Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software xxx (2006) xxx–xxx
The Journal of Systems and Software
www.elsevier.com/locate/jss

# A general model of software architecture design derived from five industrial approaches

Christine Hofmeister [a], Philippe Kruchten [b,*], Robert L. Nord [c], Henk Obbink [d], Alexander Ran [e], Pierre America [d]

[a] *Lehigh University, Bethlehem, PA, USA*
[b] *University of British Columbia, 2332 Main Mall, Vancouver, BC, Canada V6T 1Z4*
[c] *Software Engineering Institute, Pittsburgh, PA, USA*
[d] *Philips Research Labs, Eindhoven, The Netherlands*
[e] *Nokia Research Center, Cambridge, MA, USA*

Received 1 January 2006; received in revised form 8 May 2006; accepted 12 May 2006

**Abstract**

We compare five industrial software architecture design methods and we extract from their commonalities a general software architecture design approach. Using this general approach, we compare across the five methods the artifacts and activities they use or recommend, and we pinpoint similarities and differences. Once we get beyond the great variance in terminology and description, we find that the five approaches have a lot in common and match more or less the "ideal" pattern we introduced. From the ideal pattern we derive an evaluation grid that can be used for further method comparisons.


*Keywords:* Software architecture; Software architecture design; Software architecture analysis; Architectural method

## 1. Introduction

Over the last 15 years a number of organizations and individual researchers have developed and documented techniques, processes, guidelines, and best practices for software architecture design (Bass et al., 2003; Bosch, 2000; Clements et al., 2002a; Clements and Northrop, 2002; Dikel et al., 2001; Garland and Anthony, 2002; Gomaa, 2000). Some of these were cast and published as architecture design methods or systems of concepts, processes and techniques for architecture design (Hofmeister et al., 1999; Kruchten, 2003; Obbink et al., 2000; Ran, 2000).

Since many of the design methods were developed independently, their descriptions use different vocabulary and appear quite different from each other. Some of the differences are essential. Architecture design methods that were developed in different domains naturally exhibit domain characteristics and emphasize different goals. For example architectural design of information systems emphasizes data modeling, and architecture design of telecommunication software emphasizes continuous operation, live upgrade, and interoperability. Other essential differences may include methods designed for large organizations vs. methods suitable for a team of a dozen software developers, methods with explicit support for product families vs. methods for one of a kind systems, etc.

On the other hand, all software architecture design methods must have much in common as they deal with the same basic problem: maintaining intellectual control over the design of software systems that: require involvement of

* Corresponding author. Tel.: +1 604 827 5654.
*E-mail addresses:* crh@eecs.lehigh.edu (C. Hofmeister), pbk@ece.ubc.ca (P. Kruchten), rn@sei.cmu.edu (R.L. Nord), henk.obbink@philips.com (H. Obbink), alexander.ran@nokia.com (A. Ran), pierre.america@philips.com (P. America).

0164-1212/$ - see front matter © 2006 Published by Elsevier Inc.
doi:10.1016/j.jss.2006.05.024

and negotiation among multiple stakeholders; are often developed by large, distributed teams over extended periods of time; must address multiple possibly conflicting goals and concerns; and must be maintained for a long period of time. It is thus of significant interest to understand the commonalities that exist between different methods and to develop a general model of architecture design. Such a model would help us better understand the strengths and weaknesses of different existing methods as well as provide a framework for developing new methods better suited to specific application domains.

With this goal in mind, we selected five different methods: Attribute-Driven Design (ADD) Method (Bass et al., 2003), developed at the SEI; Siemens' 4 Views (S4V) method (Hofmeister et al., 1999), developed at Siemens Corporate Research; the Rational Unified Process® 4 + 1 views (RUP 4 + 1) (Kruchten, 1995, 2003) developed and commercialized by Rational Software, now IBM; Business Architecture Process and Organization (BAPO) developed primarily at Philips Research (America et al., 2003; Obbink et al., 2000), and Architectural Separation of Concerns (ASC) (Ran, 2000) developed at Nokia Research. We also assembled a team of people who have made significant contributions to developing and documenting at least one of the methods. Through extensive discussions focused on how typical architecture design tasks are accomplished by different methods, we have arrived at a joint understanding of a general software architecture design model that underlies the five methods. In this paper we document our understanding of what seems to be fundamental about architecture design.[1]

This paper is organized as follows. We introduce the five contributing methods in Section 2. Then in Section 3 we present a general model of architecture design. Section 4 describes the five contributing methods using terms and concepts of the general model, and discusses the commonalities and differences between the contributing methods. Section 5 describes how other software architecture methods can be compared against the general model using a grid, and applies the grid to another published method. Section 6 discusses related work, Section 7 proposes future work, and Section 8 concludes the paper.

## 2. Five industrial software architecture design methods

### 2.1. Attribute-Driven Design

The Attribute-Driven Design (ADD) method (Bass et al., 2003), developed at the SEI, is an approach to defining software architectures by basing the design process on the architecture's quality attribute requirements.

In ADD, architectural design 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 (see Fig. 1). The architecture designed using the ADD method represents the high-level design choices documented as containers for functionality and interactions among the containers using 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. The architecture is critical for achieving desired quality and business goals, and providing the framework for achieving functionality.

Architects use the following steps when designing an architecture using the ADD method:

1. *Choose the module to decompose*. The module to start with is usually the whole system. All required inputs for this module should be available (constraints, functional requirements, quality requirements)
2. *Refine the modules according to these steps*:
   a. Choose the architectural drivers from the set of concrete quality scenarios and functional requirements. This step determines what is important for this decomposition.
   b. Choose an architectural pattern that satisfies the drivers. Create (or select) the pattern based on the tactics that can be used to achieve the drivers. Identify child modules required to implement the tactics.
   c. Instantiate modules and allocate functionality from use cases, and represent the results using multiple views.
   d. Define interfaces of the child modules. The decomposition provides modules and constraints on the types of module interactions. Document this information in the interface document of each module.
   e. Verify and refine the use cases and quality scenarios and make them constraints for the child modules. This step verifies that nothing important was forgotten and prepares the child modules for further decomposition or implementation.
3. *Repeat the steps above for every module that needs further decomposition*.

Fig. 2 shows how the ADD method fits together with the other SEI architectural design activities. The Quality Attribute Workshop (QAW) (Barbacci et al., 2003) helps in understanding the problem by eliciting quality attribute requirements in the form of quality attribute scenarios. The Views and Beyond (VaB) approach (Clements et al., 2002a) documents a software architecture using a number of views based on stakeholders' needs. The Architecture Tradeoff Analysis Method® (ATAM®) (Clements et al., 2002b) provides detailed guidance on analyzing the design and getting early feedback on risks. The figure does not show how these methods are used in the context of an organization's own architecture process (see (Kazman et al., 2004; Nord et al., 2004) for examples of relating ADD to the Rational Unified Process® and Agile methods).

[1] A shorter version of this work was presented at WICSA (Hofmeister et al., 2005a).

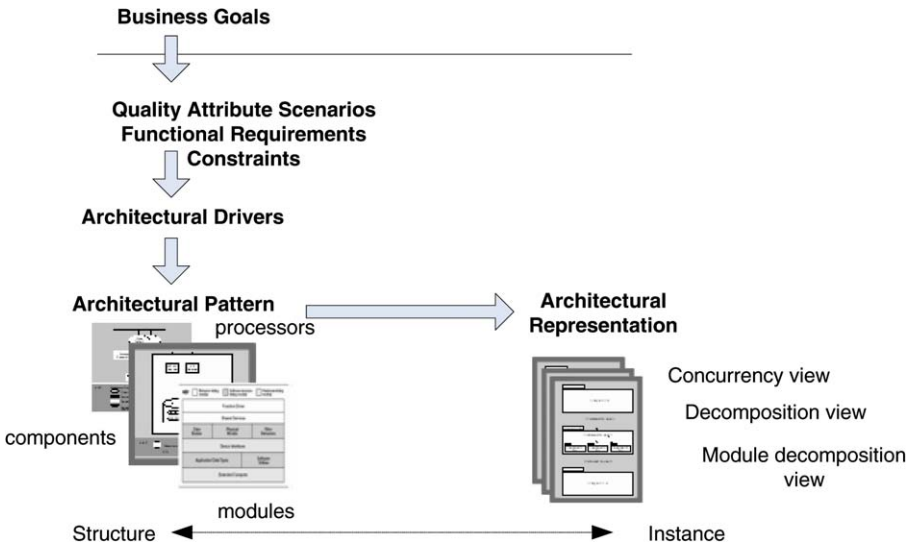


Fig. 1. Recursively designing the architecture using ADD.

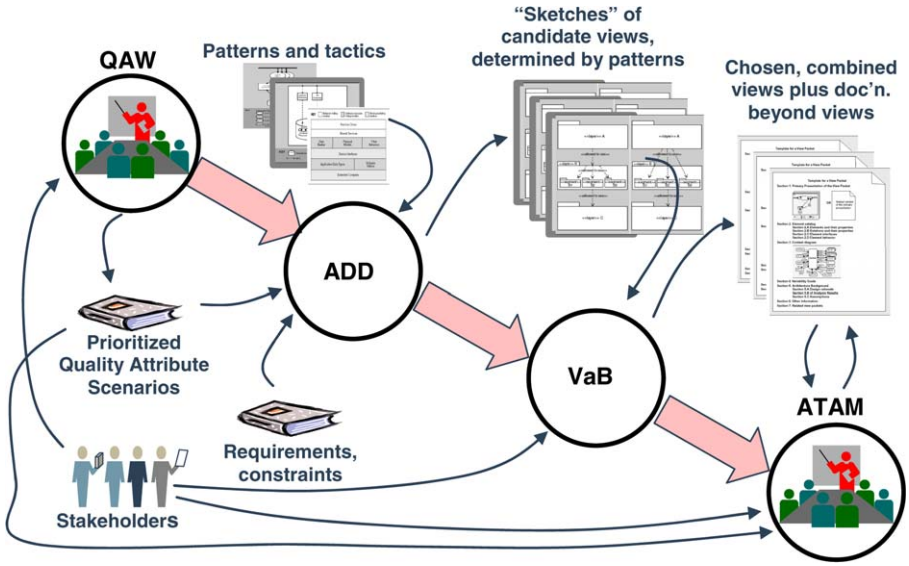


Fig. 2. ADD in relation with other SEI architectural design activities.

### 2.2. Siemens' 4 views

The Siemens Four-Views (S4V) method (Hofmeister et al., 1999; Soni et al., 1995), developed at Siemens Corporate Research, is based on best architecture practices for industrial systems. The four views (conceptual, execution, module and code architecture view), separate different engineering concerns, thus reducing the complexity of the architecture design task.

In the conceptual view, the product's functionality is mapped to a set of decomposable, interconnected components and connectors. Components are independently executing peers, as are connectors. The primary engineering concerns in this view are to address how the system fulfills the requirements. The functional requirements are a central concern, including both the current requirements and anticipated future enhancements. Global properties such as performance and dependability are addressed here as well as in the execution view. The system's relationship to a product family, the use of COTS, and the use of domain-specific hardware and/or software are all addressed in the conceptual view as well as in the module view.

For the module view, modules are organized into two orthogonal structures: decomposition and layers. The decomposition structure captures how the system is logically decomposed into subsystems and modules. A module can be assigned to a layer, which then constrains its dependencies on other modules. The primary concerns of this view are to minimize dependencies between modules, maximize reuse of modules, and support testing. Another key concern is to minimize the impact of future changes in

COTS software, the software platform, domain-specific hardware and software, and standards.

The execution architecture view describes the system's structure in terms of its runtime platform elements (e.g., OS tasks, processes, threads). The task for this view is to assign the system's functionality to these platform elements, determine how the resulting runtime instances communicate, and how physical resources are allocated to them. Other considerations are the location, migration, and replication of these runtime instances. Runtime properties of the system, such as performance, safety, and replication must be addressed here.

The last view, the code architecture view, is concerned with the organization of the software artifacts. Source components implement elements in the module view, and deployment components instantiate runtime entities in the execution view. The engineering concerns of this view are to make support product versions and releases, minimize effort for product upgrades, minimize build time, and support integration and testing.

These views are developed in the context of a recurring Global Analysis activity (Hofmeister et al., 2005b). For Global Analysis, the architect identifies and analyzes factors, explores the key architectural issues or challenges, then develops design strategies for solving these issues.

The factors that influence the architecture are organized into three categories: organizational, technological, and product factors. The purpose of the categories is to help the architect identify all influencing factors, including not just requirements but also desired system qualities, organizational constraints, existing technology, etc. These factors are analyzed in order to determine which factors conflict, what are their relative priorities, how flexible and stable is each factor, what is the impact of a change in the factor, and what are strategies for reducing that impact.

From these factors the key architectural issues or challenges are identified; typically they arise from a set of factors that, taken together, will be difficult to fulfill. Issues can arise when factors conflict, or when certain factors have little flexibility, a high degree of changeability, or a global impact on the system. The architect need not necessarily identify and analyze all factors before identifying the key issues. Sometimes it is more fruitful to let an issue give rise to a number of factors, or to alternate between these two approaches.

The next step is to propose design strategies to solve the issues, and to apply the design strategies to one or more of the views. Strategies often involve applying software engineering principles, heuristics, and architectural patterns or styles to solve the problem. When a strategy is applied to a particular part of a view, we call it a 'design decision'.

During design, the design decisions must be evaluated, particularly for conflicts and unexpected interactions. This evaluation is ongoing. Thus Global Analysis activities are interleaved with view design activities, and the design activities of each view are also interleaved (see Fig. 3).

In contrast to the ongoing evaluation that is part of the design process, periodic architecture evaluation is done in order to answer a specific question, such as cost prediction, risk assessment, or some specific comparison or tradeoff. This typically involves other stakeholders in addition to the architect. Global Analysis provides inputs to this kind of architecture evaluation, for example: business drivers, quality attributes, architectural approaches, risks, tradeoffs, and architectural approaches.

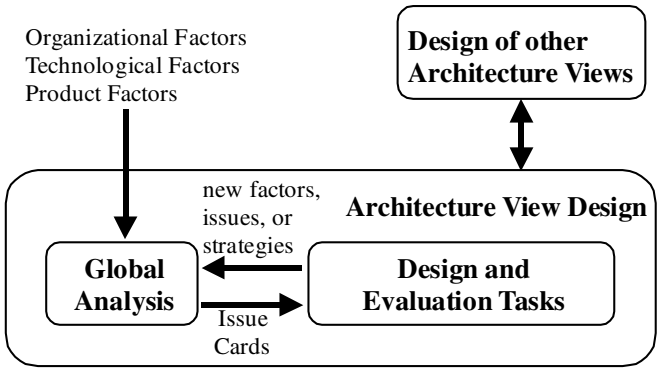


Fig. 3. Workflow between Global Analysis and Architecture View Design.

### 2.3. RUP's 4 + 1 Views

The Rational Unified Process® (RUP®) is a software development process developed and commercialized by Rational Software, now IBM (Kruchten, 2003). For RUP "software architecture encompasses the set of significant decisions about the organization of a software system:

- selection of the structural elements and their interfaces by which a system is composed,
- behavior as specified in collaborations among those elements,
- composition of these structural and behavioral elements into larger subsystem,
- architectural style that guides this organization.

Software architecture also involves: usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and tradeoffs, and aesthetic concerns." RUP defines an architectural design method, using the concept of 4 + 1 views (RUP 4 + 1) (Kruchten, 1995): four views to describe the design: logical view, process view, implementation view and deployment view, and using a use-case view (+1) to relate the design to the context and goals (see Fig. 4).

In RUP, architectural design is spread over several iterations in an *elaboration phase*, iteratively populating the 4 views, driven by architecturally significant use cases, non-functional requirements in the supplementary specification, and risks (see Fig. 5). Each iteration results in an *executable architectural prototype*, which is used to validate the architectural design.

Architectural design activities in RUP start with the following artifacts, which may not be finalized and still evolve

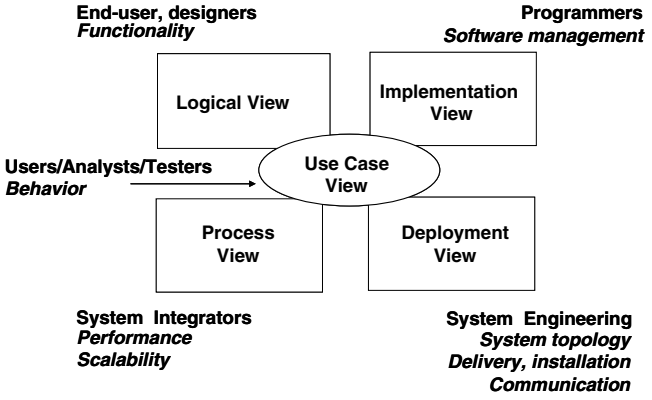


Fig. 4. RUP's 4 + 1 views.

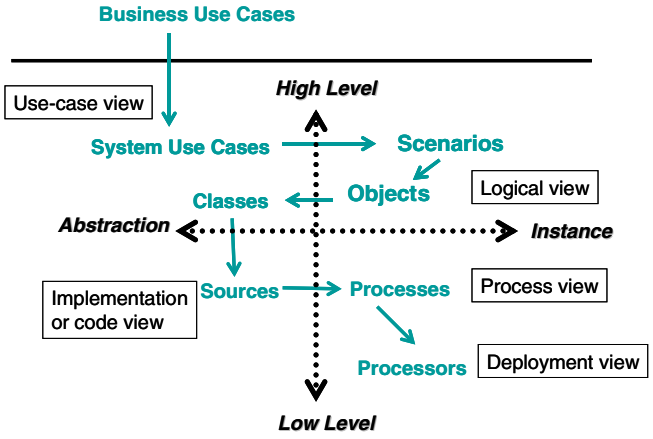


Fig. 5. Iteratively populating the five views.

during the architectural design: a vision document, a use-case model (functional requirements) and supplementary specification (non-functional requirements, quality attributes). The three main groups of activities are:

(1) Define a Candidate Architecture
This usually starts with a use-case analysis, focusing on the use cases that are deemed architecturally significant, and with any reference architecture the organization may reuse.
This activities leads to a first candidate architecture that can be prototyped and used to further reason with the architectural design, integrating more elements later on.
(2) Perform Architectural Synthesis
Build an architectural proof-of-concept, and assess its viability, relative to functionality and to non-functional requirements.
(3) Refine the Architecture
Identify design elements (classes, processes, etc.) and integrate them in the architectural prototype
Identify design *mechanisms* (e.g., architectural patterns and services), particular those that deal with concurrency, persistency, distribution.
Review the architecture.

And the RUP also provides activities related to the documentation of the architecture in each of the 5 views shown in Fig. 4.

More recently (Rozanski and Woods, 2005) have added *perspectives* to RUP's views and viewpoints, to more effectively capture certain quality properties, in a way similar to architectural aspects or SEI's tactics (Bass et al., 2003).

### 2.4. Business architecture process and organization

The BAPO/CAFCR approach (America et al., 2003; Muller, 2005; Obbink et al., 2000; van der Linden et al., 2004), developed primarily by Philips Research, aims at developing an architecture (the A in BAPO) for software-intensive systems that fits optimally in the context of business (B), (development) process (P), and organization (O). Ideally the entities at the right side of the BAPO acronym should be driven by and support the entities at their left (see Fig. 6), even though this does not always work in practice, so that the architecture must fit into a given context of business, process, and organization.

For the architecture, the five CAFCR views are defined: Customer, Application, Functional, Conceptual, and Realization. These views bridge the gap between customer needs, wishes, and objectives on the one hand and technological realization on the other hand.

In BAPO/CAFCR, the architect iteratively: (1) fills in information in one of the CAFCR views, possibly in the form of one of the suggested artifacts; (2) analyzes a particular quality attribute across the views to establish a link between the views and with the surrounding business, processes and organization (see Fig. 7). To counteract the tendency towards unfounded abstraction, specific details are often explored by story telling. The reasoning links valuable insights across the different views to each other. The architecture is complete when there is sufficient information to realize the system and the quality attribute analysis shows no discrepancies.

At a larger scale, the processes described in the BAPO/CAFCR approach deal with the development of a product family in the context of other processes in a business.

### 2.5. Architectural separation of concerns

Architectural separation of concerns (ASC) or ARES System of Concepts (Ran, 2000), developed primarily by Nokia, is a conceptual framework based on separation of concerns to manage complexity of architecture design.

Fig. 8 illustrates the ASC model of architecture-centered development. Development goals affect architectural decisions. Architectural decisions are represented by architecture descriptions. Architecture descriptions are used to verify architectural decisions. Architecture description and implementation must be consistent. A validation process exists to determine consistency of architecture and implementation. Information regarding achievement of

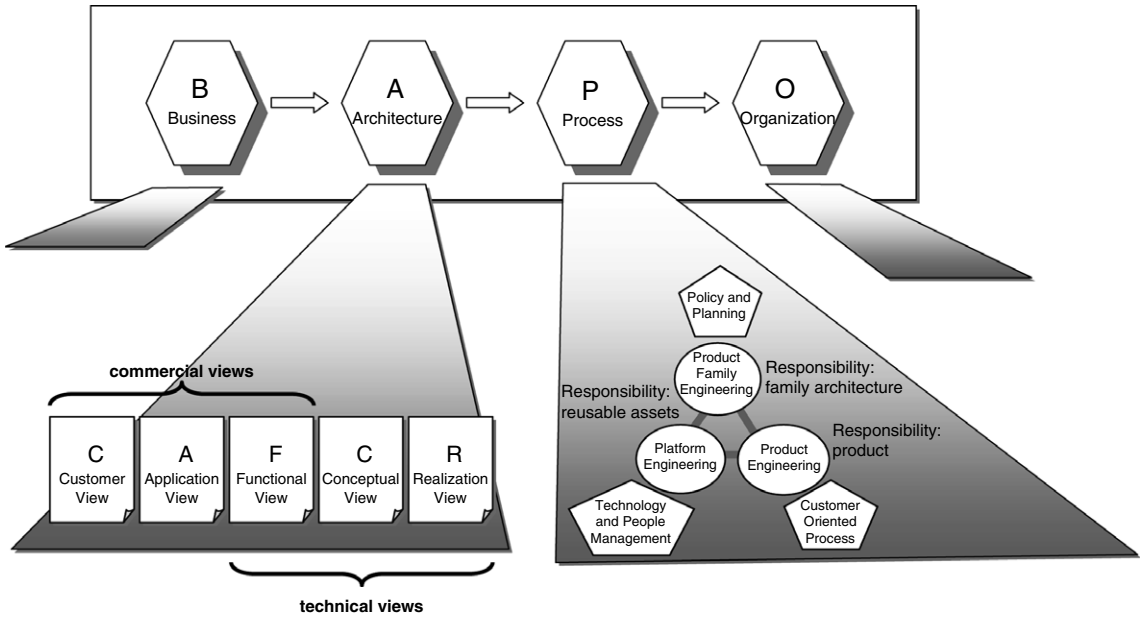


Fig. 6. The BAPO/CAFCR framework.

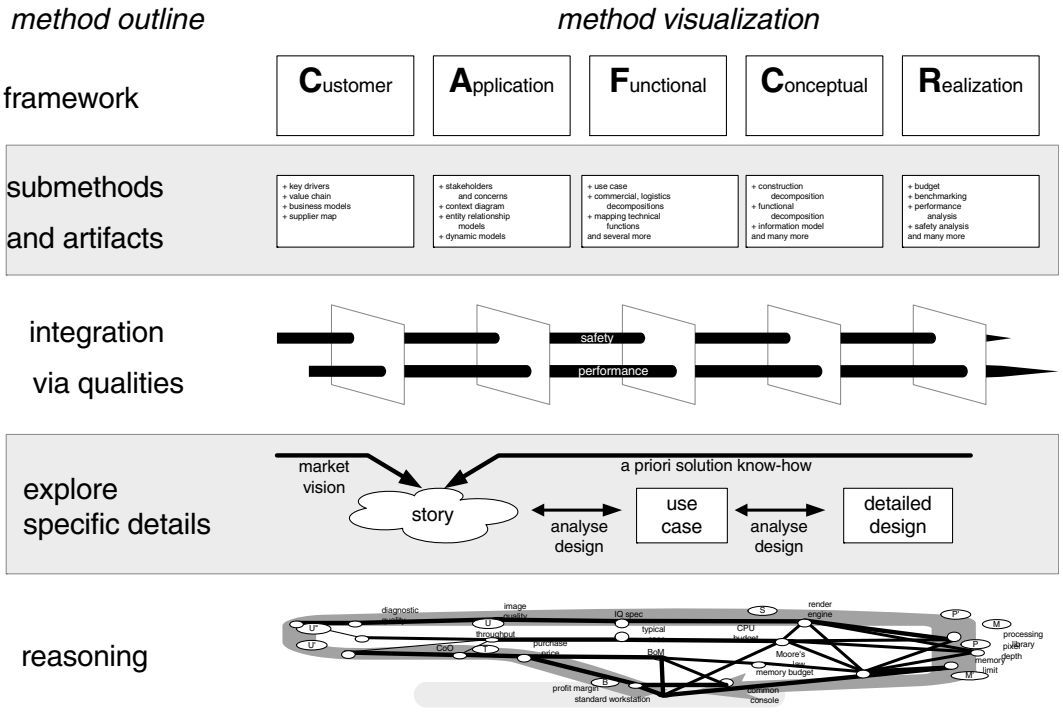


Fig. 7. Elements of the CAFCR method.

development goals is obtained once the implementation is available.

Stakeholder goals need to be analyzed and if determined to have an effect on software architecture, the goals need to be refined into architecturally significant requirements (ASR) that state how to measure the achievement of the goal and indicate how the goal constrains software architecture.

Software goes through a transformation cycle that consists of separate segments. During the design or development segment, the software is source code. During the build segment the software is object files and library

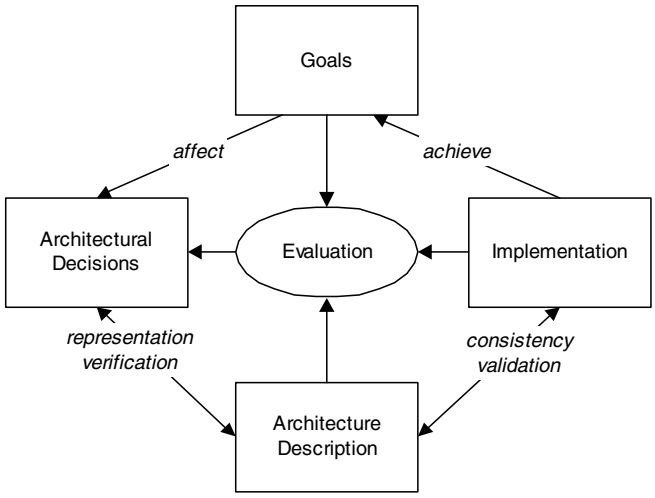


Fig. 8. A model of architecture-centered software development.

archives. During the upgrade segment the software consists of executable files. During the start-up segment the software is system state and groups of executable entities with their dependency structure. During the operation segment software is threads and objects.

Architecturally significant requirements must be grouped so that requirements in different groups may be satisfied independently, while requirements within each group may interact and even conflict. This can be achieved if we group the requirements by the segment of software transformation cycle. Architectural concerns that are related to different segments of the software transformation cycle can be addressed independently, while concerns that are related to the same segment cannot be addressed independently and must be a subject of trade-off decisions.

Fig. 9 illustrates the separation of architectural concerns based on different segments of software transformation cycle.

In addition to design of architectural structures for each segment, ASC pays special attention to design of texture. Certain design decisions that are only visible within relatively fine-grained components are nevertheless extremely expensive to revise. Consequently such decisions are architecturally significant even though they are concerned with fine-grained elements. This happens when the implementation of the decision cannot be localized, but must be replicated creating recurring uniform microstructure or texture.

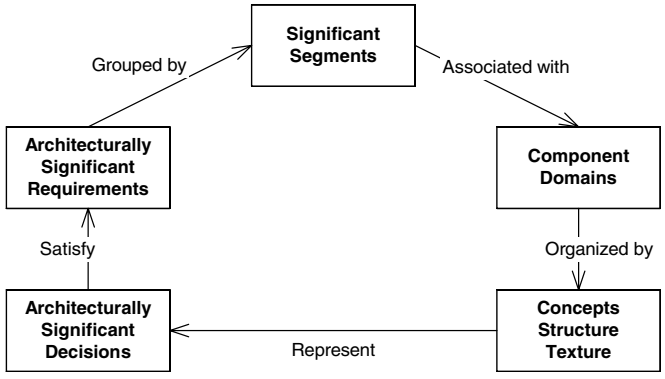


Fig. 9. Segmentation of concerns.

In ASC, the architect analyses design inputs, such as preferred technology platforms, road maps, functional and quality requirements for the product family and the product, and using a palette of techniques, produces and prioritizes ASR (architecturally significant requirements), groups ASR by segments of the software transformation cycle that they address. Implementation (write-time) design addresses the ASR concerned with the write-time segment. Design decisions make implementation technology choices, partition functional requirements between different architectural scopes of product portfolio, product family, or single product, establish portability layers for multiplatform products, allocate classes of functional requirements to different subsystems, and develop description of the API facilitating work division and outsourcing. Performance (run-time) design deals with run-time ASR addressing concurrency and protection, develops performance models and makes decisions regarding task and process partitions, scheduling policies, resource sharing and allocation. Finally, delivery/installation/upgrade design decisions address the ASR of the corresponding segments. Typical decisions address partitions into separately loadable/executable units, installation support, configuration data, upgrade/downgrade policies and mechanisms, management of shared components, external dependencies and compatibility requirements.

## 3. A general model for software architecture design

The general model for software architecture design we developed first classifies the kinds of activities performed during design. Architectural analysis articulates architecturally significant requirements (ASRs) based on the architectural concerns and context. Architectural synthesis results in candidate architectural solutions that address these requirements. Architectural evaluation ensures that the architectural decisions used are the right ones (see Fig. 10).

Because of the complexity of the design task, these activities are not executed sequentially. Instead they are used repeatedly, at multiple levels of granularity, in no predictable sequence, until the architecture is complete and validated. Thus the second part of the general model is a characterization of its workflow.

The key requirement of our model was that it be general enough to fit our five architecture design methods, and provide a useful framework for comparing them. One strong influence on the activities in our model was Gero's Function–Behavior–Structure framework for engineering design (Gero, 1990; Gero and Kannengiesser, 2004), which Kruchten applies to software design in Kruchten (2005).

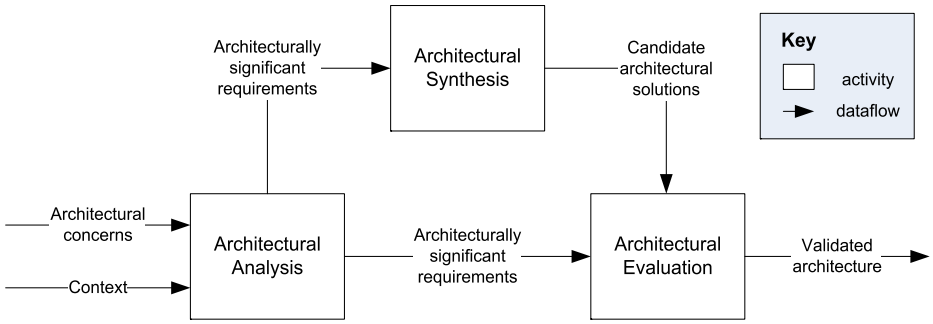


Fig. 10. Architectural design activities.

### 3.1. Architectural design activities and artifacts

First we describe the main activities of the model, and their related artifacts.

- *Architectural concerns: The* IEEE 1471 standard defines architectural concerns as "those interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders. Concerns include system considerations such as performance, reliability, security, distribution, and evolvability" (IEEE, 2000). Most architectural concerns are expressed as requirements on the system, but they can also include mandated design decisions (e.g., use of existing standards). Regulatory requirements may also introduce architectural concerns.
- *Context:* According to IEEE 1471, "a system's ... environment, or context, determines the setting *and* circumstances of developmental, operational, political, and other influences upon that system" (IEEE, 2000). This includes things like business goals (e.g., buy vs. build), characteristics of the organization (e.g., skills of developers, development tools available), and the state of technology. Note that sometimes the only distinction between a concern and a context is whether it is specifically desired for this system (a concern) or is instead a general characteristic or goal of the organization or a stakeholder (context). For example, a business goal of the architecture is a concern, whereas a business goal of the enterprise is context.
- *Architecturally significant requirements:* An ASR is "a requirement upon a software system which influences its architecture" (Obbink et al., 2002). Not all of the system's requirements will be relevant to the architecture. Conversely, not all ASRs will have originally been expressed as requirements: they may arise from other architectural concerns or from the system context.
- *Architectural analysis:* Architectural analysis serves to define the problems the architecture must solve. This activity examines, filters, and/or reformulates architectural concerns and context in order to come up with a set of ASRs.
- *Candidate architectural solutions:* Candidate architectural solutions may present alternative solutions, and/or may be partial solutions (i.e., fragments of an architecture). They reflect design decisions about the structure of software. The architectural solutions include information about the design rationale, that is, commentary on why decisions where made, what decisions were considered and rejected, and traceability of decisions to requirements.
- *Architectural synthesis:* Architectural synthesis is the core of architecture design. This activity proposes architecture solutions to a set of ASRs, thus it moves from the problem to the solution space.
- *Validated architecture:* The validated architecture consists of those candidate architectural solutions that are consistent with the ASRs. These solutions must also be mutually consistent. Only one of a set of alternative solutions can be present in the validated architecture. The validated architecture, like the candidate architectural solutions, includes information about the design rationale.
- *Architectural evaluation:* Architectural evaluation ensures that the architectural design decisions made are the right ones. The candidate architectural solutions are measured against the ASRs. Although repeated evaluations of different architectural solutions are expected, the eventual result of architectural evaluation is the validated architecture. Intermediate results would be the validation or invalidation of candidate architectural solutions.

In addition to the above-described artifacts used in the design activities, there are some less explicit inputs that are critical to the design process:

- Design knowledge comes from the architect, from organizational memory, or from the architecture community. It can take the form of styles, patterns, frameworks, reference architectures, ADLs, product-line technologies, etc.
- Analysis knowledge is needed to define the problem and evaluate the solution. Some work exists in analysis patterns (Fowler, 1997) and analytic models associated with

design fragments (Bachmann et al., 2002). Knowledge of the evaluation process itself (e.g., workflow, methods and techniques) (Obbink et al., 2002) can also be an important input.

- Realization knowledge (e.g., technologies, components, project management) is critical. In many cases analysis knowledge is not sufficient to evaluate the architecture. One example is when a partial implementation is needed upon which to do experimentation. In general the design must be evaluated using realization knowledge, in order to ensure that the system can be built.

### 3.2. Workflow and the concept of backlog

In all five of the architectural methods on which our model is based, the three main activities in Fig. 9 (architectural analysis, architectural synthesis, and architectural evaluation) do not proceed sequentially, but rather proceed in small leaps and bounds as architects move constantly from one to another, "growing" the architecture progressively over time. This is primarily because it is not possible to analyze, resolve, find solutions and evaluate the architecture for all architectural concerns simultaneously: the range and number of interrelated issues is just too overwhelming for the human mind, and moreover the inputs (goals, constraints, etc) are usually ill-defined, initially, and only get discovered or better understood as the architecture starts to emerge.

To drive this apparently haphazard process, architects maintain, implicitly or explicitly, a *backlog* of smaller needs, issues, problems they need to tackle, as well as ideas they might want to use. The backlog drives the workflow, helping the architects determine what to do next. It is not an externally visible, persistent artifact; on small projects it may only be a list in the architect's notebook, while for larger projects it might be an electronic, shared spreadsheet. See Fig. 11. It is therefore rarely explicitly described in software architectural methods, but we have found it to be actually present in all the methods we have studied, under some name or another, e.g., "worry list" in BAPO, "issue list" in RUP.

The backlog is constantly fed by

(a) selecting some architectural concern and/or ASR from architectural analysis,
(b) negative feedback in the form of issues or problems arising from architectural evaluation, and to a lesser extent,
(c) ideas from the architect's experience, discussions, readings, etc.

A backlog item can be thought of as a statement of the form:

- "We need to make a decision about X."
- or "We should look at Y in order to address Z."

This backlog is constantly prioritized, bringing to the front the items that seem most urgent. The tactics for prioritization will vary, mostly based on external forces. These forces include risks to mitigate, upcoming milestones, team pressure to start work on a part of the system, or simply perception of greater difficulty. Very often it is simply the need to relieve pressure from a stakeholder that drives an item to the top of the backlog (the "squeaky wheel" phenomenon).

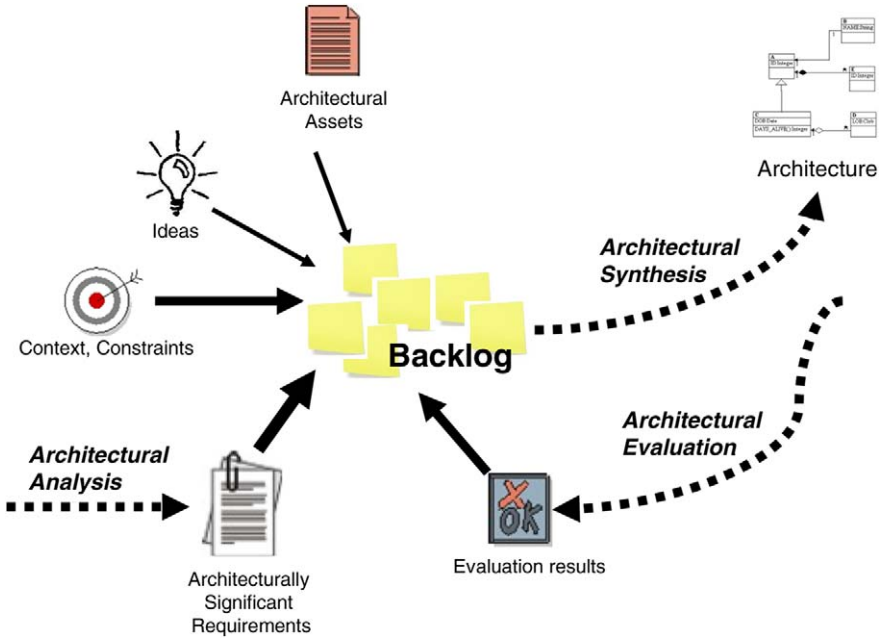


Fig. 11. Maintaining an architectural backlog.

Once a backlog item (or a small set of backlog items) is picked by the architects, they will proceed to incrementally perform architectural synthesis, making some design decisions and integrating them with the existing set of design decisions. Thus the front of the backlog serves to set the objectives for a particular iteration of architectural synthesis. Less frequently, backlog items will drive architectural analysis or architectural evaluation. Once resolved in any way (issues resolved, idea explored, requirement satisfied, risk removed, etc.), the item is removed from the backlog, and the architects proceed to the next one. If they encounter some difficulty or some input is missing, the item is returned to the backlog.

Thus the backlog is constantly changing. The cycle of adding to the backlog, reprioritizing, resolving an item, and removing an item is happening at various periods: from a few hours, to a few days, or more.

This backlog is similar to what some Agile methods use for driving projects, in particular Scrum (Schwaber and Beedle, 2002), and this is where the name came from. The backlog guides the architectural design workflow through the three main kinds of activities and provides the objectives for each iteration through the synthesis activity. The kind of tools architects may use for the backlog range from informal lists, to shared spreadsheets, to wiki-based tracking tools such as Trac.

## 4. Method comparison using the general model

The five architectural methods have been developed independently but there are many commonalities among them.

### 4.1. Side-by-side comparison

See Tables 1 and 2 for a comparison of activities and artifacts. Putting the methods side by side, helps to identify and understand this commonality as well as the important differences. The rows of the table are based on the activities and artifacts identified in the general model of the previous section.

This comparison has been an iterative process of producing a common model of design activities and artifacts, seeing how well they relate to the methods, and adjusting the model. Rather than focusing just on the creation of the architectural solution, which is the synthesis activity, the model takes a broad view of architectural design by addressing the interrelated activities of analysis, synthesis, and evaluation.

The steps of ADD follow the sequence of analysis, synthesis, and evaluation activities. Subsequent iterations of the activities follow the decomposition of the architecture – the order of which will vary (e.g., depth-first, breadth-first) based on the business context, domain knowledge, or technology.

Global Analysis from S4V plays a large role in analysis and in driving iterations through the activities. Thus it spans architectural analysis, architectural synthesis, the backlog, and describes how architectural concerns, context, ASRs, and some backlog items should be recorded. The Global Analysis artifacts, design decision table, and tables that record the relationships among views support traceability from requirements to the code (at the file and module level).

### 4.2. Commonalities

Elements the methods have in common include:

- an emphasis on quality requirements and the need to aid the architect in focusing on the important requirements that impact the architecture during analysis,
- design elements organized into multiple views during synthesis,
- and an iterative fine-grained evaluation activity (performed internally after each synthesis result by the architect) as distinct from course-grained evaluation (architectural reviews performed at key stages in the software development lifecycle).

#### 4.2.1. Common theme: dealing with quality requirements

Architecture design is often driven by quality or non-functional requirements. Since most operations of the system involve multiple components, the quality of operations cannot be guaranteed by the design of a single or a small group of components. Therefore architecture is in practice the only means to systematically address quality requirements. In this respect it is interesting to review how the five methods we discuss in this paper address quality requirements.

In RUP non-functional requirements are captured in supplementary specifications. These specifications are taken into consideration during each of the iterations. However the most important stage is often during the initial iteration when the major architectural structure is established. Non-functional requirements play an important role at this stage because the selection of the main architectural structure is mainly driven by non-functional requirements. Once an executable prototype becomes available, the capability of the system to address the non-functional requirements is assessed and when improvement is necessary further iterations are planned. Perspectives are a recent addition to RUP that help the architect address non-functional requirements (Rozanski and Woods, 2005). Perspectives are views that bring together different elements of design that affect a particular domain of concerns. Common domains of concerns are security, data integrity, availability, etc.

ADD uses quality requirements or quality attributes as the driving force for architecture design. ADD relies on a repository of architectural patterns and tactics to address the non-functional requirements. ADD follows a recursive process of hierarchical decomposition of the system. At

Table 1
Comparing methods: activities

| Activity | ADD | S4V | RUP 4 + 1 | BAPO/CAFCR | ASC |
|---|---|---|---|---|---|
| Architectural analysis | *Choose the module to decompose* (Step 1) determines the context and architectural concerns that will be addressed *Choose the architectural drivers* (Step 2a) looks for the combination of functional and quality requirements that "shape" the architecture. The QAW and quality attribute models help elicit and structure the requirements as needed | Global Analysis involves (1) identifying influencing factors; (2) analyzing them to identify their importance to the architecture, flexibility, and changeability; (3) identifying key issues or problems that arise from a set of factors | Build or extract a subset of the use case model and the supplementary specification, as key drivers for architectural design | BAPO analysis identifies those elements of context that are relevant for the architectural fit and determine the scope of the architecture. These elements are already entered into the various artifacts in the CAFCR views. Strategic scenarios can be used to explore multiple possible futures and their impact on business and architecture | Concept definition, identification and refinement of ASR, partition of ASR by software segments: runtime, development, load, etc. Thus analysis results in a collection of semi separable problems |
| Architectural synthesis | Synthesis is iterative as drivers are considered and the candidate architectural solution evolves This takes place in two parts. In the first part, *Choose an architectural pattern that satisfies the architectural drivers* (Steps 2b) architectural tactics and patterns determine the structure of the solution In the second part, *Instantiate modules* (Step 2c) and *Define interfaces* (Step 2d) the decomposition is documented and remaining requirements allocated to the structure | The fourth part of Global Analysis, identifying solution strategies, is the beginning of arch. synthesis. Then strategies are instantiated as design decisions that determine the number and type of design elements for one of the software architecture views. Design decisions can be captured in a table | During the elaboration phase, incrementally build an architecture organized along 4 different views; in parallel implement an architectural prototype | Iteratively elaborate the five CAFCR views, adding or refining artifacts suitable for the particular system. No particular order is prescribed. Often a particular quality attribute is "chased" through the different views. Architecture scenarios are defined as responses to the strategic scenarios | Address the ASR, segment by segment in an iterative process, resolving conflicts between the ASR within the same segment and integrating solutions from different segments |
| Architectural evaluation | *Verify and refine use cases and quality scenarios* (Step 2e) verifies the decomposition can collectively satisfy the requirements and prepares the child modules for their own decomposition, preparing for the next iteration of ADD. A given quality scenario might not be satisfied by the current decomposition. If it is high priority, reconsider the current decomposition. Otherwise record the rationale for why it is not supported (e.g., because of a justified tradeoff between quality attributes) | S4V splits evaluation into global evaluation (done by the architect as the design progresses ) and architecture evaluation, led by a team of external reviewers, and done at major checkpoints (e.g., to validate arch. concepts and after design is complete) | Build an executable prototype architecture to assess whether architectural objectives have been met, and risks retired (elaboration phase) | The CAFCR views are evaluated in the BAPO context to see whether they fit. Furthermore quality attributes are analyzed across the CAFCR views to determine whether the solutions in the technical views satisfy the requirements from the commercial views | Architectural decisions are evaluated with respect to ASR that they address. Typical procedure of evaluation may nclude model-based analysis (LQN, Petri nets, Q nets) simulation, prototyping, and discussion of change/use scenarios |

Table 2
Comparing methods: artifacts

| Artifact | ADD | S4V | RUP 4 + 1 | BAPO/CAFCR | ASC |
|---|---|---|---|---|---|
| Architectural concerns | Functional requirements, system quality attribute requirements, design constraints (requirements for which the design decisions are prespecified) | Influencing factors are organizational, technological, and product factors. Product factors, describing required characteristics of the product, are always architectural concerns, so are technological factors (state of technology including standards) that could affect the product | Vision document, Supplementary specification (for non-functional requirements); the Risk List identifies, among others, technical issues: elements that are novel, unknown, or just perceived as challenging | These concerns are expressed in the Customer and Application views. The overriding meta-concern is bridging the gap between customer needs, wishes, and objectives and technological realization | Each product family has lists of typical concerns that need to be addressed by products in the domain. Stakeholders contribute product specific concerns during product conception phase |
| Context | Business quality goals (e.g., time to market, cost and benefit), architecture qualities (e.g., conceptual integrity, buildability), system constraints | Organizational factors (see above) are usually context, not concerns. | Business case and Vision document | Business goals and constraints (including the scope of the market to be addressed), process goals and constraints, organizational goals and constraints | Preferred technology platforms<br>Technology/product road maps<br>Product family functional and quality requirements<br>System/hardware architecture<br>Implementation constraints |
| Architecturally significant requirements (ASR) | Architectural drivers are the combination of functional, quality attribute, and business requirements that "shape" the architecture or the particular module under consideration. To identify them, locate the quality attribute scenarios that reflect the highest priority business goals relative to the module and have the most impact on the decomposition of the module | Issue cards describe issues or problems that arise from sets of factors that, taken together, pose significant architectural challenges. These issues and their influencing factors are equivalent to the architecturally significant requirements | ASR are identified out of the requirements documents (Vision, use case model, supplementary specification), and the risk list. Some of the ASRs are expressed in the form of scenarios (use case instances) that are allocated as objectives in the upcoming iteration; this forms a requirements view (+1) | Those elements of the BAPO context that are relevant for the architectural fit and determine the scope of the architecture. Traditional types of requirements are represented in the Customer and Application views, which can be influenced by the architect in order to obtain a better BAPO fit | A specific process is used to identify ASR based on stake-holder concerns, domain and product family specific checklists, and patterns for analysis. ASR are partitioned by segments of software transformation cycle to establish semi-separable solution domains. ASR that are in the same segment are prioritized and analyzed for potential conflicts |
| Candidate architectural solutions | A collection of views, patterns, architectural tactics, and a decomposition of modules. Each module has a collection of responsibilities, a set of use cases, an interface, quality attribute scenarios, and a collection of constraints. This aids the next iteration of decomposition | Part of the four views (conceptual, module, execution, and code arch. views). These represent design decisions taken in accordance with strategies that solve one or more issues. Issue Cards capture the issues, their influencing factors, and solution strategies. Factors are listed and characterized in Factor Tables | Design decisions are incrementally captured in four views (logical, process, implementation, deployment), supplemented with a use-case view and with complementary texts, and plus an architectural prototype | Consistent and partially complete CAFCR views (Customer, Application, Functional, Conceptual, and Realization), filled with various artifacts (models, scenarios, interfaces, etc.) | A collection of patterns, frameworks, and reference architectures constitute the source for alternative decisions. An often used practice is to analyze alternatives along with any proposed solutions |

| | | | | | |
|---|---|---|---|---|---|
| Validated architecture | Architecture describes a system as containers for functionality and interactions among the containers, typically expressed in three views: module decomposition, concurrency, and deployment. The architecture is validated for satisfaction of requirements/constraints with respect to the decomposition | The description of the four views, the Issue Cards, and the Factor Tables represent the validated architecture | Baseline a complete, executable architectural prototype at the end of the elaboration phase. This prototype is complete enough to be tested, and to validate that major architectural objectives (functional and non-functional, such as performance) have been met, and major technical risks mitigated | Consistent and complete CAFCR views. Consistency means that the artifacts are mutually corresponding and quality attribute analysis shows no discrepancies (for example, all quality requirements from the Application view are satisfied by the Conceptual and Realization views). Completeness means that artifacts have been elaborated in sufficient detail to enable realization | Concepts, structure and texture for each significant segment of software transformation cycle (development/load/runtime) |
| Backlog | Information to be processed in subsequent steps including: requirements to be analyzed, decisions to be merged, patterns to be instantiated, requirements to be verified and refined<br>The order in which information is processed will vary, for example, the order of the decomposition varies based on business context, domain knowledge, and technology risk; the determination of drivers is not always a top down process, it might depend on a detailed investigation to understand the ramification of particular requirements and constraints | Supported in part by Issue Cards, which help the architect identify important issues to address and drive the bigger iterations through the activities. Issue Cards are intended to be permanent artifacts. S4V also recommends the capture of certain inputs to the backlog: Issue Cards can capture strategies (ideas) that do not work | In larger projects, an Issue List is maintained, which contains elements of the backlog. Architectural objectives are allocated to upcoming iterations, and captured in the form of iteration objectives in the iteration plan | Worry List contains: Artifacts to be completed; Quality attributes to be analyzed; Quality requirements to be satisfied; BAPO analysis to be done; BAPO issues to be improved<br><br>Design knowledge comes from the architect (or organizational memory or community best practice) and is recorded as an influencing factor. A large amount of general architectural knowledge is documented in the Gaudì website Muller (2005) | The initial backlog is a result of the analysis. As the design progresses ASR are partitioned into solvable problems and some are left on the backlog to be addressed later while some are being addressed earlier. Thus entries in the backlog represent finer and finer grained problems or issues |

each stage, depending on the desired quality attributes, a suitable architectural pattern is selected and the system is decomposed according to the pattern. ADD uses prioritization of quality attributes when architectural patterns cannot support all desired quality attributes at the same time. The ATAM architecture evaluation method complements ADD design and analysis, and is used to analyze how well design choices achieve the trade-off between all desired quality attributes.

In S4V design decisions that affect quality requirements might be reflected in multiple views. Both the conceptual and the execution view may reflect some decisions that affect dependability and performance. Each of the four views has associated quality concerns that may or may not be explicitly designated as quality attributes but are important for most systems. Code architecture view is used to minimize build time, facilitate product upgrades, etc. Module view is used to achieve reuse, and execution architecture view serves to address performance, reliability and other operational concerns. Global analysis is the process in which non-functional requirements are analyzed and strategies to address them are selected. S4V used categorization and prioritization of requirements and applies this to make sequence design process and to find proper trade-off when conflicts arise.

In BAPO/CAFCR, quality aspects do not belong to a particular view. Rather they form the cross-cutting concerns that glue the views together and enable reasoning across the views. Typically, a quality concern arises from the Customer or Application View, where it becomes clear that the system should fulfill a certain customer need or support a particular usage. In the Functional View it is formulated as a specific system quality requirement. The Conceptual and Realization Views should then make sure that this requirement is met, by a combination of applying suitable mechanisms and using suitable technology. Especially for challenging quality concerns, this process may require a significant number of iterations across the views.

ASC does not separate quality requirements from other ASR. In fact ASC expects all quality requirements to be defined by specific scenarios where the corresponding quality requirement plays an important role. This is quite similar to the way functional requirements are defined. ASC relies on segmentation of ASR to make sure that decisions affecting a particular architectural structure can be analyzed independently with respect to the corresponding quality ASR. Within the same segment ASR are prioritized to resolve conflicts and to arrive at optimal trade off choices. Specific choices in the design of each architectural structure are guided by a domain specific collection of architectural patterns which are known for specific qualities. Texture design in ASC plays a very important role in addressing a broad range of quality requirements that are orthogonal to the main decomposition pattern of the architectural structure. For example to address quality requirements related to security, flow-control, overload control, fault detection and handling all the components in the system must observe consistent behavior. However specifics of this behavior are not visible in the common architectural views and may have no impact on the design of architectural structures. Such design decisions are captured by ASC in texture design.

#### 4.2.2. Common theme: multiple views

Table 3 summarizes the use of multiple views in the various methods. The primary motivation for multiple views is separation of concerns: to group together closely related concerns, and to have looser coupling between concerns of different views.

Most of the views used in the methods can be categorized as structural views, meaning they describe architectural structure of the system. These are summarized in the first nine rows of the table. The structural views both guide the architectural synthesis activity and furnish the main documentation of this activity.

S4V and RUP 4 + 1 prescribe specific views. Although they use different names for the views, their views cover similar structures for decomposition and dependencies, instantiation and synchronization, resource usage and organization of development artifacts. They differ in support for inheritance and dataflow.

The other methods are less prescriptive and instead indicate categories or typical views. ADD produces the first articulation of the architecture typically using module decomposition and dependencies, concurrency and deployment. Other views can be added: their inclusion and the amount of detail is driven by what is needed to satisfy the architecturally significant requirements. The views for the other two methods are tied to specific concerns. ASC organizes views according to segments that are tied to the software transformation cycle: design/development, build, upgrade, start-up, operation. In BAPO/CAFCR, concerns flow from the customer needs and wishes represented in the commercial views within the context of the business, process, and organization.

Two methods, RUP 4 + 1 and BAPO/CAFCR have one or more additional views that do not describe architectural structure but instead focus on goals or requirements of the system, and these play a role mainly during the architectural analysis activity. Note that ADD, S4V, and ASC also analyze these concerns, but rather than using views they rely on other methods within the software development process to elicit this information. BAPO/CAFCR has the broadest set of views, which are intended to be successive models along a path from requirements to realization.

Finally, in addition to structural views, ASC has the notion of "texture," a replicated microstructure that applies to most components and crosscuts the structural views.

### 4.3. Variations

There are also important variations between the methods:

Table 3
The use of multiple views

| | | ADD | S4V | RUP4 + 1 | BAPO/CAFCR | ASC |
|---|---|---|---|---|---|---|
| *Structural views* | | The nature of the project determines the views. Typical views are module decomposition, concurrency, and deployment | Four views are used: conceptual, module, execution, and code architecture views | Four views are used: logical, process, implementation, and deployment views | Structural information is mainly in the Conceptual View. Its representation is not fixed by the method, but suggestions are provided | Each segment typically calls for the use of one or more structural views |
| Development time: structure exhibited in the source artifacts | Decomposition (e.g., of subsystems, modules, classes) | Module decomposition | Module Arch. View | Logical View | Conceptual View: System decomposition and component models | Functionality clusters, view of the module structure |
| | Dependencies (e.g., among modules, interfaces, classes; constrained by layers) | Often part of module decomposition | Module Arch. View | Logical View | Conceptual View: System decomposition and component models | Functionality layers, Portability layers, |
| | Inheritance (e.g., of classes) | Could be covered by an additional view | Not covered. (Modules are not usually individual classes.) | Logical View | Conceptual View: Information models | Could be covered by an additional view |
| Runtime: structure exhibited at runtime | Instantiation (e.g., components, processes, tasks, objects) | Could be covered by an additional view | Conceptual Arch. View Execution Arch. View | Process View | Conceptual View: Collaborations | Views used in Operation segment |
| | Synchronization (e.g., control flow, threads and their synchronization) | Concurrency | ConceptualArch. View Execution Arch. View | Process View | Conceptual View: Collaborations | Views used in Operation segment |
| | Dataflow (e.g., how data logically flows through system) | Could be covered by an additional view | Conceptual Arch. View | Not covered | Conceptual View: Collaborations | Could be covered by an additional view |
| Resource usage (e.g., mapping to processes, shared libraries, or other OS resources; mapping to hardware) | | Deployment | Execution Arch. View | Deployment View | Conceptual View and Realization View | Views used in Start-up segment |
| Organization of and relationships among permanent and temporary artifacts (e.g., of source, intermediate, executable files) | | Could be covered by an additional view | Code Arch. View | Implementation View | Conceptual View: Variation Models | Views used in Build segment, and in Delivery/installation/ upgrade segment |
| *Non-structural views* | | None: results of analysis activity are not captured in a view | None: results of analysis activity are not captured in a view | The +1 (requirements) view is used primarily during analysis. It focuses mainly on functional requirements. It crosscuts the four structural views | The Customer and Application Views describe goals, and the Functional View focuses on functional requirements. These are used primarily during analysis. The Realization View covers technology choices | None: results of analysis activity are not captured in a view |

- Intent – ADD was developed as a design approach based on making a series of design decisions (aided by the application of architectural tactics). Other view-based approaches were initially centered on design artifacts, with their dependencies suggesting a sequencing of activities. 4 + 1 embedded in RUP provides full process support. BAPO/CAFCR has been especially developed to support the development of product families.
- Emphasis – RUP puts a strong emphasis on incrementally building an evolutionary prototype, forcing the designers to a more experimental style of architectural design. BAPO/CAFCR puts a strong emphasis on the scoping of the architecture and once the scope of the architecture using a BAPO analysis has been established, the focus is on ensuring the consistency and the completeness of the CAFCR views. ADD puts its emphasis on constructing the architecture by applying architectural tactics (design options for the architect that are influential in the control of a quality attribute response).
- Driving forces – ADD is quality attribute scenario focused; experience suggests that a handful of these shape the architecture and all other requirements are then mapped to this structure. This fact is also recognized in ASC, which ties architecture design to architecturally significant requirements. ASR are broader than quality attributes and may include key functional requirements. RUP is mostly driven by risk mitigation.
- Architectural Scope – ASC recognizes a hierarchy of architectural scopes like product portfolio, product family, a single product, and a product release. Each architecture design project uses the enclosing scope as the context of the design.
- Process Scope – ADD provides a step for choosing the architectural drivers but its scope is such that it depends on more analysis types of activities from other methods, such as global analysis from S4V. However, S4V does not recommend or advocate specific evaluation techniques. Thus ADD and S4V complement each other in these respects.

## 5. A template for analyzing software architecture design methods

One other outcome of this work is the proposal of a kind of grid, or template, based on Fig. 10 and Tables 1 and 2, to analyze other proposed software architecture design methods, or even to drive the development of new architecture design methods; see Table 4.

To illustrate the use of this grid, we have applied it to the architectural method proposed by Garland and Anthony (2002), which we will denote here as G&A; see Table 5. Numbers in this table indicate page in the book. A more complete table could also be created for this method, containing the kind of detail given for the other methods in Tables 1–3.

Using our grid to analyze this method, we can rapidly identify the following points:

1. There is a strong focus on viewpoints and views, and the description of these views using UML to document the architecture; this occupies about half of the book.
2. Many of the activities and techniques are either sketchy (chapter 11) or refer to other sources.

Table 4
A grid to analyze a software architecture design method

| | Generic artifacts | Artifacts in X | Activities in X | Techniques and tools in X |
|---|---|---|---|---|
| | | *Has the method X provision for the following artifacts? How are they named and represented?* | *Is the method X providing activities to produce these artifacts? How are these activities named and documented?* | *What specific tools and technique is associated with the method X?* |
| Architectural analysis | – Context<br>– Requirements, and<br>– Architecturally significant requirements (ASR) | | | |
| Architectural synthesis | Candidate architectural solutions<br>– Architectural design (e.g., views, perspectives)<br>– or Prototypes<br>– Rationale | | | |
| Architectural evaluation | – Quality attributes<br>– Architectural assessment | | | |
| Overall process driver | – Backlog | | | |
| Other | | *Other key artifacts of the method X, not matching the generic ones* | *Other key activities of the method, not fitting in the boxes above.* | |

Table 5
Analysis of the Garland and Anthony (2002) method (G&A)

| | Generic artifacts | Artifacts in G&A | Activities in G&A | Techniques and tools in G&A |
|---|---|---|---|---|
| Architectural analysis | – Context<br>– Requirements, and<br>– Architecturally significant requirements (ASR) | – contextual and analysis viewpoint (p. 13, 87–108)<br>– ?? | – Analysis process, similar to OOSE | OOSE Jacobson et al. (1992) |
| Architectural synthesis | Candidate architectural solutions<br>+ Architectural design (e.g., views, perspectives)<br>+ Prototypes<br>+ Rationale | – Logical design viewpoints (decomposes in component viewpoint, logical data, layered subsystem,...) (p. 87–174)<br>– Environment/physical viewpoint (deployment, process, physical data) (p.177–200)<br>– skeleton system (p. 205) & prototyping (p. 206) | *Small activities scattered in the book, to populate the views, but not named explicitly.*<br>– partitioning strategies (208–213)<br>– changeability and dependency management (213–216) | – UML (p. 69–86)<br>– use of ADL (p. 208)<br>– pattern (p. 216–218) |
| Architectural evaluation | – Quality attributes<br>– Architectural assessment | – ??<br>– ?? | – Commonality and variability analysis (p. 202)<br>– Architecture evaluation (p. 208) | |
| Overall process driver | – Backlog | – ?? | – ?? | |

3. Building a skeleton or architectural prototype is mentioned as a possibility.
4. There seems to be no concept of Architectural Significant Requirements (except for a couple of mentions to ''key requirements'' and ''key use cases'' with no explanation how they are identified).
5. There is no mention of design rationale (except for some tracing between elements in views, and a mention of requirements tracing p. 49).
6. Architecture evaluation is only mention briefly, and no technique described, nor any artifact.
7. Beyond emphasizing iterative development in chapter 3, there is no indication of any driver for architectural activities.
8. There is very few explicit links to quality attributes.

Such an analysis would lead to the rapid identification of element from other techniques that could come to complement some of its deficiencies. For example it would be nicely complemented by the use of SEI's techniques, such QAW (Barbacci et al., 2003) or ATAM (Kazman et al., 1994; Clements et al., 2002b) , or a better linkage to quality using the ASC approach (see Section 2.5), or ADD (see Section 2.1).

Applying the grid to other methods is often impeded by the lack of detailed material on the proposed methods, associated maybe to their lack of maturity and/or limited use in industry. For example, trying to apply our grid to VTT's QAD or QADA approach (Matinlassi et al., 2002a,b), we found that views are well-articulated and justified, but there are very little description of the activities that take place. Similarly for methods associated with Model-Driven Architecture (Selic, 2004; Fowler, 2005), which are not sufficiently mature and well-described to be analyzed by our approach.

## 6. Related work

We have found four main approaches to comparing design methods. Some researchers compare the methods by comparing their results or artifacts. Others compare the activities done when following the methods. Each of these approaches breaks down further into comparisons based on applying the methods to a particular example application, or comparing the methods by classifying the artifacts or activities.

The first group compares the artifacts for an example application. Bahill et al. (1998) first provide a ''benchmark'' application to be used for comparing design methods. Then they provide a qualitative analysis of the results of applying eleven design methods to the benchmark application. Sharble and Cohen (1993) use complexity metrics to compare the class diagrams that result from applying two different OO development methods on a brewery application.

The next group also compares artifacts, but by classifying them according to what they can model. Wieringa (1998) does this for a number of structured and OO specification methods. Hong et al. (1993) do this as part of their comparison of six OO analysis and design methods, so does Fowler (1993).

The third kind of comparison examines the activities undertaken when designing particular applications. Kim and Lerch (1992) measure the cognitive activities of designers when using an OO design method versus a functional

decomposition approach. Each participant in this study designed two variants of a Towers of Hanoi application.

The approach we take in this paper falls into the fourth category, characterizing and classifying activities then comparing them across methods. Song and Osterweil (1994) use process modeling techniques to model the activities and, to a lesser extent, the artifacts of the methodologies. Although this approach is similar to ours, the approaches differ in the development of the comparison model. They decompose the activities of each methodology, then classify and compare them. Thus the classification and comparison is begun with the low-level elements. In contrast we create a general model where only one level of decomposition is done, resulting in the three activities of architectural analysis, synthesis, and evaluation and the corresponding artifacts. We then determine which elements of each methodology map to these activities and artifacts, and compare to what extent each methodology covers the various aspects of the activities and artifacts.

Hong et al. (1993) also compare activities by first characterizing and classifying them. They repeatedly decompose the activities of each method, then create a "super-methodology" that is a union of all the finest granularity of the subactivities. Each method is compared to the super-methodology. Fichman and Kemerer (1992) take a similar approach, comparing methods using the eleven analysis activities and ten design activities that are the superset of the activities supported by methods. Both of these approaches decompose the activities to very specific tasks that are tightly related to the artifacts produced by the method (e.g., Identify classes, Identify inheritance relationships). We did not want our general model to be restricted by the kinds of artifacts our five methods produce (e.g., specific views used by the method), so we did not decompose activities to the low level.

Dobrica and Niemelä (2002a,b) presented a survey and comparison of software architecture analysis methods, such as SAAM (Kazman et al., 1994) or ATAM (Kazman et al., 1996), mostly focusing on how analysis attempted to satisfy quality attributes. Their approach to comparing methods is perhaps closest to ours. However, rather than comparing architectural design methods, they are comparing methods for software architecture evaluation. Thus the eight methods have a much narrower scope, and in addition a number of them are variants of each other. Like us they compare activities and workflow at a fairly coarse granularity, but they add a few other dimensions for comparison, such as scope of method, stakeholders involved, etc.

In a more recent paper, Matinlassi (2004) extended the approach to a comparison of software product-line design methods, looking at five aspects: (1) context (goal, application domain, etc.), (2) user (stakeholders), (3) contents (method structure, artifacts, viewpoints, tool support, etc.) and (4) validation (maturity).

Kruchten (2005) shows that if software engineers were to use the term "design" analogously to the way other engineers use it, design would include "some requirements activities and all coding and testing activities." In a similar spirit, our use of the term "architecture design" encompasses analysis and evaluation activities. Architectural synthesis, the activity that goes from the problem space to the solution space is what others might equate with the term "architecture design." Fowler (1997) discusses the importance of analysis, or understanding the problem, in moving from the problem space to the solution space. (Roshandel et al., 2004) reinforce our conviction that evaluation is an integral part of architecture design. They demonstrate that the kinds of automated evaluation possible depend on the architectural view described (where each of the two views studied is represented in a different ADL).

Finally we note that our general model and the methods it is derived from are for the architecture design of new systems, not for evolving or reconstructing the architecture of existing systems. While parts of the model may be relevant for architecture evolution, when comparing our model to the Symphony architecture reconstruction process (van Deursen et al., 2004) we see that the activities and artifacts are not related at all. In both cases the activities can be categorized into (1) understand the problem, (2) solve it, and (3) evaluate the solution, but the same can be said of nearly any problem-solving activity.

## 7. Future work

The future work we envision is centered on the three fundamental architecting activities: architectural analysis, architectural synthesis and architectural evaluation (see Fig. 10). They are present in all the investigated methods. We foresee ongoing harmonization and integration of the best concepts from each of the investigated methods. This work can take two routes. The first route is the most obvious one in the sense that certain concepts in a particular method are replaced by a better one from the other methods. The second route would be along a kind of standardization effort in which the community as a whole accepts a set of standard concepts, inspired by or derived from the investigated methods.

In addition to the above described evolution along the activity axes. We expect that the notion of backlog will be elaborated further as a way to systematically relate the various activities and to guarantee "architecting progress." Yet another area of investigation would be the inclusion of notions from the agile community in the backlog concept.

Next to these more generic evolution paths we foresee that more domain specific variants of architecting methods might arise. Further new development paradigms like model-driven development (MDD) will have significant influence. And future software architecting methods will need to deal more explicitly with the system aspects.

## 8. Conclusion

In this paper we have analyzed a number of industrially validated architectural design methods. Using a general model for architectural design activity, we have identified

the common and variable ingredients of these methods. Despite the different vocabulary used for the individual methods they have a lot in common at the conceptual level. The basic architecting activities, like architectural analysis, architectural synthesis and architectural evaluation are present in all of the investigated methods. The major variation can be observed in the different details with respect to guidance and process focus across the various methods. Here the concept of backlog is crucial to relate the various activities. For our general model many of the concepts we use are already part of the IEEE 1471 (IEEE, 2000) vocabulary: views, architectural concerns, context, stakeholders, etc. Our more process-oriented model introduces the following concepts: backlog, analysis, synthesis and evaluation.

An important part of our model is the inclusion of analysis and evaluation activities as part of architecture design. While architecture evaluation has been the focus of much prior work, the emphasis is typically on identifying candidate architectures or evaluating the completed architecture. There has been far less work on incremental or ongoing evaluation, and on architectural analysis. Our model reveals these to be important research topics.

Our model also introduces the concept of a backlog as the driving force behind the workflow. The backlog is a much richer workflow concept than simply noting that iteration is expected.

We offer a systematic grid or pattern for the further analysis and comparison or other methods, as well as the creation or synthesis of new software architecture design methods (see Table 3). We hope that our increased understanding of the commonalities and differences of the various approaches will contribute to future methods that combine the strong points of the existing ones and provide specific support for software architecture design in a large variety of different contexts. As an example, two of the authors looked at ways of combining ADD and RUP $4 + 1$ by modeling ADD as a RUP activity, and found that they complement each other well (Kazman et al., 2004). ADD fills a need within the RUP: it provides a step-by-step approach for defining a candidate architecture. The RUP fills a need in the ADD by placing it in a lifecycle context; the RUP provides guidance on how to proceed from the candidate architecture to an executable architecture, detailed design and implementation.

## Acknowledgement

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

## References

America, P., Obbink, H., Rommes, E., 2003. Multi-view variation modeling for scenario analysis. In: Proceedings of Fifth International Workshop on Product Family Engineering (PFE-5), Siena, Italy. Springer-Verlag, pp. 44–65.

Bachmann, F., Bass, L., Klein, M., 2002. Illuminating the Fundamental Contributors to Software Architecture Quality (No. CMU/SEI-2002-TR-025). Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Bahill, A.T., Alford, M., Bharathan, K., Clymer, J.R., Dean, D.L., Duke, J., Hill, G., LaBudde, E.V., Taipale, E.J., Wymore, A.W., 1998. The design-methods comparison project. IEEE Transactions on Systems, Man and Cybernetics, Part C 28 (1), 80–103.

Barbacci, M.R., Ellison, R., Lattanze, A.J., Stafford, J.A., Weinstock, C.B., Wood, W.G., 2003. Quality Attribute Workshops (QAW), third ed. (CMU/SEI-2003-TR-016). Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice, second ed. Addison-Wesley, Reading, MA.

Bosch, J., 2000. Design and Use of Software Architecture: Adopting and Evolving a Product-Line Approach. Addison-Wesley, Boston.

Clements, P., Northrop, L., 2002. Software Product Lines: Practice and Patterns. Addison-Wesley, Boston.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J., 2002a. Documenting Software Architectures: Views and Beyond. Addison-Wesley, Boston.

Clements, P., Kazman, R., Klein, M., 2002b. Evaluating Software Architecture. Addison-Wesley, Boston.

Dikel, D.M., Kane, D., Wilson, J.R., 2001. Software Architecture: Organizational Principles and Patterns. Prentice-Hall, Upper Saddle River, NJ.

Dobrica, L., Niemelä, E., 2002a. Software architecture quality analysis methods. In: Proceedings of Software Reuse: Methods, Techniques, and Tools: 7th International Conference (ICSR-7), Austin, TX, USA. Springer-Verlag, pp. 337–338.

Dobrica, L., Niemelä, E., 2002b. A survey on software architecture analysis methods. IEEE Transactions on Software Engineering 28 (7), 638–653.

Fichman, R.G., Kemerer, C.F., 1992. Object-oriented and conventional analysis and design methodologies. IEEE Computer 25 (10), 22–39.

Fowler, M., 1993. A comparison of object-oriented analysis and design methods. In: Proceedings of 11th international conference on Technology of Object-Oriented Languages and Systems, Santa Barbara, California, United States. Prentice-Hall, Inc., p. 527.

Fowler, M., 1997. Analysis Patterns: Reusable Object Models. Addison-Wesley, Boston.

Fowler, M., 2005. Language Workbenches and Model Driven Architecture. Available from <http://www.martinfowler.com/articles/mdaLanguageWorkbench.html> (Retrieved 1.5.2006).

Garland, J., Anthony, R., 2002. Large-Scale Software Architecture: A Practical Guide using UML. John Wiley & Sons, Inc., New York.

Gero, J.S., 1990. Design prototypes: a knowledge representation scheme for design. AI Magazine 11 (4), 26–36.

Gero, J.S., Kannengiesser, U., 2004. The situated function–behaviour–structure framework. Design Studies 25 (4), 373–391.

Gomaa, H., 2000. Designing Concurrent, Distributed and Real-time Applications with UML. Addison-Wesley, Boston.

Hofmeister, C., Nord, R., Soni, D., 1999. Applied Software Architecture. Addison-Wesley, Boston.

Hofmeister, C., Kruchten, P., Nord, R., Obbink, H., Ran, A., America, P., 2005a. Generalizing a model of software architecture design from five industrial approaches. In: Proceedings of 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 5), Pittsburgh, PA. IEEE Computer Society, pp. 77–86.

Hofmeister, C., Nord, R., Soni, D., 2005b. Global analysis: moving from software requirements specification to structural views of the software architecture. IEE Proceedings Software 152 (4), 187–197.

Hong, S., van den Goor, G., Brinkkemper, S., 1993. A formal approach to the comparison of object-oriented analysis and design methodologies. In: Proceedings of 26th Hawaii International Conference on System Sciences, Wailea, HI, USA, pp. iv 689–698.

IEEE, 2000. IEEE std 1471:2000–Recommended Practice for Architectural Description of Software Intensive Systems. IEEE, Los Alamitos, CA.

Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G., 1992. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, Reading, MA.

Kazman, R., Bass, L., Webb, M., Abowd, G., 1994. SAAM: a method for analyzing the properties of software architectures. In: Proceedings of 16th International Conference on Software Engineering (ICSE-16), Sorrento, Italy. IEEE Computer Society Press, pp. 81–90.

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

Kazman, R., Kruchten, P., Nord, R., Tomayko, J., 2004. Integrating Software Architecture-Centric Methods into the Rational Unified Process (No. CMU/SEI-2004-TR-011). Software Engineering Institute, Pittsburgh, PA.

Kim, J., Lerch, F.J., 1992. Towards a model of cognitive process in logical design: comparing object-oriented and traditional functional decomposition software methodologies. In: Proceedings of SIGCHI Conference on Human Factors in Computing Systems, Monterey, California, United States. ACM Press, pp. 489–498.

Kruchten, P., 1995. The 4 + 1 View Model of Architecture. IEEE Software 12 (6), 45–50.

Kruchten, P., 2003. The Rational Unified Process: An Introduction, third ed. Addison-Wesley, Boston.

Kruchten, P., 2005. Casting software design in the function–behavior–structure (FBS) framework. IEEE Software 22 (2), 52–58.

Matinlassi, M., 2004. Comparison of software product line architecture design methods: COPA, FAST, FORM, KobrA and QADA. In: Proceedings of 29th International Conference on Software Engineering (ICSE 2004), Edinburgh, Scotland. IEEE, pp. 127–136.

Matinlassi, M., Niemela, E., 2002b. Quality-driven architecture design method. In: Proceedings of 15th International Conference of Software & Systems Engineering and their Applications (ICSSEA 2002), Paris, France, Conservatoire National des Arts et Métiers, vol. 3, session 11.

Matinlassi, M., Niemelä, E., Dobrica, L., 2002a. Quality-driven architecture design and quality analysis method (Report VTT256), VTT Technical Research Centre of Finland, Espoo, Finland, 128p.

Muller, G., 2005. The Gaudi Project Website. Available from: <http://www.gaudisite.nl/>.

Nord, R., Tomayko, J., Wojcik, R., 2004. Integrating Software-Architecture-Centric Methods into eXtreme Programming (XP) (CMU/SEI-2004-TN-036). Software Engineering Institute, Pittsburgh, PA.

Obbink, H., Müller, J.K., America, P., van Ommering, R., Muller, G., van der Sterren, W., Wijnstra, J.G., 2000. COPA: a component-oriented platform architecting method for families of software-intensive electronic products (Tutorial). In: Proceedings of SPLC1, the First Software Product Line Conference, Denver, Colorado.

Obbink, H., Kruchten, P., Kozaczynski, W., Hilliard, R., Ran, A., Postema, H., Lutz, D., Kazman, R., Tracz, W., Kahane, E., 2002. Report on Software Architecture Review and Assessment (SARA), Version 1.0. Available from: <http://philippe.kruchten.com/architecture/SARAv1.pdf>.

Ran, A., 2000. ARES Conceptual Framework for Software Architecture. In: Jazayeri, M., Ran, A., van der Linden, F. (Eds.), Software Architecture for Product Families Principles and Practice. Addison-Wesley, Boston, pp. 1–29.

Roshandel, R., Schmerl, B., Medvidovic, N., Garlan, D., Zhang, D., 2004. Understanding Tradeoffs among Different Architectural Modeling Approaches. In: Proceedings of 4th Working IEEE/IFIP Conference on Software Architecture (WICSA-04), Oslo, Norway, pp. 47–56.

Rozanski, N., Woods, E., 2005. Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, Boston.

Schwaber, K., Beedle, M., 2002. Agile Software Development with SCRUM. Prentice-Hall, Upper Saddle River, NJ.

Selic, B., 2004. The pragmatics of model-driven development. IEEE Software 20 (5), 19–25.

Sharble, R.C., Cohen, S.S., 1993. The object-oriented brewery: a comparison of two object-oriented development methods. ACM SIGSOFT Software Engineering Notes 18 (2), 60–73.

Song, X., Osterweil, L.J., 1994. Experience with an approach to comparing software design methodologies. IEEE Transactions on Software Engineering 20 (5), 364–384.

Soni, D., Nord, R., Hofmeister, C., 1995. Software architecture in industrial applications. In: Proceedings of 17th International Conference on Software Engineering (ICSE-17). ACM Press, pp. 196–207.

van der Linden, F., Bosch, J., Kamsteries, E., Känsälä, K., Obbink, H., 2004. Software product family evaluation. In: Proceedings of Software Product Lines, Third International Conference, SPLC 2004, Boston, MA. Springer-Verlag, pp. 110–129.

van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L., Riva, C., 2004. Symphony: view-driven software architecture reconstruction. In: Proceedings of 4th Working IEEE/IFIP Conference on Software Architecture (WICSA-04), Oslo, Norway. IEEE, pp. 122–134.

Wieringa, R., 1998. A survey of structured and object-oriented software specification methods and techniques. ACM Computing Surveys 30 (4), 459–527.

**Christine Hofmeister** is an assistant professor in Computer Science & Engineering at Lehigh University, in Bethlehem, PA. Her current research areas are software architecture and component-based systems. She received her A.B in Mathematics from Bryn Mawr College, M.S. in Computer Science from Lehigh University, and Doctorate from the University of Maryland, with a dissertation on dynamic reconfiguration of distributed systems. Prior to joining Lehigh's faculty in 2000, she spent a number of years at Siemens Corporate Research in Princeton, NJ. While at Siemens, she coauthored the book *Applied Software Architecture* with Robert Nord. crh@eecs.lehigh.edu

**Philippe Kruchten** is a professor of software engineering at the University of British Columbia, in Vancouver, Canada. His current interests are software process modeling and software architecture, and the impact of culture on global engineering projects. Prior to UBC, he spent 16 years at Rational Software (now IBM), where he was associated as a consultant with several large-scale defense and aerospace projects around the world, and where he developed the Rational Unified Process®, a software engineering process. He also spent 8 years at Alcatel in France, developing telephone switches. He has a mechanical engineering diploma and a doctorate degree in information systems from French institutions. He is the author of three books about RUP. pbk@ece.ubc.ca

**Robert L. Nord** is a senior member of the technical staff in the Product Line Systems Program at the Software Engineering Institute (SEI) where he works to develop and communicate effective methods and practices for software architecture. Prior to joining the SEI, Dr. Nord was a member of the software architecture program at Siemens, where he balanced research in software architecture with work in designing and evaluating large-scale systems. He earned a Ph.D. in Computer Science from Carnegie Mellon University. He is co-author of *Applied Software Architecture and Documenting Software Architectures: Views and Beyond.* rn@sei.cmu.edu

**Henk Obbink** is a principal scientist at Philips Research Laboratories in Eindhoven. He heads the architecture research team for software-intensive healthcare systems. His research interests have included computer systems, communication systems, defense systems, and consumer products. He received his doctorate in chemistry and physics from the University in Utrecht. He is a member of the IFIP Working Group 2.10 on Software Architecture and the steering committees of the Working IEEE/IFIP Conference on Software Architecture and the Software Product Line Conference. He was recently one of the guest editors of the March/April IEEE special Issue on the Past, Present and Future of Software Architectures. henk.obbink@philips.com

**Alexander Ran** is a Research Fellow at Nokia Research Center (NRC), in Cambridge, MA, USA. He is currently a Principal Investigator in

collaboration between Nokia and Massachusetts Institute of Technology focusing on architectures for task-oriented user interfaces. Prior to that he lead a research group working on software architecture analysis for improving performance characteristics of mobile phones, conducted software architecture reviews, contributed to establishing software architecture processes, and served as a consultant for software architecture groups of several Nokia business units. alexander.ran@nokia.com

**Pierre America** received a Master's degree from the University of Utrecht in 1982 and a Ph.D. from the Free University of Amsterdam in 1989. Since he joined Philips Research in 1982, he has been working in different areas of computer science, ranging from formal aspects of parallel object-oriented programming to music processing. During the last few years he has been working on software and system architecting approaches for product families. He has been applying and validating these approaches in close cooperation with Philips Medical Systems. pierre.america@philips.com