Extending the ATAM Architecture Evaluation to Product Line Architectures[1]

Femi G. Olumofin and Vojislav B. Mišić[2]

technical report TR 05/02
Department of Computer Science, University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2
June 2005

[1]draft paper submitted to Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA 5), Pittsburgh, Pennsylvania, USA, 6-9 November 2005
[2]University of Manitoba, Winnipeg, Manitoba, Canada

Extending the ATAM Architecture Evaluation to Product Line Architectures

Femi G. Olumofin and Vojislav B. Mišić

Abstract: Successful development of software product lines requires an architecture-centric approach with well established methodologies for both product line architecture (PLA) development and assessment. While several methodologies for PLA development have been proposed, the assessment of PLAs has mostly relied on methods developed for single product architectures. In this paper, we extend the popular ATAM (Architecture Tradeoff Analysis Method) method into a holistic approach that analyzes the quality attribute tradeoffs not only for the product line architecture, but for the individual product architectures as well. In addition, it prescribes a qualitative analytical treatment of variation points using scenarios. We present the main tenets of the extended method, and illustrate its use through a small case study.

Keywords: software architecture evaluation, software product line architecture, ATAM (Architecture Tradeoff Analysis Method)

Contents

1 Introduction

Software architecture practices are increasingly finding wider applicability and significance in most software development process [20, 14] as the key area for controlling software quality such as performance, reliability, modifiability, and availability. The architecture exerts even a greater influence on product quality in the case of software product line architecture where it assumes a dual form [7] – the core architecture or *product line architecture* (PLA) and the *individual product architectures* (PA). The priorities of different requirements, and sometimes the requirements themselves, differ between single product architectures and product line architectures. Consequently, traditional system development practices for single products cannot be directly applied to product line development [4]. As the quality goals and their priorities for the two kinds of architectures also differ, this observation holds for architecture assessment methods[1].

Existing architecture assessment methods focus on single product architectures and offer little support for the specific characteristics of product line architectures. The increasing importance of architecture-based development of software product lines requires the existence of appropriate architecture evaluation methods specifically addressing the quality of software product line architectures. Such methods may be developed from scratch or by extending one of the existing methods with suitable concepts and techniques. The latter approach is adopted in the development of the method which will be referred to as the Holistic Product Line Architecture Assessment (HoPLAA) method. This method is described in detail in the present paper.

Our starting point is the Architecture Tradeoff Analysis Method (ATAM) [13, 6], which is well developed and has a good record of successful applications in practice [2, 1]; furthermore, its trademark feature, the tradeoff analysis between different quality attributes, is not found in other methods. However, quality assessment of product line architectures requires that specific requirements of such architectures must be accounted for. To that end, HoPLAA extends the ATAM with the qualitative analytical treatment of variation points and the context-dependent generation, classification, and prioritization of quality attributes scenarios. In this manner, we are able to leverage the large body of existing research and industrial experiences on architecture evaluation of single product architectures, while setting the focus on the specific characteristics of software product line architectures.

The remainder of this paper is organized as follows. After a brief review of related work in Section 2, we introduce the HoPLAA method in Section 3, and present it in some detail in Section 4. Section 5 illustrates the use of the HoPLAA method through a small case study. Section 6 highlights the main benefits obtained through the use of HoPLAA and outlines the directions for further work.

2 Related work

The concept of software product lines can be traced to Parnas [18]: ”we consider a set of programs to be a program family if they have so much in common that it pays to study their common aspects before looking at the aspects that differentiate them.” Reuse of development effort and knowledge to achieve improved product quality, faster time-to-market, and reduced cost, are the key motivation for the product line approach to software development [2, 7]. The PLA captures the common requirements (or commonalities) of the products and represents varying requirements (or variability) using the so-called variation points. Individual products are subsequently built on the basis of both common and product-specific requirements. This can be accomplished either by extending the

[1] In the discussions that follow, we will use the terms *assessment*, *evaluation*, and *analysis* interchangeably.

PLA through direct addition of components, or by modifying the PLA to obtain the individual PA, and then building the product from that PA.

Products may be developed from a single, common architecture

In the first approach, the PLA (i.e., the core asset) is developed to explore commonalities and to create explicit space for variation points; in addition, a number of reusable components are developed as part of the product line assets [16]. Individual products are simply composed by adding the necessary components to the core architecture, without developing a distinct architecture for each of them.

Since there is only a single architecture to deal with, it might seem that any suitable single product architecture evaluation method would suffice. The range of available methods includes the Software Architecture Analysis Method (SAAM) [11, 12], the Architecture Tradeoff Analysis Method (ATAM) [2, 6, 13], and Active Reviews for Intermediate Designs (ARID) [6], all of which use scenarios as vehicles for both describing and analyzing software architectures. (Scenarios and other questioning techniques, such as questionnaires and checklists, are well suited to architectural evaluation because no running system is necessary [6].) Among those, ATAM is probably the most widely used; it has been the first one to introduce the concept of tradeoff analysis between quality attributes. Furthermore, it has been successfully applied to the evaluation of product line architectures [1]; however, the PLA in question used the single-architecture approach described above.

The problem with this approach lies in the fact that the core architecture allows only the quality attributes common to all products, while those attributes specific to a single product only, or some (but not all) of them, are not taken into account at all. At best, a separate evaluation of architectures of individual products (which were not defined in the first place) would be needed – a time consuming and not very useful exercise, at best.

Furthermore, the simplified, single-architecture approach cannot be applied to large, complex product lines in which the characteristics of individual products differ widely from one product to another.

Products should have architectures of their own

This problem is addressed by the alternative approach to the development of product line architectures, in which the concept of architecture assumes a dual role [7]. The product line architecture covers the common features, while individual product architectures pertain to the requirements specific to individual products; ‘the latter are produced from the former by exercising the built-in variation mechanisms to achieve instances [and] both should be evaluated’ [7]. In other words, while the separation of PLA from the PAs is convenient from the viewpoint of developers as it helps reduce complexity and increase focus, it also provides a convenient context for reasoning about the quality attributes of both core architecture and individual products architectures [15]. While some steps have been taken to make use of that context, the results are far from being satisfactory.

PuLSE-DSSA is the evaluation-focused component of the PuLSE methodology for the iterative creation and evaluation of reference architectures [3]. However, it has limited applicability as the evaluation process is bound to the PuLSE methodology. Further, there is no tradeoff analysis as it iteratively defines evaluation criteria per scenario. It however, claims to have the capability for regression evaluation [8, 3].

An iterative PLA evaluation framework has been developed by the VTT Technical Research Centre of Finland [9]. It uses the measurement instrument ‘defined by a taxonomy for quality attributes, which is organized with respect to three main elements: quality attribute priority, architecture view, and analysis method [9]. For

example, [performance, process view, ATAM] describes the evaluation of the process view of an architecture for performance, using the ATAM evaluation method [9]. The VTT TRC framework is fairly flexible, as it can leverage the advantages of several other methods, including architecture evaluation methods, and has a definite product line focus. However, it does rely on both evaluators and developers possessing good skills with different methods and, thus, poses a steep learning curve.

The product line engineering capability of an organization is addressed through the European ITEA projects and the resulting Family Evaluation Framework (FEF), which is a four-dimensional framework of business, architecture, process, and organization [17, 22]. The FEF is useful for benchmarking the product line engineering capability (or maturity)of an organization, analogous to the well known SEI Capability Maturity Model Integration framework [5]. Despite its treatment of architecture concerns, it is not architecture-centric and cannot not serve the purpose of a risk-mitigating architectural evaluation exercise. Besides, its architecture dimension only applies to architectures of existing systems, which essentially renders it useless for the evaluation of PLAs under development.

3 The HoPLAA method

Our study of existing architectural evaluation methods reveals some missing ingredients that disqualify them as the best approach to product line architecture evaluation. The architecture-centric and risk-mitigating methods are developed for single product architectures (e.g., ATAM, SAAM). On the other hand, the newer methods are dependent on a specific development methodology (e.g., PuLSE-DSSA) or not architecture-centric (e.g., FEF), or they simply present a technique for selecting which single product architecture evaluation method is the best to use for analyzing specific quality attributes (e.g., VTT TRC).

However, the development of product line architectures presents significant challenges that may never occur in single product development [10]. Apart from the need to exploit commonality across multiple products in a single product line architecture, there is also the issue of managing requirements of product instances as it relates to the product line scope. This confirms the Parnas' views expressed above, i.e., that in a product family, we consider the product commonalities before considering their differences or variability [18]. The problem of evaluating product line architectures must be approached by considering not only the quality attributes common to the family of systems, but also those specific to some members only, and their interrelationships. It should be noted that individual product architectures may require a different prioritization of the quality goals common to the product line architecture, and they may even be associated with quality goals which are not present in other members of the product line.

These notions provided the foundation for the development of the HoPLAA architecture evaluation method specifically focused on software product line architectures.

The HoPLAA method addresses the requirements for the evaluation of software product line architectures in an integrated, *holistic approach* with two separate but inter-dependent analysis steps for the PLA and the PA. The main goal of the holistic approach is to simplify the analysis of quality attributes and their interactions, as the architectural decisions are made right from the product line architecture creation to the individual product architectures derivations. This method is executed in two stages; the first stage focuses on the PLA evaluation, while the second stage targets individual PA evaluation. This specialization is schematically shown in Fig. 1, which illustrates the HoPLAA and its inputs and outputs.

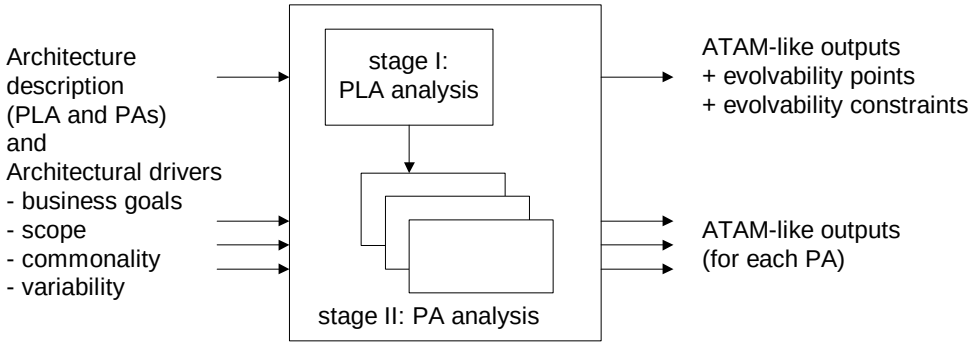
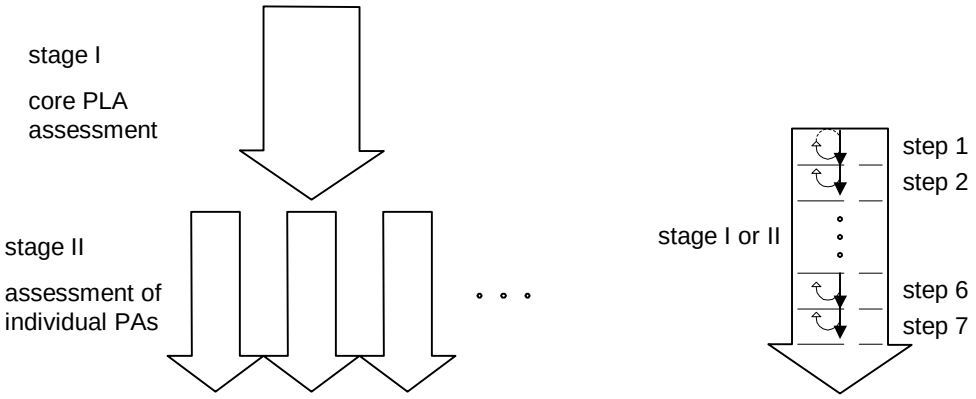


Figure 1: HoPLAA: Inputs and Outputs.

Since the proposed method is an extension of the ATAM, it should come as no surprise that the outputs are similar to those obtained through an ATAM evaluation. However, the PLA evaluation outputs generated in Stage I, such as architecture approaches, evolvability constraints, scenarios, and the like, have to be structured so as to facilitate the subsequent PA evaluation activities in Stage II. In particular, the PLA evaluation outputs put more emphasis on the evolvability points and evolvability constraints, as opposed to the output obtained through ATAM analysis of a single product architecture, as will be explained in the following.



(a) Stage breakdown. (b) Steps within a stage.

Figure 2: Pertaining to the stage breakdown of the HoPLAA method.

Furthermore, each of the stages is actually composed of individual steps, seven in stage I and seven in stage II, with each step designed to meet the specific need of the respective target architecture. Stage I is used for evaluation of the common architecture; in fact, it may be used for single product architecture evaluation with little customization. In this respect, PLA evaluation in stage I may be viewed as a customized version of the ATAM method. (Note that the ATAM approach, which focuses on the evaluation of single product architectures, consists of nine steps [6].) Stage II of the HoPLAA method consists of PA evaluations applied to each individual product architecture separately, as shown in Fig. 2. These evaluations may proceed in sequence, in parallel, or a mixture of the two, depending on the size of the available architecture evaluation team.

We will now present the main features of the HoPLAA method. In particular, we highlight the concepts

borrowed from traditional, single product architecture evaluation methods and the ways in which those concepts are adapted to suit the needs of product line architecture evaluation.

Extension of Quality Tradeoff Analysis

The notion of architectural tradeoff analysis between two or more quality attributes is among the most prominent features of the ATAM approach [13]. Since design decisions at the architectural level have far-reaching consequences, the quality attributes (which are addressed by those decisions) cannot be treated in isolation; almost every decision will affect more than one quality attribute.

Consequently, each design decision involves a tradeoff between quality attributes; the essence of the ATAM approach is to identify those tradeoffs and make them visible to the architects so as to minimize the risks incurred in the architecture definition process. Architectural design decisions that impact quality attributes interactions are classified into sensitivity points and tradeoff points. A sensitivity point applies to a decision about specific aspects of the architecture that may affect—either benefit or impair—at least one quality attribute; a tradeoff point is a sensitivity point between two or more quality attributes which interact in opposing ways [6].

Although initially developed for the analysis of single product architectures, the tradeoff analysis is perfectly applicable to product line evaluation. When building the core PLA, design decisions have to fulfill a dual goal. They have to address the quality attributes common to all the variants. At the same time, they have to accommodate quality attributes which may be specific to some of the variants only. Moreover, not all of these variants and, by extension, not all of the associated quality attributes, are known at the architecture evaluation time. The best we can do, then, is to satisfy the common quality goals without making it too difficult to satisfy the product-specific quality goals (some of which are not known) at a later time.

In the HoPLAA method, tradeoff analysis of quality attributes for the core PLA is performed in stage I, in a fashion similar to the ATAM evaluation but with extra outputs. Further, we extend the quality tradeoff analysis to individual PAs in stage II, in order to determine the validity of the quality goals addressed in the core PLA and also to verify how individual product-specific quality attributes goals are supported. We employ a technique based on the relationship between variation points and sensitivity points for handling the resulting complexity. It is worth noting that this technique may well find more general applicability in the case of n-dimensional and hierarchical product lines [21]. The distinguishing characteristic of such product lines is that each PA allows variation points, as does its parent PLA, and thus could be instantiated for several other products. By following this technique, the architects could simplify the analysis of tradeoffs in adjacent planes of product line architectures.

Context-dependent Scenario Treatment

Most single product architecture evaluation methods employ scenarios as the vehicle both for describing the architecture and eliciting the quality goals for analysis purposes. Once quality attributes and scenarios that exercise them are identified, it is common to eliminate redundant ones and prioritize the remaining ones through voting. While this procedure may be feasible for single product architectures, it becomes cumbersome even for large architectures in which the initial number of attributes is high, and the number of scenarios may easily be a hundred or more. For product line architectures, the scope is even wider and the numbers are higher.

Hence, the simple scenario generation and voting approach is simply too inefficient and some other way of reducing the complexity of the quality attribute lists must be found. The most obvious one is to use the already

existing distinction between the core PLA and individual PAs as the basis for the decomposition. In this manner, we may be able to simplify the management and analysis of the scenarios and focus the evaluation on specific aspects of the product line.

In stage I of our method, we generate quality attributes scenarios without constraining the evaluation team to the commonalities defined by the scope of the product line. In other words, we allow the generation of both common and product-specific scenarios. While the former are analyzed and prioritized in stage I, the latter are recorded but not analyzed until stage II. The reason for this deferment lies in the fact that some of the quality attribute-related scenarios may not be fully known during the generation phase. Instead of trying to get their details no matter what the cost may be, it is more appropriate (and certainly more productive) to allow the architecture evaluation team to freely generate and/or brainstorm all possible scenarios, common and product-specific alike.

Furthermore, the elicitation of some product-dependent scenarios in stage I saves time during the particular product evaluation later on in stage II, and gives indication of possible product-specific quality goals so that architectural decisions made in designing PLA will not rule them out. Of course, additional product-specific scenarios are generated, brainstormed, and analyzed during the assessment of a particular PA.

Therefore, part of our goal in providing a wholesome approach to the analysis of product line architectures is this careful consideration given to quality attributes scenarios generation, classification and prioritization. Each stage is focused only on the analysis of those scenarios in the context of the generalized product line architecture or instance architecture under consideration at any one time. More of this will be described in the section elaborating the details of HoPLAA.

Qualitative Analytical Treatments for Variation Points

Variation points are placeholders for future architectural components in a PLA. They are realized as concrete variants in individual product architectures. The reader should recall that sensitivity points are those architectural decisions that affect one or more quality goals [13]. For example, the encryption of sensitive message exchange between two components may improve the security quality of a software-intensive system. The architectural decision to introduce cryptographic components between the two communicating components is a sensitivity point to realizing security as far as message exchange between the two components is concerned.

Architectural decisions made in the analysis of the PLA (i.e., in stage I), and found to be the sensitivity points to one or more quality attributes, continue to remain valid for individual product architectures. A possible exception would be the case in which the creation of a PA involves the addition of component variants to those parts of the architecture which interact with the sensitivity points. In the example given above, consider adding a third component to periodically receive exception messages from both components. If such notification messages to this third component are not similarly encrypted, the security of the system may be jeopardized.

An area of the product line architecture which is a sensitivity point, and which contains at least one variation point, will be referred to as an *evolvability point*. We provide special treatment for variation points that could alter quality using evolvability points. In particular, we accompany each evolvability point in the PLA with guideline to constrain or guide subsequent PA design decisions and evaluation. The intention is to guide against PA design decisions that could invalidate quality goals already allowed in the product line architecture.

4 The steps of HoPLAA

The common practice for specifying an architecture evaluation method is to outline the steps of activities to perform to arrive at an evaluation result for the architecture. In almost all cases, these steps of activities may be repeated whenever a new architecture is to be evaluated irrespective of the level of similarity (or dissimilarity) between the architectures being evaluated, e.g., in the case of a product line. Such architectural evaluation method, although successful for single-product architectures, fail to recognize the duality of the architectures in a product line context. While the product line architecture features commonalities in functional and quality requirements with explicit room for variation points, the product architecture may feature additional functional and quality requirements but generally with no provision for variation points. A one-size-fits-all approach typical of most single product evaluation method is inadequate in a product line context.

The prescribed steps of the two stages of HoPLAA are presented in this section. Note that the steps may be customized to suit the particular environment in the organization (or part thereof) that develops the software product line.

4.1 Stage I: PLA evaluation

Stage I consists of the tradeoff-oriented analysis of the common architecture (PLA).

I.1 Present the HoPLAA Stage I: Present an overview of the HoPLAA and the activities of the two stages. Provide more details on the activities of Stage I.

I.2 Present the product line architectural drivers: This is a description of the product line in terms of the motivating business needs, the product line scope definition, and the commonality and variability of the conceived products in the line especially in terms of quality goals.

I.3 Present the product line architecture: The architects describe the product line architecture (i.e., the core architecture or PLA).

I.4 Identify architectural approaches: The evaluation team identifies the architectural approaches used in the architecture. The list is documented but not analyzed. In a product line context, there is need for consistency in the use of architectural approaches throughout the design of the PLA and the individual PAs. When the set of architectural approaches used is from a finite known set, the analysis of any architecture in the product line is simplified.

I.5 Generate, classify, brainstorm, and prioritize quality attribute scenarios: There are two categories of quality goals anticipated based on the architectural drivers – those common or mandatory to all products in the line, and those peculiar to some of the products only. The former must be verified in the current stage against the PLA, while the latter will be elicited at this stage but will not receive any special treatment until the particular product architecture evaluation in Stage II. The aim is to have the PLA address all quality attribute concern common to every product in the line. Besides, we ensure architectural decisions made on the PLA do not rule out the achievement of other product-specific quality goals – hence the reason for eliciting product-specific scenarios in this stage.

In addition to the other quality goals, the quality attribute of variability (also called evolvability or modifiability) should always be analyzed in a PLA. The key point here is the need for large-scale reuse of architecture, which is essential to the product line concept itself and is best realized when the variability is fully supported by the architecture.

We represent quality attribute scenarios with the utility tree similar to the ATAM utility tree. All possible attribute concerns and associated quality attribute scenarios are shown in the utility tree irrespective of their generality (which may be mandatory, alternative or optional). In this paper, we refer to generic scenarios as concrete scenarios addressing the quality-attribute-specific goals of the PLA. (Note that these are not the same as general scenarios that 'provides a framework for generating a large number of generic, system-independent, quality-attribute-specific scenarios' [2].) In a way, generic scenarios could be considered as system-dependent (PLA) instances of general scenarios.

We rank each scenario using three indexes: Generality, Significance and Cost, each of which is assigned a value in the enumeration [10, 20, 30] for Low (L), Medium (M), and High (H), respectively. Generality may be mandatory, alternative or optional, with values 30, 20 and 10, respectively. Significance denotes how important the quality attribute scenario is to the business driver. Cost is the level of effort or difficulty involved in enhancing the architecture to provide the right responses to the scenario. Once assigned to individual scenarios, the values of indexes are added up so as to prioritize the list of scenarios (generic scenarios and product specific scenarios).

Using the high-priority scenarios as input, a larger (possibly new) set of scenarios is elicited from the same or different group of stakeholders. Generic scenarios are extracted from the combined list of brainstormed scenarios and prioritize by voting. All other scenarios, both the low-priority generic scenarios and the product-specific ones, are incorporated into the utility tree of the product line for use in Stage II. It is anticipated that the most important attribute concerns shared among all products in the line will characterize the scenarios on top of the list.

I.6 Analyze architectural approaches/generic scenarios: Analyze high-priority generic scenarios from step 5 to obtain a set of architectural risks, non-risks, sensitivity points, tradeoff points [6], and evolvability points. Evolvability points are those areas in the product line architecture that are either sensitivity point or tradeoff point and that contain at least one variation point. In order words, evolvability points are those areas in the architecture where (a) their composition is associated with at least a variation points, and (b) the associated design decisions affect the realization of one or more quality attribute goals. Associate guidelines to evolvability points to constrain subsequent changes that attempt to move the architecture away from the quality goals already allowed, or to guide future analysis of product architectures that realize associated variation points in the PLA.

I.7 Present results: A report is prepared for HoPLAA Stage I, containing the list of architectural approaches, utility tree, generic scenarios, product-specific scenarios identified, areas of risks in the PLA, architectural decisions that are non-risks, sensitivity points, tradeoffs, evolvability points, evolvability guidelines, and risk theme as it affects the product line mission.

4.2 Stage II: individual PA evaluation

Stage II of the HoPLAA method focuses on evaluation of individual product architectures, and it consists of the following steps.

II.1 Present the HoPLAA Stage II: Present an overview of method and give details of Stage II. This is a very short step, as the essence of the methods should already be known to the evaluation team.

II.2 Present the architectural drivers: Give a brief overview of the PLA, the set of driving requirements for the particular product architecture, a description of the variable features of this product in terms of functional and quality goals.

II.3 Present the product architecture: Place emphasis only on areas of the architecture that have been recently enhanced through the realization of variation points.

II.4 Identify architectural approaches: The architect identifies new or different architectural approaches used in the architecture. This is documented but not analyzed. When a newer architectural approach is used in the realization of a variation point, the architect must also give the rationale for doing so. In addition, the team identifies and documents the specific variation points that has been realized as variants.

II.5 Generate, brainstorm and prioritize quality attribute scenarios: Reproduce quality attribute scenario that are specific to this product from the utility tree generated in Stage I (Step 5). Double check to confirm agreement with the quality drivers identified in Step 2. In addition, elicit possibly new set of product-specific scenarios from the same or different group of stakeholders and prioritize the entire list of scenarios by significance (e.g., by voting).

All previously unidentified scenarios may be reflected in the utility tree for the product line. Alternatively, a separate utility tree may be created for the product architecture which should contain scenarios relating to mandatory and product specific quality goals. The mandatory quality goals may only focus on those quality attributes related to evolvability points whose variation points have been realized in this product architecture. The list of variants is used to obtain the affected evolvability points from the results of Stage I.

II.6 Analyze architectural approaches: The two classes of scenarios relating to the PA under consideration must be analyzed in this step. The architect must show how generic quality attribute scenarios are not precluded by the product architecture. Generic quality attributes should still continue to be satisfied when design decisions do not violate the evolvability guidelines. If the converse is true, one or more architectural risks have been introduced in realizing the product architecture that may preclude one or more generic quality goals.

In addition, the prioritized product-specific scenarios in step 5 should be analyzed to obtain a set of architectural risks, non-risks, sensitivity points, and tradeoff points. Essentially, the architect must demonstrate how generic scenarios are not precluded in the product architecture design and also how the architecture realizes quality goals that are specific to the product being analyzed.

II.7 Present results: An evaluation report, similar to the one described in Step 7 of HoPLAA Stage I but without evolvability points and evolvability guidelines (as architecture-based variations are no more supported), is prepared.

Certain preliminary and post-evaluation activities as in the ATAM industrial case study experiences are recommended to set the stage for, and conclude the activities of this method.

5 HoPLAA analysis example

We illustrate the use of HoPLAA with the Arcade Game Maker (AGM) Product Line, a pedagogical product line example jointly developed by Clemson University and the SEI. The product line is made up of three games (Brickle, Pong, and Bowling) to be produced for three different hardware/software platforms, for a total of nine products. In the following, we will briefly describe the AGM product line as well as some slight changes in quality requirements that have been made to suit this example; complete documentation, including the ATAM evaluation report, can be found in [19].

The [functional] commonalities of the game products are: single-player, graphical presentation of games, animation-driven, moveable and stationary game pieces or sprites, and some set of common rules based on physical laws of physics. Mandatory quality requirements for all products are:

1. Performance

a. The action of the game must be sufficiently fast to seem continuous to the user.

b. The graphics refresh rate must allow for smooth animations without blurring of graphics.

2. Modifiability

a. A product architecture could be instantiated from the PLA in two weeks' time or less.

b. Using the assets of any of the initial products, a single programmer should be able to deliver next two increments of products in less than a week.

3. Maintainability

a. The PLA and/or PA should not inhibit products updates to newer versions of their environment or platforms as they are released from time to time. Such updates should be within three days by a single experienced systems programmer.

The [functional] variability includes rules of the game, type and number of pieces, behaviour of pieces and the physical environment of platform for the games which could be freeware arcade games, commercial personal computer games and commercial wireless games.

Commercial wireless versions of the games are required to support auto-save of game state and scores whenever the user shuts down the game from a pause mode, or powers off the wireless device (such as a cell phone) while the game is running.

Brickles variability: The domain of game programs has the usability quality as a crucial requirement. One of the business drivers for the AGM product line is to provide users with engaging and interesting games; in fact, a game where users are unable to meet usability problems will not have a second chance. We enhance the Brickle product for usability as follows:

- Sound support: plays feedback sound which can be set on/off from within the game.
- User-defined customization: replacement of sprites (game pieces) and sound feedback files at run-time.

A sound feedback should be provided whenever the puck is in motion, hit the blocks, hit the walls or hit by the paddle within 0.75 seconds.

This modification introduces variability in quality and functional requirements to the Brickle product architecture.

5.1 Stage I: PLA Evaluation

Next, we present the result of the HoPLAA evaluation, starting with stage I – PLA evaluation.

Architecture Description

The PLA description has an additional variation point for a sound device for the Brickle game. The static deployment diagram of the PLA is shown in Fig. 3; other views are available in the AGM documentation [19].

The *SoundDriver* is the software interface to the audio device shown as a variation point for interface with audio player in the Brickles product architecture.

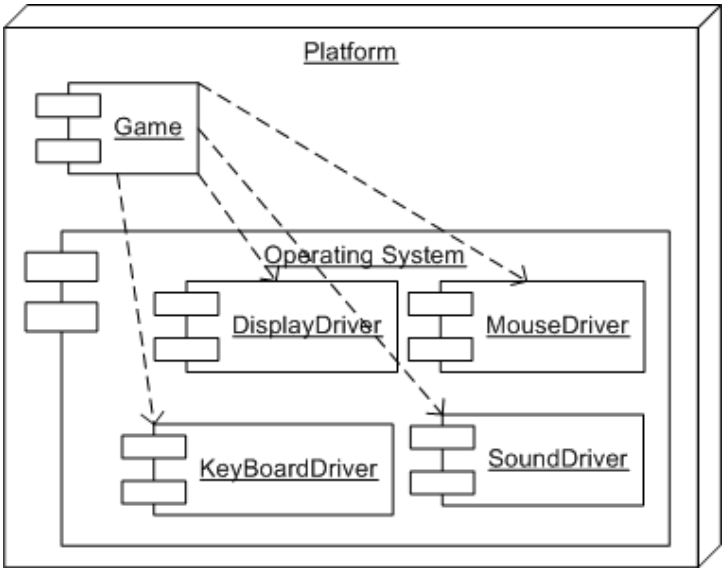


Figure 3: Enhanced AGM Product Line Deployment Diagram (adapted from [19]).

Architectural Approaches

Some of the architectural approaches used are:

- A *Model-View-Controller* (MVC) approach;
- A modified MVC approach;
- Object oriented specialization and generalization as means of variability; and
- Parameterized class configuration for variability.

Scenarios

Some of the scenarios elicited are:

1. Performance: Game response time

i. A paddle hits the puck in response to a user input while the game graphics display reflects the effect of the collision in less than 0.75 seconds. (H, H, H)

ii. Several blocks (stationary sprites) are configured for the Brickles game and the start-up time for the game is less than 5 seconds on a PC running Pentium II processor. (M, H, M)

iii. The user changes the game speed to the minimum and maximum possible while the systems animations are not jerky and the paddle could still be controlled to intercept the puck. (H, H, H)

iv. A commercial wireless game user in a game session mistakenly hit the power-off button while the game auto-saves its state and scores in 0.5 minutes before power went down. (M, H, M)

2. Modifiability

i. The AGM Company decides to implement database persistence for commercial wireless game state and scores while a single programmer from the Arcade Team completed an updated PLA in three days. (H, H, H)

ii. The AGM Company decides to field pinball game that requires more than one view of the game state while the Arcade Team created the PA to use the full MVC in less than one week. (H, M, M)

Risks

The following risks were found in the architecture:

i. As the number of stationary sprites grows, the time for collision detection would increase. The architecture approaches collision detection by checking each stationary sprites whether they are being hit by a movable spite.

ii. The cycle time for the animation ticks (managed using the *SpeedControl* Interface) may become too fast for the user to intercept the puck at maximum speed. On the other hand, the ticks that drive animation may have become too slow in a minimum setting to make for smooth motion.

iii. The usability of the game is questionable because the graphics model is rather rudimentary. If the game interface is not satisfactory, it may not have a second chance with users, thereby presenting a negative impression of the company itself.

iv. The current architecture manages persistence by having game state and scores saved to flat files in its host platform. But the ultimate goal is an architecture that implements database persistence for scores and game state savings. There is the possibility that a single programmer will be unable to deliver next product increments that uses database for persistence in less than a week. Further, an experienced programmer may require more than three days to migrate from one database platform to another (e.g., migrating from PC-based database client to mobile databases on wireless devices).

Non-Risks

i. The game and score representation is generalized. This could improve maintainability. For example, every game formats its score as common string.

ii. The game pieces are specialized as movable sprites and stationary spites. This architecture decision to make this separation helps improve the speed of collision detection.

Sensitivity Point

i. The performance of a game is sensitive to the number of stationary sprites.

ii. The latency between a user action (e.g., hit the puck) and the update of the game graphics display is sensitive to resource consumption (e.g., CPU, memory) for collision detection and handling. For example,

when a brickle puck collides with a pile of blocks, a sound feedback is played in addition to the collision handling itself.

Tradeoff Point

i. Two of the architectural approaches used in the architecture description, namely the MVC and the Modified MVC, introduce architectural tradeoffs. The MVC is extremely demanding in terms of memory, especially for low memory wireless devices, because of duplication of data in the model and views. It is also a performance hog because every change in the model is forwarded to all views. However, it enhances modifiability since separate views may be easily connected to the model. On the other hand, the Modified MVC allows good performance in view updates, but makes it very difficult to create additional views of data (i.e., it opposes modifiability).

ii. There is a tradeoff between performance and development cost. The architecture currently employs a simple technique for collision detection instead of subdividing the game board into grids to improve the speed of search for collision.

Evolvability Points

i. The performance of a game is sensitive to the number of sprites in the game board. And number of sprites is a variation point for game products. Hence, number of sprites is also an evolvability point.

ii. The sound device is a variation point for implementing sound feedback variants. The fact that this variation point has to do with an area of the architecture that is also sensitivity point for graphics display response time, makes it an evolvability point

Evolvability Constraints or Guidelines

i. The number of sprites and their sizes must be controlled based on the environment or platform. A large number of sprites may impair response time for game graphics redisplay. Similarly, pieces that uses large-sized images may deplete memory resources. Smaller number of sprites is recommended for commercial wireless games.

ii. The resource requirements (CPU time, memory and sound device) for audio feedback should be controlled. Especially, it must be completed such that total CPU time for graphics redisplay does not exceed the 0.75 seconds limit.

5.2 Stage II: PA Evaluation

A separate architecture description was created for the Brickle product to satisfy functional and quality goals already described. For space constraint, only a layered representation of the Brickle PA is shown in Fig. 4.

Scenarios

i. The Brickle game is running and the sound option is set. The user paddle hits the puck and the system responds with sound feedback in 0.75 seconds.

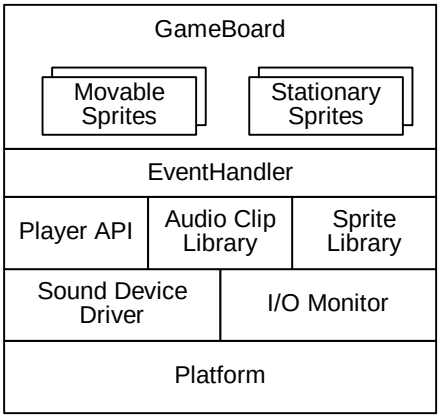


Figure 4: A layered view of the Brickles architecture.

ii. The user selects a menu option that turns off the sound, but the sound feedback stops in 1 second.

iii. The game is running and the puck hit the block but the sound did not play until after 1 second later.

iv. The user interrupts the Brickle while running to replace the blocks in the piles with custom-blocks from the Sprite Library. The system replaces all blocks in the pile with the custom block and resumes game correctly.

v. The user imports a sound clip from the Audio clip library to replace sound feedback for collision of the puck with the blocks. The game subsequently takes up to 5 seconds to playback the sound clip, placing the sound feedback system out of sync with the game action.

Some scenarios generated in Stage I are included in the list of the Brickles PA scenarios.

Scenario Analysis

The Brickles PA realizes two variation points from the PLA, i.e., the number of sprites in the game board, and the sound feedback.

Design decisions needed to implement the variation points affect one or more quality, and hence are treated as evolvability points. The evolvability constraints in Stage I are used to simplify this analysis.

Risks

i. A user may specify large number of blocks per block pile and equally import large-sized images for sprites graphics. This could slow down the animation and action of the game. A low memory wireless device would easily run out of memory for that reason.

ii. The flexibility of allowing users to provide custom sound clips may result in use of expensive audio clips (e.g., long-playback sound clip) that could place the game action out of sync with sound feedback. The performance goal of graphics redisplay in 0.75 could be affected as well.

Sensitivity Point

Usability of the game as a result of sound feedback is sensitive to the size of audio clips allowed to the user for importation.

Tradeoffs

i. Enhancing games usability by allowing users to customize sprites, and sounds by importation of the appropriate files could improve games usability but adverse on performance and memory usage. Some users may use heavy graphics for sprites or may import large sound files.

The risk theme for the PA is to constrain user to specified image sizes and constrain playback time of audio clips. Although this requirement is aimed at improving usability, it could equally inhibit it. The architecture does not currently feature a means for achieving this control. The architecture should also show synchronization techniques for the sound feedback vis-à-vis the game action.

6 Summary and conclusion

Each of the two stages of the HoPLAA was structured to fit the evaluation demands of the PLA and the PA evaluation, respectively. Architecture evaluation is expensive and limited time is spent for this exercise in reality. The steps of Stage I are only seven, two less than the number of steps in a comparable ATAM evaluation. We combined scenario generation and brainstorming in one step and the analysis of scenarios in one step, since the product line analysis performed during the scoping and requirements identification suffices for the purpose. We expected that important scenarios would have sufficed, without the need for additional two steps of scenario generations/brainstorming and analysis.

Further, as the product line architecture modification is an ongoing process, other requirements that suffice in the lifetime of the product line may be added or incorporated to this core asset or product instances. ATAM, being well suited for single-product architecture, anticipates product developments that solidifies the architecture design. Therefore, care should be taken to ensure the set of most important quality attribute concern are considered.

The separate architecture created for the PA is evaluated within Stage II of the HoPLAA. It reuses some of the scenarios generated from Stage I, and employs the evolvability points and evolvability constraints as guidelines to direct the analysis to those areas of the PA that may have changed due to the realization of variation points. For example, the provision of sound feedback is found to have introduced performance risk to Brickles. Design decisions that introduce tradeoffs in quality concerns are given separate treatment in both analyzes. All of these issues would remain uncovered in the traditional ATAM analysis of the PLA.

In this manner, separate treatment for the PAs separates concerns and focuses analysis to important product-specific qualities. Completing this evaluation in the same context as the PLA evaluation would complicate the entire process. This justifies the separation along the PLA-PA demarcation line.

In terms of efficiency, stage I of the HoPLAA analysis should take less time than the equivalent ATAM analysis of the core PLA, since some steps have been merged, and the analysis of some scenarios is deferred until stage II. Regarding stage II, steps II.1 through II.3 can be much shorter than their stage I counterparts, since the

attendees are already familiar with the procedure. Furthermore, step II.1 can be 'shared' between individual PA analyses, i.e., one session would suffice for product lines with several PAs. In addition, step II.5 will reuse some scenarios already discovered in stage I, giving additional savings.

Overall, we can say that the HoPLAA analysis of a PLA will take more time than the equivalent ATAM analysis of the core PLA architecture alone, but less time than would be needed to perform the ATAM analysis to each individual PA separately. In either case, HoPLAA out-performs ATAM in terms of comprehensiveness and effectiveness of the analysis, as explained above.

Of course, the HoPLAA method presented here is still undergoing development; it requires further validation and refinements through more focused case studies. Nonetheless, we believe that the results are encouraging, and that the concept of extending the existing evaluation methods for single product architectures to cover software product line architectures offers distinct advantages.

Acknowledgements

The work presented in this paper was partially supported by the NSERC Discovery Grant.

References

[1] M. Barbacci, P. Clements, A. Lattanze, L. Northrop, and W. Wood. Using the Architecture Tradeoff Analysis Method (ATAM) to evaluate the software architecture for a product line of avionics systems: A case study. CMU SEI Technical Note CMU/SEI-2003-TN-012, Software Engineering Institue, Pittsburgh, PA, 2003.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. The SEI Series in Software Engineering. Addison-Wesley, Reading, MA, 2nd edition, 2002.

[3] J. Bayer, Anastasopoulos, C. Gacek, and O. Flege. A process for product line architecture creation and evaluation: PuLSE-DSSA version 2.0. Technical Report No.038.00/E, Fraunhofer IESE, Kaiserslautern, Germany, June 2000.

[4] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. PuLSE: a methodology to develop software product lines. In *SSR '99: Proc. 1999 Symp. on Software Reusability*, pages 122–131, 1999.

[5] M. B. Chrissis, M. Konrad, and S. Shrum. *CMMI: Guidelines for Process Integration and Product Improvement*. The SEI Series in Software Engineering. Addison-Wesley, Reading, MA, 2003.

[6] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures – Methods and Case Studies*. The SEI Series in Software Engineering. Addison-Wesley, Reading, MA, 2002.

[7] P. Clements and L. Northrop. *Software Product Lines*. The SEI Series in Software Engineering. Addison-Wesley, Reading, MA, 2002.

[8] J.-M. DeBaud, O. Flege, and P. Knauber. PuLSE-DSSA: a method for the development of software reference architectures. In *ISAW '98: Proc. Third Int. Workshop on Software Architecture*, pages 25–28, Orlando, FL, 1998.

[9] L. Dobrica and E. Niemelä. A strategy for analyzing product line software architectures. Technical Report VTT-PUBS-427, VTT Electronics, Oulu, Finland, 2000.

[10] D. Garlan. Software architecture: a roadmap. In *ICSE'00: Proc. 22nd Int. Conf. Software Engineering*, pages 91–101, Limerick, Ireland, 2000.

[11] R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. *IEEE Software*, 13(6):47–55, 1996.

[12] R. Kazman, L. Bass, M. Webb, and G. Abowd. SAAM: a method for analyzing the properties of software architectures. In *ICSE '94: Proc. 16th Int. Conf. Software Engineering*, pages 81–90, Sorrento, Italy, 1994.

[13] R. Kazman, M. Klein, and P. Clements. ATAM: Method for architecture evaluation. CMU SEI Technical Note CMU/SEI-2000-TR-004, ADA382629, Software Engineering Institue, Pittsburgh, PA, 2000.

[14] R. Kazman, P. Kruchten, R. L. Nord, and J. E. Tomayko. Integrating software-architecture-centric methods into the Rational Unified Process. CMU SEI Technical Note CMU/SEI-2004-TR-011, Software Engineering Institue, Pittsburgh, PA, 2004.

[15] R. C. Malveau. *Software Architect Bootcamp: A Programmer's Field Manual*. Prentice Hall, Upper Saddle River, NJ, 2000.

[16] E. Niemelä and T. Ihme. Product line software engineering of embedded systems. In *Proc. 2001 Symp. on Software Reusability*, pages 118–125, 2001.

[17] E. Niemelä, M. Matinlassi, and A. Taulavuori. Practical evaluation of software product family architectures. In *Proc. Third Software Product Line Conference SPLC 2004*, pages 130–145, Boston, MA, Aug/Sep 2004.

[18] D. L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proc. 3rd Int. Conf. Software engineering*, pages 264–277, 1978.

[19] The Product Line Systems Program of the Software Engineering Institute, Luminary Software and Clemson University Dept of Computer Science. *Arcade Game Maker Product Line*, 2004. available at `http://www.cs.clemson.edu/~johnmc/`.

[20] Rational Corp., Cupertino, CA. *Rational Unified Process*, 2001.

[21] J. M. Thompson and M. P. E. Heimdahl. Extending the product family approach to support n-dimensional and hierarchical product lines. In *Proc. Fifth IEEE Int. Symp. Requirements Engineering (RE'01)*, page 56, Washington, DC, 2001.

[22] F. van der Linden, J. Bosch, E. Kamsties, K. Känsälä, and J. H. Obbink. Software product family evaluation. In *Proc. Third Software Product Line Conference SPLC 2004*, pages 110–129, Boston, MA, Aug/Sep 2004.