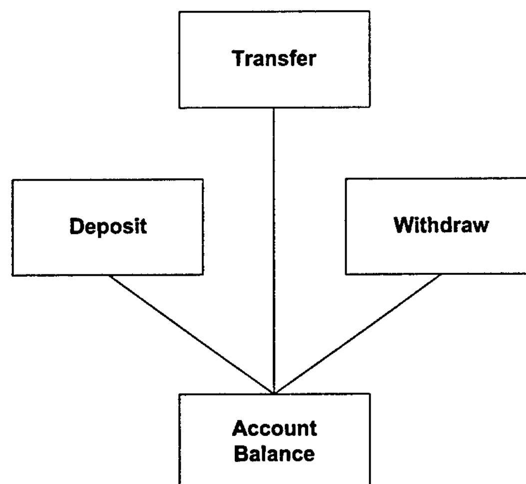


Figure 4: A Simple Design (Notation: UML)

During the Planning Game practice, the developers may elect to perform a *spike* to explore the implications of implementing the user stories with new technologies. A *spike* consists of building a simple prototype that allows the team to understand the new technology. Besides functional spikes, spikes related to quality attributes can be performed to guide the design of a deployment view, which supplements the naïve metaphor that represents a logical view.

Since the user can check balances remotely, the team needs to explore the implications of an N-tier design. A candidate architecture might be posited by the team based on its experience and structured using the three-tiered, client-server, and repository styles (see Figure 5). The ATM is, thus, a client of a transaction-processing system.

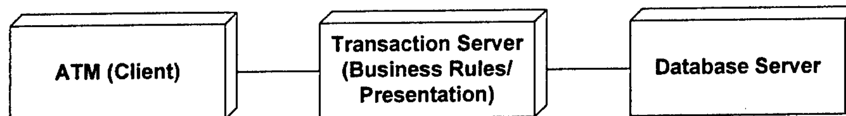


Figure 5: A Candidate Architecture—Deployment View (Notation: UML)

XP doesn't address evaluating the design explicitly. Code is tested continually through the practices of Test-Driven Development and Continuous Integration.

2.2 The ATM Example Revisited

Let us now revisit our example with the aim of showing how the SEI's architecture-centric methods can enhance XP. A candidate architecture, such as the (primitive) one shown in Figure 4 and Figure 5, is a product of the relevant user stories that have been identified. But this architecture is dependent, for its shape and quality, on the experience of the development team. The SEI architecture-centric methods can inform and regularize this process. They emphasize quality attributes and focus early on architectural decisions.

The QAW can help elicit quality attribute requirements in the form of quality attribute scenarios. The ADD method defines a software architecture by basing the design process on the prioritized quality attribute scenarios (architectural drivers) that the software must fulfill. The ADD method documents a software architecture using several views. ADD depends on an understanding of the system's constraints, as well as its functional and quality requirements, which are represented as six-part quality attribute scenarios. The ATAM, CBAM, and ARID provide detailed guidance on analyzing the resulting design.

2.2.1 Eliciting Quality Attribute Scenarios

The feature requirements of the ATM are those that are desired by the customer and are part of the user stories. However, other qualities of the system are important in addition to its functionality; for example, performance requirements for the ATM that specify the customer must get a response from the system in less than 10 seconds. Let's assume there are also availability requirements that say the system must be available 24 hours a day, 7 days a week, except for a 15-minute "service time" period each day. Furthermore, the system must be able to recognize and report faults within 30 seconds of their occurrence. Security properties dictate that the transaction must be authorized properly and communicated securely to the bank's database. In addition, modifiability properties dictate that the system must be easily

changeable to take advantage of new platform capabilities (for example, it must not be tied to a single database or a single kind of client hardware or software) and must be extensible to allow new functions and business rules to be added.

The QAW can help elicit quality attribute requirements in the form of quality attribute scenarios that complete the set of functional requirements and constraints. A *quality attribute scenario* is a quality-attribute-specific requirement [Bass 03] that consists of six parts:

1. *stimulus:* a condition that needs to be considered when it arrives at a system
2. *source of the stimulus:* the entity (an actor) that generated the stimulus
3. *artifact stimulated:* Some system artifact is stimulated by the stimulus. This artifact may be the entire system or some portion of it.
4. *environment:* The stimulus occurs within a specified context. For example, the system may be in a normal state, a degraded mode, or in an overload condition when the stimulus occurs.
5. *response:* the activity undertaken when the stimulus arrives
6. *response measure:* When the response occurs, it should be measurable in some fashion, so the quality attribute requirement can be tested.

Scenarios, as elicited and elaborated in the architecture-centric methods, are very similar to use cases: they indicate what must be present, what is done, and what the outcome will be. Therefore, use cases and scenarios can and should be developed simultaneously. Essentially, the scenarios inspire and are inspired by use cases. The difference is that quality attribute scenarios always include the six elements above and, hence, are always focused on the elicitation and documentation of quality-attribute-specific information. Scenarios may mention functionality, but that is not their point. Rather, their point is that the six elements embody the quality attribute requirements, and those requirements inspire and shape an architecture. Simply put, the architecture is determined by the quality attribute requirements, not by the functionality. Returning to our example, a quality attribute (performance) scenario that corresponds to the use case for an ATM is as follows: "The user can withdraw a limit of $300 from an account that has sufficient funds in less than 10 seconds." The scenario has two functional requirements and one performance requirement. One function is a withdrawal, and one is a limit (a constraint of $300 if it is in the account). The scenario also has a performance constraint of "less than 10 seconds," which is a quality attribute. Typically, a use case would not include such a performance constraint.

The SEI's architecture-centric methods provide several techniques for eliciting scenarios. The QAW, for example, is a facilitated method that engages system stakeholders early in the life cycle to discover the driving quality attributes of a software-intensive system and to record them in the form of scenarios in the six-part documentation structure outlined above. In addition to the brainstorming activity during the QAW, the architecture-centric methods elicit and capture quality attribute scenarios in two other ways: (1) through general-scenario-

generation tables and (2) through utility trees. Both ways are described by Bass and colleagues [Bass 03].

2.2.2 Creating and Evaluating a Design

One quality attribute requirement mentioned earlier is that the system must be easily modifiable to take advantage of new platform capabilities (such as a new database or client) and extensible to allow new functions and business rules to be added. Through the process of the QAW, this vague requirement would be refined into several six-part scenarios. For example, the following modifiability scenarios would be typical of an ATM system:

- A developer wants to add a new auditing business rule at design time in 10 person-days without affecting other functionality.
- A developer wants to change the relational schema to add a new view to the database in 30 person-days without affecting other functionality.
- A system administrator wants to employ a new database in 18 person-months without affecting other functionality.
- A developer wants to add a new function to a client menu in 15 person-days without causing any side effects.
- A developer needs to add a Web-based client to the system in 90 person-days without affecting the functionality of the existing ATM client.

To achieve these modifiability requirements, one or more architectural tactics will need to be employed. An architectural tactic is a means of satisfying a quality-attribute-response measure (such as average latency or mean time to failure) by manipulating some aspect of a quality attribute model (such as performance-queuing models or reliability Markov models) through architectural design decisions [Bachmann 02]. In this way, tactics provide a "generate and test" model of architectural design. The ADD method defines a software architecture by basing the design process on the high-priority quality attribute requirements of the system. The ADD approach follows a recursive decomposition process where, at each stage in the decomposition, architectural tactics and patterns are selected to satisfy a chosen set of high-priority quality scenarios.

In the case of modifiability, relevant architectural tactics include *Localize Changes* and *Use an Intermediary*. The *Localize Changes* tactic suggests that the business rules, database, and client should be localized into components, and the *Use an Intermediary* tactic suggests that these components should be separated to insulate them from potential changes in each other. A three-tier client-server model (shown in Figure 5) would emerge from the application of the *Localize Changes* tactic, since this architecture allocates the client, database, and business rules to their own tiers and, hence, localizes the effects of any changes to a single tier. The *Use an Intermediary* tactic suggests that the communication between the tiers be mediated by some abstract interface (such as a data access layer that uses Open Database Connectivity [ODBC] between the business rules and the database) and a translation layer between the

business rules and the client that understands the Extensible Markup Language (XML). The existence of such intermediaries makes it simple to add new databases or clients. For example, a developer can now add a Web-based client and server as a simple addition to the architecture, without affecting the ATM client.

To achieve the quality attribute requirement of a "10-second latency on a withdrawal" in the ADD method, a different set of architectural tactics is employed. Performance tactics are divided into three categories: (1) resource demand, (2) resource management, and (3) resource arbitration. Since we cannot control resource demand with an ATM (or, more precisely, because doing so would be bad for business), we must look towards managing and/or arbitrating the use of resources to meet performance goals. Some resource management tactics that are potentially applicable here are *Introducing Concurrency, Maintaining Multiple Copies of Either Data or Computations,* and *Increasing Available Resources.* By employing the *Introducing Concurrency* and *Increasing Available Resources* tactics, we may choose to deploy additional database servers and business rule servers or to make any of them multithreaded so they can execute multiple requests in parallel. Once we have multiple resources, we need some way of arbitrating among them, so we introduce a new component—a load balancer—that employs one of the resource arbitration tactics such as *Fixed-Priority Scheduling* or *First-In First-Out Scheduling*. This component will ensure that the processing load is distributed among the system's resources according to a chosen scheduling policy.

This leads us to the design shown in Figure 6: a slightly revised and elaborated version of the architecture initially presented in Figure 5. Obviously, both of these architectures are still simple, and much more work needs to be done to turn them into complete design specifications for development. The purpose of this example is not to show the entirety of a sophisticated architecture being developed, but rather to emphasize the difference in how we arrived at the architectures of Figure 5 and Figure 6. In the former, the architecture was created out of the architect's experience and knowledge. When using ADD, on the other hand, tactics and a structured set of steps provided design guidance for the creation and nature of each tier. In this way, each architectural structure is created via an engineering process that codifies experience and best practices (see Figure 6).

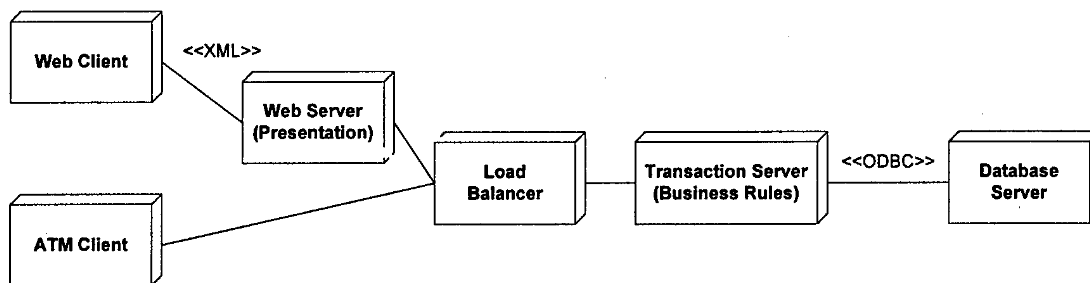


Figure 6: A Candidate Architecture Revised Using ADD—Deployment View (Notation: UML)

In this view, we have not yet specified the precise degree of replication of any deployed clients or servers, or the size of the thread pool in them. This more detailed specification is the next step in the design process. Once these characteristics have been specified, the latency characteristics of the architecture can be evaluated via a performance-queuing model. However, architectural decisions are complex and interact. For example, the degree to which changes in the database schema will affect the business rules, Web server, or client software also needs to be analyzed. Each abstraction layer (XML and ODBC) will mask some class of changes and expose others. And each layer will impose a performance cost. Similarly, the addition of a load-balancing component will create additional computation and communication overhead but provide the ability to distribute the load among a larger resource pool.

Because design decisions interact, we need a way to understand how those made during the creation of a complex system architecture will interact. The ATAM provides software architects with a framework for understanding the technical tradeoffs and risks they face as they make architectural design decisions. In addition, the CBAM helps software architects consider the return on investment (ROI) of any architectural decision and provides guidance on the economic tradeoffs involved. Finally, ARID evaluates whether the design can be used by the software engineers who must work with it.

2.3 Summary

In XP, the first iteration plays a crucial role in defining the overall structure of the system. "The first iteration puts the architecture in place. Pick stories for the first iteration that will force you to create 'the whole system,' even if it is in skeletal form" [Beck 04]. Modifiability is implied by XP, but it's hard to characterize. Developers grow the system incrementally, and when the system does not support new functionality, they refactor the design.

The SEI architecture-centric methods can provide explicit and detailed guidance on eliciting the architectural requirements (such as modifiability), on designing the architecture, and on analyzing the resulting design. In summary

- The architecture-centric methods place an emphasis on quality attributes rather than functionality. They also help facilitate communication.
- The architecture-centric methods help fill gaps in the XP design process, by providing specific advice on
  - the elicitation and documentation of quality attribute requirements
  - which design operation will achieve a desired quality attribute response
  - how to analyze the result to understand and predict the consequences of the design decisions in terms of risks, tradeoffs, and ultimately ROI

- The architecture-centric methods all use common concepts: quality attributes, architectural tactics, and a "Views and Beyond" approach to documentation that leads to more efficient and synergistic use [Clements 03].

Next, we look at the SEI methods in more detail and see how they can be integrated into the XP software development process.

3 Identifying Requirements: User Stories and the QAW

A Quality Attribute Workshop (QAW) can be held early when user stories are developed to elicit and analyze the quality attribute requirements in the form of scenarios. These requirements are not addressed explicitly in user stories but are often part of a hidden agenda within XP. Scenarios developed during the QAW can be refined further into user stories and be used to develop the Test Plan.

3.1 The QAW

The QAW is a facilitated method that engages system stakeholders early in the life cycle to discover the driving quality attribute requirements of a software-intensive system. The key points about the QAW are that it is system-centric, stakeholder focused, used before the software architecture has been created, and scenario based.

The QAW has its roots in, and was developed to complement, the ATAM. The QAW provides a way to identify important quality attributes and clarify system requirements *before* the software architecture has been created. This timing is perfect for XP. The QAW elicits, collects, and organizes software quality attribute requirements in the form of scenarios, which are converted to user stories.

Figure 7 provides a summary of the inputs, outputs, and participants of the QAW. This figure is based on a functional modeling notation [IEEE 98] where inputs flow in from the left, outputs flow out to the right, and participants in the method are noted at the bottom. More details about the QAW are described by Barbacci and colleagues [Barbacci 03].

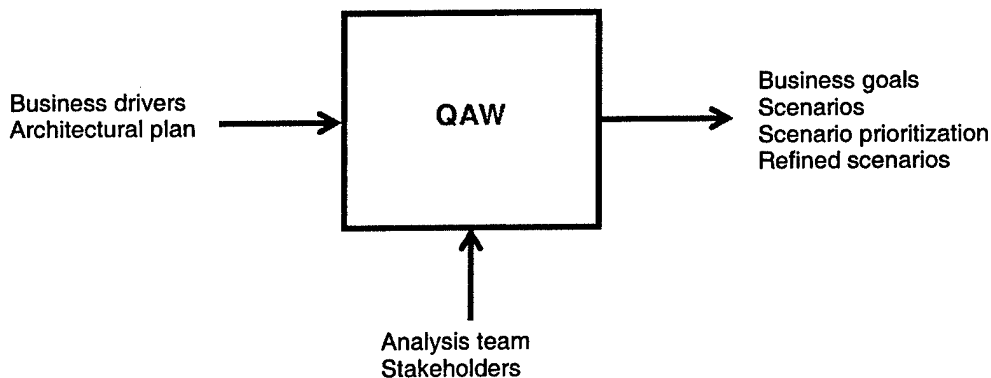


Figure 7: QAW Inputs, Outputs, and Participants

3.2 The QAW and XP

Requirements elicitation, capture, documentation, and analysis are accomplished in an XP project by developers acting as system analysts. After examining user stories at the beginning of each iteration, developers lead and coordinate requirements elicitation and modeling by outlining the system's functionality and delimiting the system. The result is a specification of details for one or more parts of the system's functionality.

A QAW, which can enhance this process, would be appropriate for the first iteration of an XP project and aid in identifying key system quality attributes. In a one-day workshop format, facilitators who do not play a stakeholder role are best used as QAW analysis team members. The stakeholders attending the workshop include the on-site customer and others with an interest in the system (e.g., end users, maintainers, project managers, members of the development team, testers, and integrators). An exemplary sample of scenarios is refined at the workshop. In subsequent iterations, additional scenarios can be elicited and refined as needed by developers in collaboration with the on-site customer.

The inputs to the QAW include the business drivers and the architectural plan. Business drivers include the business vision, goals, and key system quality attributes. The architectural plan contains information about development including known technical constraints such as an operating system (OS), hardware, or middleware prescribed for use; other systems with which the system must interact; key technical requirements that will drive architectural decisions; and existing context diagrams, high-level system diagrams, and descriptions. If not already documented, that information needs to be elicited from the customer. Some of it may be implicit in the user stories.

The outputs from the QAW feed into other practices in XP. For example, business goals are elicited and refined during the QAW and could be used by the customer to organize existing user stories, inspire additional user stories, or prioritize requirements along the lines of the customer's business needs. The scenarios can help determine what is in and out of the system's scope and can lead to the creation or refinement of the system context diagram or its equivalent. Scenario generation can also lead to the creation of use cases.

Typically, customers develop user stories for requirements and then work on acceptance test cases for the end of development. Many customers do not know how to build these test cases. The QAW can give them clues, if not encourage them to build the test cases. This way of using the QAW fits in with XP's "test first" or "build for the test" philosophy, as test cases are available to test—early in the development process—whether the code implements the requirements. These test cases can be built as code is being built, so the product of the software development test team can be checked at the end of development.

3.3 Reflections

The QAW complements XP practices as shown in Table 1.

Table 1: QAW and XP Practices

XP Practices	Value Added Through the QAW
Planning Game	User stories are supplemented with quality attribute information in the form of six-part scenarios. Scenario prioritization and refinement give additional information to the customer and developers to help them choose user stories for each iteration.
On-Site Customer	The single on-site customer is supplemented with additional stakeholders during a one-day workshop.
Test-Driven Development	Scenarios can be used later to evaluate the design and provide input for analysis during testing.

There is no explicit XP practice that corresponds to QAW scenarios. Unless otherwise directed, most stakeholders tend to focus on functionality, not on quality attributes. The QAW provides an explicit method for gathering quality attribute scenarios. The QAW approach favors augmenting use cases—or whatever technique the developers use to extract requirements from stories—with quality attribute information in the six-part scenario format. In addition to giving the developer guidelines for being more precise, this approach brings more prominence to the quality attributes and their role in shaping the architectural design.

On-Site Customer is one of the most criticized practices of XP. The idea is to speed communication with the customer by keeping them on-site and to have a more accurate accounting of changed requirements. Opponents figure that anyone the customer is willing to give up as a permanent loan to the development group is either no good or too technical. Stakeholders' points of view are not always easy to obtain; although on-site, the customer is sometimes removed from knowing the needs of end users and other stakeholders. The gathering of stakeholders that occurs during the QAW complements the on-site customer prescribed by XP. To be successful, the workshop needs to gather a wide group of stakeholders from the business organization. The QAW engages those stakeholders to discover and prioritize the quality attributes. The workshop setting facilitates open communication among the stakeholders and provides a forum where conflicts or tradeoffs among the requirements can be acknowledged and discussed.

In addition to the more immediate benefits cited above, the scenarios turned into user stories continue to provide benefits during later phases of development. Stakeholders' concerns and any other rationale information that is captured should be kept individually in a form that can be included in the appropriate architecture documentation—usually on a public whiteboard. Stories provide input for analysis throughout the life of the system and can be used to drive

the system's development. Scenarios can also help the on-site customer prepare the acceptance test suite that will grow with the product.

4 Using the ADD Method to Form an Architecture Design

The Attribute-Driven Design (ADD) method defines software architecture by basing the design process on the quality attributes the software must exhibit. The ADD method can and should be applied in the early iterations of XP to set the stage for a more complete architecture design.

4.1 The ADD Method

The ADD method is an approach to defining software architectures by basing the design process on the architecture's quality attribute requirements. ADD follows a recursive decomposition process where, at each stage in the decomposition, architectural tactics and patterns are chosen to satisfy a set of quality attribute scenarios.

The ADD method creates and documents a software architecture using a number of views. The nature of the project determines the views; most commonly one or more of the following are used: a module decomposition view, a concurrency view, and a deployment view [Clements 03]. The module view prescribed by the ADD method shows system partitioning and information exchange among modules. The concurrency view prescribed by the ADD method shows conceptual threads of control and synchronization relationships among design elements that are more abstract than the notion of active classes. The deployment view prescribed by the ADD method shows the allocation of responsibilities to the deployment environment.

The ADD method depends on an understanding of the system's constraints and its functional and quality requirements. Figure 8 provides a summary of the method's inputs, outputs, and participants. More details about the ADD method are provided by Bass, Clements, and Kazman [Bass 03].

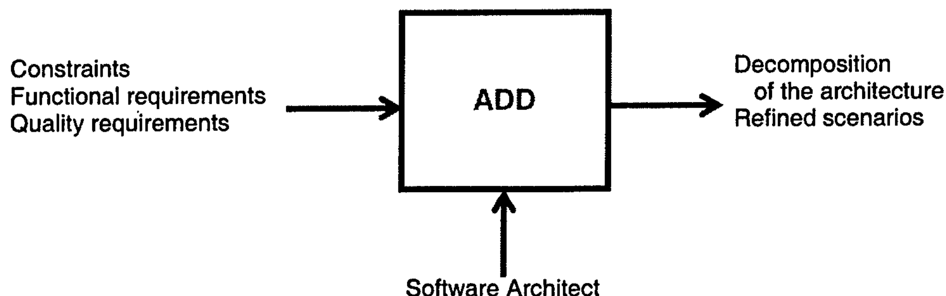


Figure 8: ADD Inputs, Outputs, and Participants

4.2 ADD and XP

Each development iteration in XP has analysis and design that ultimately lead to the software design. The key technical decisions that constrain the design and implementation of the project are figured out during these iterations. The developers might also construct an architectural proof-of-concept during the inception of the product.

The architecture is used by the developers to communicate with the customer and to come to a common understanding of how to build the system. Note that even though the architecture is produced as part of the first iteration, it is almost certain to be revisited in later phases (as more requirements are known) and could be modified in subsequent iterations.

ADD, which can enhance this process, is performed by the software architect and contributes to the initial software architecture design known as the candidate architecture. ADD is a specific design method with a detailed set of steps aimed at producing an architecture that both satisfies the desired qualities and business goals and provides the framework for realizing the desired functionality.

In the first half of the ADD method, the focus is on identifying the architectural drivers and producing an initial structure of the architecture that satisfies those desired qualities. The ADD method uses architectural tactics associated with quality attribute scenarios to help guide these activities. In the second half of the method, functionality (from the other requirements) is allocated to the structure identified in the candidate architecture. More detailed XP design activities begin where ADD ends.

The ADD method concentrates on something often ignored by XP developers—the overall system structure that is shaped by the quality attributes. This concentration should occur in the first iteration and recur in later iterations, as substantial changes or additions to the software architecture need to be explored.

A whiteboard in the workroom where everyone can see it easily is where the architecture, quality attributes, and constraints are maintained during the XP process. The architecture created as an output of ADD is a representation of the most important design choices. That architecture describes a system as containers for functionality and the interactions among them. Because it is the first articulation of the architecture during the design process, it is necessarily coarse-grained.

The inputs to the ADD method come from other XP products. The functional requirements are embodied in the user stories, constraints are in the architectural proof-of-concept that is the first thing on the board, and quality requirements can be elicited during a QAW.

The outputs include the initial decomposition of the architecture and refined scenarios that could be the focus of later iterations.

4.3 Reflections

The ADD method complements XP practices as shown in Table 2.

Table 2: The ADD Method and XP Practices

XP Practices	Value Added Through ADD
Planning Game	Building a utility tree to identify architectural drivers is useful in choosing user stories during the Planning Game practice.
Metaphor	The ADD method provides a step-by-step approach to defining the architecture in terms of module decomposition, concurrency, and deployment views.
Simple Design	The ADD method provides an architecture that's just course-grained enough to ensure that the design will meet its quality attribute requirements and to mitigate any associated risks. ADD defers all other architecture decision making.
Refactoring	Refactoring, which is driven by quality attribute needs (make it faster, make it more secure, etc.), is aided by the application of architectural tactics.

Boehm and Turner demonstrate a solution for adapting XP to develop complex, large-scale applications by introducing elements of plan-driven methods [Boehm 04]. These elements include high-level architectural plans to provide essential big-picture information and use of design patterns and architectural solutions rather than simple design to handle foreseeable change. Including architecture in this way might also delay refactoring. However, investing in the architecture means that it will take longer to get to code, because the first iteration is what some people call a "zero-feature release." In such a release, the architecture is put in place, but no user-visible features are delivered to the customer. ADD can provide this kind of architectural information.

Incorporating the ADD method into XP involves modifying the steps dealing with the high-level design of the architecture. XP design is guided by the principle of "you aren't going to need it," but when change can be anticipated, it makes sense to plan for it.

The ADD method supports both a breadth-first and depth-first decomposition approach to design. The order of decomposition will vary based on the business context, domain knowledge, and changing technology, for example. ADD would support an XP approach to design by allowing an initial breadth-first decomposition for the first decomposition level, followed by depth-first decompositions to explore the risks associated with change through prototyping.

The ADD method provides a step-by-step approach to defining course-grained architecture that can be evaluated by the ATAM and be used as a blueprint for implementation. Scenarios

and architectural tactics are critical to architecture design. ADD differs from XP core practices by its emphasis on addressing quality attribute requirements in an explicit way using architectural tactics. The quality attributes are what shape the structure of the architecture, with functionality being allocated to that structure.

The ADD method provides abstract notions of module, concurrency, and deployment views early on, allowing the developers greater flexibility and the opportunity to defer making more detailed decisions. These views are described in the "Views and Beyond" approach to documenting software architectures [Clements 03], which also describes a process for choosing appropriate views based on stakeholders' needs.

5 Evaluating Architecture with the ATAM and CBAM

A combined Architecture Tradeoff Analysis Method (ATAM) and Cost Benefit Analysis Method (CBAM) [Nord 03] is best done late in every XP iteration to ensure that the architecture is complete. This activity also highlights the risks that might be faced by further development, maintenance, and evolution.

5.1 The Integrated ATAM/CBAM

The ATAM provides software developers with a framework for understanding the technical tradeoffs and risks they face as they make design decisions. The purpose of the ATAM is to assess the consequences of architectural decisions in light of quality attribute requirements and business goals. The ATAM helps stakeholders ask the right questions to discover potentially problematic architectural decisions. Discovered risks can then be made the focus of mitigation activities; for example, further design, further analysis, and prototyping. Surfaced tradeoffs can be identified and documented explicitly.

The CBAM helps software architects consider the ROI of any architectural decision and provides guidance on the economic tradeoffs involved. The CBAM takes the architectural decision analysis performed during the ATAM and helps make it part of a strategic roadmap for software design and evolution by associating priorities, costs, and benefits with each architectural decision.

Figure 9 provides a summary of the inputs, outputs, and participants of the combined ATAM/CBAM. For more details, see the work of Nord and colleagues [Nord 03].

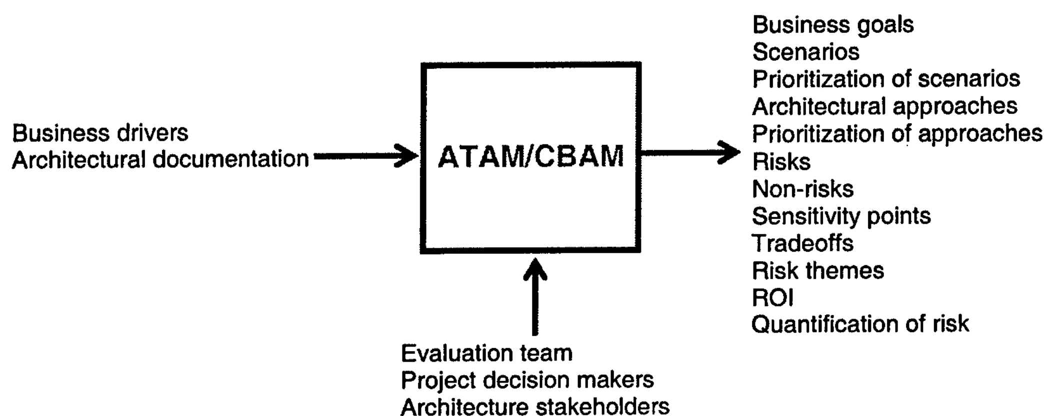


Figure 9: The Combined ATAM/CBAM Inputs, Outputs, and Participants

5.2 The ATAM/CBAM and XP

An evaluation can take place at different times during the development life cycle:

- Usually, at the end of an initial development iteration, there isn't much of a concrete architecture in place. But a review may uncover some unrealistic objectives, missing pieces, missed opportunities for reusing existing products, and so forth.
- It is possible to have a small evaluation at the end of each iteration during which a broad range of architectural qualities is examined.
- Damage-control evaluations may take place late in the iterations, when things have gone really wrong; for example, if construction does not complete or an unacceptable level of problems arises in the installed base.
- Finally, an evaluation may take place before delivery of the software, in particular to inventory reusable assets for an eventual new product or evolution cycle.

The ATAM can deliver more robust reviews to the XP team that are repeatable and produce consistent output. The CBAM prepares for the next iteration by considering costs and benefits, ultimately leading to a determination of ROI. The XP business case reveals the economic value of the product, so, clearly, CBAM-type reasoning can contribute to this evaluation.

An external evaluation team and stakeholders can be convened in a workshop setting (as was done for the QAW) when a more formal evaluation is desired at a major project milestone. Alternatively, the evaluation concepts can be applied incrementally at the end of an iteration. The developers act as ATAM/CBAM evaluation team members and also represent the stakeholders. Students participating in studio projects in the Masters of Software Engineering Program at Carnegie Mellon University have been using the ATAM in this way.

The inputs to the combined ATAM/CBAM include business drivers and architectural documentation. The ATAM/CBAM business drivers describe the system's most important functions (any relevant technical, managerial, economic, or political constraints; the business goals and context as they relate to the development project; and the major stakeholders) and the architectural drivers (that is, the major quality attribute goals that shape the architecture).

The ATAM/CBAM describes the driving architecture requirements and important architectural information: context diagram; module or layer view; component-and-connector view; deployment view; and architectural approaches, patterns, or tactics employed, including which quality attributes they address and how. The requirements and information are placed on the whiteboards for all developers to see and use.

The outputs from the combined ATAM/CBAM may feed into other XP iterations. For example, business goals are elicited or reviewed during the ATAM and could result in other stories to define further business value. The ROI computed by the CBAM can be used during

the Planning Game practice to provide information about the costs and benefits of choosing user stories at the beginning of each iteration.

5.3 Reflections

The ATAM/CBAM complements XP practices as shown in Table 3.

Table 3: ATAM/CBAM and XP Practices

XP Practices	Value Added Through the ATAM/CBAM
Planning Game	User stories are supplemented with quality attribute information in the form of six-part scenarios. Architectural strategy ROI and scenario prioritization and refinement give additional information to the customer and developers to help them choose user stories for each iteration.
On-Site Customer	The single on-site customer is supplemented with additional stakeholders during an evaluation workshop.
Refactoring	The ATAM contributes the artifacts (such as sensitivity points and tradeoffs) necessary for understanding the design before refactoring. The CBAM provides information about the cost of change so that differing refactoring strategies can be compared with respect to the value they bring to the customer.

The ATAM provides an architecture evaluation method. The ATAM adds value to XP by defining a step-by-step approach to evaluating software architecture that produces risk themes and shows the impact they have on achieving the business goals. The ATAM makes the evaluation of decisions to accommodate quality attribute requirements explicit. The method also contributes artifacts not necessarily found in a typical XP pairing. Sensitivity points and tradeoffs provide enhanced documentation for the architecture, concentrating on areas where risk is potentially highest. Scenarios provide feedback for existing and future requirements. All those things help improve the architecture.

The CBAM provides more details on the business consequences of architecture decisions implied by the architecture, allowing informed choices among architectural options to be made.

6 Evaluating Intermediate Designs with ARID

Software architectures often consist of complicated component designs. If these intermediate designs are inappropriate, the architecture can be undermined. Active Reviews for Intermediate Designs (ARID) is a lightweight evaluation approach that can be used to examine a design as it is developed and before it is released. For example, it may be used as the overall architecture is being designed to determine the design's viability.

6.1 ARID

ARID is a scenario-based, stakeholder-centric review of a *portion* of a software architecture, typically, a coherent software-invocable service. The ARID method blends Active Design Reviews [Parnas 01] with the ATAM, creating a technique for investigating partially completed designs. The review is focused on whether the design is sufficient to support the software developers who will use it. The ARID method helps to find the issues and problems that hinder the successful use of the design as it has been conceived.

Figure 10 provides a summary of the ARID method's inputs, outputs, and participants. For more details, see Clements' work [Clements 00].

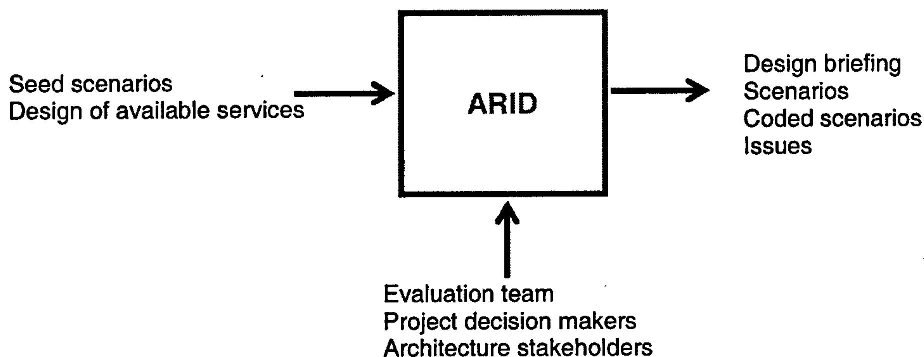


Figure 10: ARID Inputs, Outputs, and Participants

6.2 ARID and XP

ARID is used for reviewing elements of the architecture. It could occur as a form of architecture review during an iteration or following an ATAM for more detailed analysis. The

developers are the review's evaluators and stakeholders. The outputs from the ARID method feed into other XP activities such as defining or refining the design.

The ARID method can be used within XP's Small Releases practice, during which developers release iterative versions of the system to the customer often. A series of growing releases of a product demonstrates important functionality to the customer and allows time for early feedback to the developers. Small releases are easier to certify as well.

6.3 Reflections

The ARID method complements XP practices as shown in Table 4.

Table 4: ARID and XP Practices

XP Practices	Value Added Through ARID
Continuous Integration	Helps identify architectural mismatches at the interface level early on
Small Releases	Focuses on a portion of the architecture, typically a coherent software-invocable service
Test-Driven Development	Moves evaluation earlier in the software development life cycle. Rather than testing code, interface-detailed design is being "tested" via scenario walkthroughs.

The ARID method provides an architectural evaluation method but only for specific elements of the architecture that are investigated in greater detail.

7 Summary

In this report, we have summarized XP as an example of an agile process. We have also shown how SEI concepts and methods can be used in keeping with the agile philosophy of rapid and flexible development and the XP values of communication, simplicity, feedback, and courage (see Table 5).

Table 5: The Architecture-Centric Methods and XP Values

XP Values	Value Added Through Architecture-Centric Methods
Communication	Cost-effective methods facilitate interaction among a diverse group of stakeholders. Stakeholders' concerns regarding quality attribute requirements are captured and communicated to developers so they influence design. Communication between developers is also facilitated with a focus on the flow of information between interface users and developers.
Simplicity	Architecture design is course-grained, and only enough architecting is done to ensure that the design will produce a system that will meet its quality attribute requirements. Architecture evaluation has a notion of triage and uses techniques such as utility trees and prioritization to focus efforts.
Feedback	Architecture evaluation provides early feedback for understanding the technical tradeoffs, risks, and ROI of architectural decisions. Risks are related back to technical decisions and business goals.
Courage	Risks are exposed early in the life cycle, giving developers justification for investing resources to mitigate them. Architecture allows better planning so developers can better estimate the impact of requirements change. Change that is foreseen can be planned for and localized in the design.

In situations where requirements are changing rapidly and a lightweight approach is warranted, the concepts of quality attributes and architectural tactics can enhance the process of designing a system that will meet its requirements. Students participating in studio projects in the Masters of Software Engineering Program at Carnegie Mellon University have been using the Architecture-Centric Development Method—that uses concepts from the QAW, the ADD method, and the ATAM—in this way. This method was developed and taught by Anthony Lattanze, a former staff member of the SEI. When developing complex, large-scale applications, XP needs to be adapted to include more kinds of architectural information.

Table 6 provides a summary of how specific SEI architecture-centric methods can enhance the activities of XP by including quality attribute scenarios and architectural design in a goal-directed way to mitigate risks, thus enhancing the value of XP as a design process.

Table 6: The Architecture-Centric Methods and XP Activities

Method	Its Primary Task	XP Practices and Artifacts It Affects
QAW	To understand stakeholders' concerns for quality attribute requirements	Planning Game, On-Site Customer, Test-Driven Development, User Stories
ADD	To define a course-grained architecture	Planning Game, Metaphor, Simple Design, Refactoring, architectural spike
ATAM/ CBAM	To evaluate the architecture	Planning Game, On-Site Customer, Refactoring, release plan
ARID	To evaluate a portion of the architecture	Continuous Integration, Small Releases, Test-Driven Development

The benefit of including the SEI methods is to address quality attributes in an explicit, methodical, engineering-principled way. We believe that quality attribute requirements drive the software architecture and that architecture-centric activities (with an explicit focus on quality attributes) drive the software system's life cycle.

The architecture developed by the ADD method is influenced by the quality attribute requirements; it is not affected by changing functional requirements. The architecture helps localize the effects of design changes caused by changing functional requirements. In our experience, quality attribute requirements do not change as rapidly as functional requirements, thereby providing a more stable basis for the architectural design as well.

However, as with all such methods, XP—augmented with architecture-centric methods—is a garbage-in, garbage-out process. The willing participation of the appropriate stakeholders is crucial to the success of any such methods. Properly managed, the architecture-centric methods can be a low-cost addition to XP that will dramatically increase the quality of the systems and products developed.

References

URLs are valid as of the publication date of this document.

[Agile 01] Agile Alliance. *Manifesto for Agile Software Development.* http://www.agilemanifesto.org/ (2001).

[Bachmann 02] Bachmann, F.; Bass, L.; & Klein, M. *Illuminating the Fundamental Contributors to Software Architecture Quality* (CMU/SEI-2002-TR-025, ADA407778). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. http://www.sei.cmu.edu/publications/documents/02.reports /02tr025.html

[Barbacci 03] Barbacci, M. R.; Ellison, R.; Lattanze, A. J.; Stafford, J. A.; Weinstock, C. B.; & Wood, W. G. *Quality Attribute Workshops (QAWs), Third Edition* (CMU/SEI-2003-TR-016, ADA418428). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/publications/documents /03.reports/03tr016.html

[Bass 03] Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice, Second Edition.* Boston, MA: Addison-Wesley, 2003. http://www.sei.cmu.edu/publications/books/engineering /sw-arch-practice-second-edition.html

[Beck 04] Beck, K. *Extreme Programming Explained: Embrace Change, Second Edition.* Boston, MA: Addison-Wesley, 2004.

[Boehm 04] Boehm, B. & Turner, R. *Balancing Agility and Discipline: A Guide for the Perplexed.* Boston, MA: Addison-Wesley, 2004.

[Clements 00] Clements, P. *Active Reviews for Intermediate Designs* (CMU/SEI-2000-TN-009, ADA383775). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. http://www.sei.cmu.edu/publications/documents/00.reports /00tn009.html

[Clements 02] Clements, P.; Kazman, R.; & Klein, M. *Evaluating Software Architectures: Methods and Case Studies.* Boston, MA: Addison-Wesley, 2002. http://www.sei.cmu.edu/publications/books/engineering/eval-sw-arch.html

[Clements 03] Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Nord, R.; & Stafford, J. *Documenting Software Architectures: Views and Beyond.* Boston, MA: Addison-Wesley, 2003. http://www.sei.cmu.edu/publications/books/engineering/documenting-sw-arch.html

[Cockburn 01] Cockburn, Alistair. *Agile Software Development.* Boston, MA: Addison-Wesley, 2001.

[Fowler 99] Fowler, Martin. *Refactoring.* Boston, MA: Addison-Wesley, 1999.

[Herbsleb 03] Herbsleb, Jim & Tomayko, Jim. *How Useful Is the Metaphor Component of Agile Methods? A Preliminary Study* (CMU-CS-03-152). Pittsburgh, PA: School of Computer Science, Carnegie Mellon University, 2003.

[Highsmith 00] Highsmith, James, III. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems.* New York, NY: Dorset House Publishing, 2000.

[IEEE 98] Institute of Electrical and Electronics Engineers. *IEEE Standard for Functional Modeling Language* (IEEE Std 1320.1-1998). New York, NY: IEEE Computer Society, 1998 (ISBN 0738103403).

[Kazman 96] Kazman, R.; Abowd, G.; Bass, L.; & Clements, P. "Scenario-Based Analysis of Software Architecture." *IEEE Software 13,* 6 (Nov. 1996): 47-55.

[Kazman 02] Kazman, R.; Asundi, J.; & Klein, M. *Making Architecture Design Decisions: An Economic Approach* (CMU/SEI-2002-TR-035, ADA408740). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. http://www.sei.cmu.edu/publications/documents/02.reports/02tr035.html

[Kazman 03] Kazman, R.; Nord, R. L.; & Klein, M. *A Life-Cycle View of Architecture Analysis and Design Methods* (CMU/SEI-2003-TN-026, ADA421679). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/publications/documents/03.reports /03tn026.html

[Kazman 04] Kazman, R.; Kruchten, P.; Nord, R. L.; & Tomayko, J. E. *Integrating Software-Architecture-Centric Methods into the Rational Unified Process* (CMU/SEI-2004-TR-011). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/publications/documents/04.reports /04tr011.html

[Kruchten 04] Kruchten, P. *The Rational Unified Process: An Introduction, Third Edition.* Boston, MA: Addison-Wesley, 2004.

[McBreen 02] McBreen, Pete. *Questioning Extreme Programming.* Boston, MA: Addison-Wesley, 2002.

[Nord 03] Nord, R.; Barbacci, M.; Clements, P.; Kazman, R.; O'Brien, L.; & Tomayko, J. *Integrating the Architecture Tradeoff Analysis Method (ATAM) with the Cost Benefit Analysis Method (CBAM)* (CMU/SEI-2003-TN-038, ADA421615). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/publications/documents/03.reports /03tn038.html

[Nord 04] Nord, R. L.; Wood, W. G; & Clements, P. C. *Integrating the Quality Attribute Workshop (QAW) and the Attribute-Driven Design (ADD) Method* (CMU/SEI-2004-TN-017). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/publications/documents/04.reports /04tn017.html

[Palmer 02] Palmer, Stephen & Felsing, John. *A Practical Guide to Feature-Driven Development.* Upper Saddle River, NJ: Prentice-Hall, 2002.

[Parnas 01] Parnas, D. & Weiss, D. Ch. 17, "Active Design Reviews," 337-351. *Software Fundamentals: Collected Papers by David L. Parnas.* Hoffman, D. & Weiss, D., eds. Boston, MA: Addison-Wesley, 2001.

[Schwaber 02] Schwaber, Ken & Beedle, Mike. *Agile Software Development with Scrum.* Upper Saddle River, NJ: Prentice Hall, 2002.

[West 02] David West. "Metaphor, Architecture, and XP," 101-104. *Proceedings of the Third International Conference on Extreme Programming and Agile Processes in Software Engineering.* Alghero, Sardinia, Italy, May 26-29, 2002. Cagliari, Italy: University of Cagliari, 2002.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	September 2004	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Integrating Software-Architecture-Centric Methods into Extreme Programming (XP)	F19628-00-C-0003

6. AUTHOR(S)

Robert L. Nord, James E. Tomayko, Rob Wojcik

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2004-TN-036

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

This technical note fits the architecture-centric methods of the Carnegie Mellon® Software Engineering Institute (SEI) into the framework of Extreme Programming (XP). These methods include the Architecture Tradeoff Analysis Method®, the SEI Quality Attribute Workshop, the SEI Attribute-Driven Design method, the SEI Cost Benefit Analysis Method, and SEI Active Reviews for Intermediate Design. This report presents a summary of XP and examines the potential uses of the SEI's architecture-centric methods.

14. SUBJECT TERMS	15. NUMBER OF PAGES
architecture-centric methods, Architecture Tradeoff Analysis Method, ATAM, Active Reviews for Intermediate Design, ARID, Attribute-Driven Design method, ADD method, Cost Benefit Analysis Method, CBAM, agile software development, Extreme Programming, XP	44

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102