Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA5).
Pittsburgh, PA, November 6-9, 2005. Named one of the five best papers of the conference.

Generalizing a Model of Software Architecture Design from Five Industrial Approaches

Christine Hofmeister
Lehigh University
Bethlehem, PA, USA
crh@eecs.lehigh.edu

Philippe Kruchten
University of British Columbia
Vancouver, B.C., Canada
pbk@ece.ubc.ca

Robert L. Nord
Software Engineering Institute
Pittsburgh, PA, USA
rn@sei.cmu.edu

Henk Obbink
Philips Research Labs
Eindhoven, The Netherlands
henk.obbink@philips.com

Alexander Ran
Nokia
Burlington, MA, USA
alexander.ran@nokia.com

Pierre America
Philips Research Labs
Eindhoven, The Netherlands
pierre.america@philips.com

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 5 approaches have a lot in common and match more or less the "ideal" pattern we introduced.

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 [4, 5, 6, 7, 8, 12, 15]. Some of these were cast and published as architecture design methods or systems of concepts, processes and techniques for architecture design [16, 22, 26, 27].

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 is concerned with continuous operation, live upgrade and interoperability. Other essential differences may include methods designed for large organization 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 large software systems that: require involvement of and negotiation among multiple stakeholders; are developed by large, often distributed teams over extended periods of time; and have to address multiple possibly conflicting goals and concerns.

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 [4], developed at the SEI; Siemens' 4 Views (S4V) method [16], developed at Siemens Corporate Research; the Rational Unified Process 4 + 1 views (RUP 4+1) [21, 22] developed and commercialized by Rational Software, now IBM; Business Architecture Process and Organization (BAPO) developed primarily at Philips Research [1, 26], and Architectural Separation of Concerns (ASC) [27] 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.

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 4describes the five contributing methods using terms and concepts of the general model, and discusses the commonalities and differences between the contributing methods. Section 5 discusses related work, and Section 6 concludes the paper.

2. Five Industrial Software Architecture Design Methods

2.1. Attribute-Driven Design

The Attribute-Driven Design (ADD) method [4], developed at the SEI, is an approach to defining software architectures by basing the design process on the architecture's quality attribute requirements. It 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.

In ADD, the architects, for each module to decompose, 1) choose the architectural drivers, 2) choose an architectural pattern that satisfies the drivers, 3) instantiate modules and allocate functionality from use cases, and represent the results using multiple views, 4) define interfaces of the child modules, and 5) verify and refine the use cases and quality scenarios, making them constraints for the child modules.

2.2. Siemens' 4 views

The Siemens Four-Views (S4V) method [16, 32], 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.

These views are developed in the context of a recurring Global Analysis activity. For Global Analysis, the architect identifies the organizational, technological, and product factors that influence the architecture: requirements, desired system qualities, organizational constraints, existing technology, etc. From these the key architectural issues or challenges are identified; typically they arise from a set of factors that, taken together, will be difficult to fulfill. Design strategies are proposed to solve the issue, and they are applied to one or more of the views. In addition to interleaving Global Analysis with the view design, the architect is expected to iterate among the design tasks of the four views.

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. RUP includes an architectural design method, using the concept of 4+1 views (RUP 4+1) [21, 22]; four views to describe the design: logical view, process view, implementation view and deployment view, using a use-case view to relate the design to the context and goals.

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

2.4. Business Architecture Process and Organization

The BAPO/CAFCR approach [1, 24, 26, 33], 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), process (P), and organization (O). For that purpose, the five CAFCR views are described: 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. The architecture is complete when there is sufficient information to realize the system and the quality attribute analysis shows no discrepancies.

2.5. Architectural Separation of Concerns

Architectural Separation of Concerns (ASC) or ARES System of Concepts [27], developed primarily by Nokia, is a conceptual framework based on separation of concerns to manage complexity of architecture design. ASC relies on the fact that concerns related to different segments of the software transformation cycle (typically including design, build, upgrade, load, and run time) are often separable. In addition to design of architectural structures for each segment, ASC pays special attention to design of texture – replicated microstructure that addresses concerns that cannot be localized within the main structure.

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 Figure 1).

Because of the complexity of the design task, these activities are not executed sequentially. Instead they are used repeatedly, at multiple levels of granularity, 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 [13, 14], which Kruchten applies to software design in [23].

3.1. Architectural Design Activities & 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" [18]. 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" [18]. 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 in-

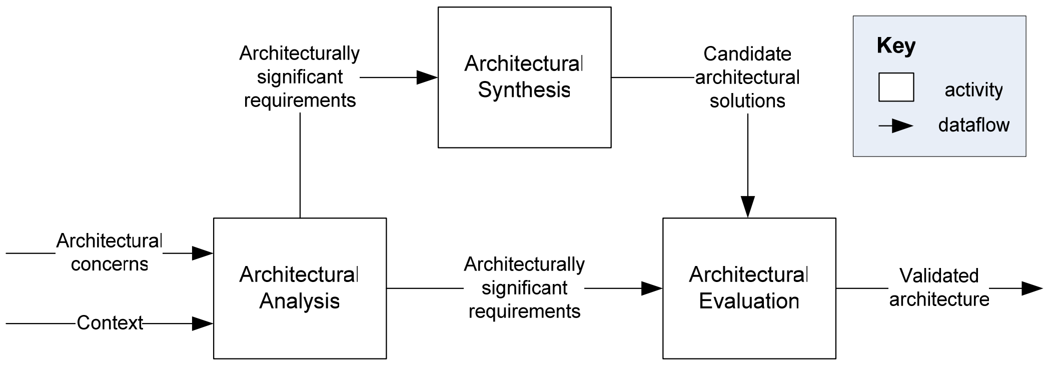

Figure 1: Architectural design activities

stead 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" [25]. 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 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 multiple iterations 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 [11] and analytic models associated with design fragments [2]. Knowledge of the evaluation process itself (e.g., workflow, methods and techniques) [25] can also be an important input.
- Knowledge necessary to produce the system (technologies, components, project management). 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

In all five of the architectural methods on which our model is based, the three main activities in Figure 1 (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 and only get better understood or discovered 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 and ideas they might want to use. The backlog drives the workflow, helping the architect 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 Figure 2.

The backlog is 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."

The 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.

Table 1 – Comparing methods: Activities

Activity	ADD	S4V	RUP 4+1	BAPO/CAFCR	ASC
Architectural analysis	Step 2a: Choose the architectural drivers. Quality attribute models help elicit and structure the requirements.	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 as key drivers for architectural design	BAPO analysis identifies those elements of the BAPO context that are relevant for the architectural fit and determine the scope of the 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	Steps 2b: Choose an architectural pattern that satisfies the architectural drivers; 2c: Instantiate modules and allocate functionality from the use cases using multiple views; 2d: Define interfaces of the child modules.	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.	Gradually build during the elaboration phase architecture organized along 4 different views; in parallel implement an architectural prototype.	Elaborate the five CAFCR views, adding or refining artifacts suitable for the particular system	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	Step 2e: Verify and refine use cases and quality scenarios and make them constraints for the child modules. Note: this step bridges evaluation and analysis, preparing for the next iteration of ADD.	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).	Evaluation of the CAFCR views in the BAPO context and quality attribute analysis across the CAFCR views	Architectural decisions are evaluated with respect to ASR that they address. Typical procedure of evaluation may include 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.	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)	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 Technology/product road maps Product family functional and quality requirements System / hardware architecture Implementation constraints

Artifact	**ADD**	**S4V**	**RUP 4+1**	**BAPO/CAFCR**	**ASC**
Architec-turally significant requirements (ASR)	Architectural drivers are the combination of functional, quality attribute, and business requirements that "shape" the architecture. To identify them, locate the quality attribute scenarios that reflect the highest priority business goals and have the most impact on the decomposition.	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 ASR 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 stakeholder 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, and architectural tactics. The architecture also has associated with it refined scenarios that show mapping from requirements to decisions and also aid the next iteration of design.	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. Consistent in the sense that these 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). Complete in the sense that artifacts have been sufficiently elaborated 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.	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 don't 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. 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 [24].	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.

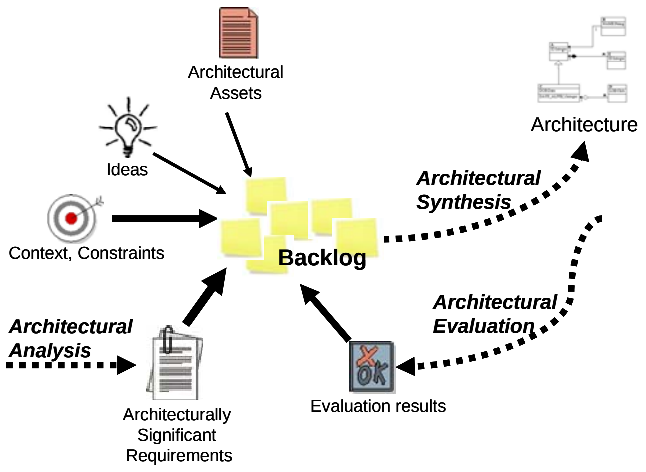

Figure 2: Backlog

Once a backlog item (or a small set of backlog items) is picked by the architects, they will proceed to incrementally do architectural synthesis, making some design decisions and integrating them with the existing set of design decisions. Thus it 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, an 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, in particular Scrum [29]. It guides the workflow through the three kinds of activities and provides the objectives for each iteration through the synthesis activity. In addition to using some kind of backlog, the architect should also make sure that each iteration of each activity is preceded by the setting of objectives for that step.

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 & 2 for a comparison of activities and artifacts By putting the methods side by side, we are working 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 models. We take a broad view of architectural design activities and see that the methods address interrelated activities centered on 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 attribute 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 life-cycle).

4.3. Variations

There are also important *variations* between the methods:

- **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 is putting 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 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.
- **Description** – Although four specific views are recommended, the view-related aspects of S4V are limited to the second part of arch synthesis. Thus other views could be substituted or added while still using all the other parts of S4V. The views used for architecture design in ASC are tied to architectural structures that are important for the specific system, which in turn, are determined by the ASR. Thus ASC offers a process to determine which views should be used for architecture design of a specific system. The views used in ADD are determined by the ASR, though typically there is one for each of the three kinds of viewtypes: module, component & connector, and allocation [6].

5. 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.* [3] 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 & Cohen [30] 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 [35] does this for a number of structured and OO specification methods, and Hong *et al.* [17] do this as part of their comparison of six OO analysis and design methods.

The third kind of comparison examines the activities undertaken when designing particular applications. Kim & Lerch [20] 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 & Osterweil [31] 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.* [17] 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 & Kemerer [10] 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 & Niemela's approach to comparing methods is perhaps closest to ours [9]. 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 [23] Kruchten 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." In [11] Fowler discusses the importance of analysis, or understanding the problem, in moving from the problem space to the solution space. Roshandel *et al.* [28] 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 [34] 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.

6. 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 [18] 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 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 [19]. 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.

References

[1] P. America, H. Obbink, and E. Rommes, "Multi-View Variation Modeling for Scenario Analysis," in *Proceedings of Fifth International Workshop on Product Family Engineering (PFE-5)*, Sienna, Italy, 2003, Springer-Verlag, pp. 44-65.

[2] F. Bachmann, L. Bass, and M. Klein, *Illuminating the Fundamental Contributors to Software Architecture Quality*, CMU/SEI-2002-TR-025, Software Engineering Institute, Carnegie Mellon University, 2002.

[3] A.T. Bahill, M. Alford, K. Bharathan, J.R. Clymer, D.L. Dean, J. Duke, G. Hill, E.V. LaBudde, E.J. Taipale, and A.W. Wymore, "The design-methods comparison project," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 28, no. 1, 1998, pp. 80-103.

[4] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed., Reading, MA: Addison-Wesley, 2003.

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

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

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

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

[9] L. Dobrica and E. Niemela, "A survey on software architecture analysis methods," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, 2002, pp. 638-653.

[10] R.G. Fichman and C.F. Kemerer, "Object-oriented and conventional analysis and design methodologies," *IEEE Computer*, vol. 25, no. 10, 1992, pp. 22-39.

[11] M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison Wesley, 1997.

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

[13] J.S. Gero, "Design prototypes: A knowledge representation scheme for design," *AI Magazine*, vol. 11, no. 4, 1990, pp. 26-36.

[14] J.S. Gero and U. Kannengiesser, "The situated function–behaviour–structure framework," *Design Studies*, vol. 25, no. 4, 2004, pp. 373-391.

[15] H. Gomaa, *Designing Concurrent, Distributed and Real-time Applications with UML*, Boston: Addison-Wesley, 2000.

[16] C. Hofmeister, R. Nord, and D. Soni, *Applied Software Architecture*, Boston: Addison-Wesley, 2000.

[17] S. Hong, G. van den Goor, and S. Brinkkemper, "A formal approach to the comparison of object-oriented analysis anddesign methodologies," in *Proceedings of Twenty-Sixth Hawaii International Conference on System Sciences*, Wailea, HI, USA, 1993, pp. iv 689-698.

[18] IEEE, *IEEE 1471:2000--Recommended practice for architectural description of software intensive systems.*, Los Alamitos, CA: IEEE, 2000.

[19] R. Kazman, P. Kruchten, R. Nord, and J. Tomayko, *Integrating Software Architecture-Centric Methods into the Rational Unified Process*, Technical report CMU/SEI-2004-TR-011, Software Engineering Institute, 2004.

[20] J. Kim and F.J. Lerch, " 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, 1992, ACM Press, pp. 489-498.

[21] P. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, vol. 12, no. 6, 1995, pp. 45-50.

[22] P. Kruchten, *The Rational Unified Process: An Introduction*, 3 ed., Boston: Addison-Wesley, 2003.

[23] P. Kruchten, "Casting Software Design in the Function-Behavior-Structure (FBS) Framework," *IEEE Software*, vol. 22, no. 2, 2005, pp. 52-58.

[24] G. Muller, *The Gaudi Project website, at* http://www.extra.research.philips.com/natlab/sysarch/index.html, 2005.

[25] H. Obbink, P. Kruchten, W. Kozaczynski, R. Hilliard, A. Ran, H. Postema, D. Lutz, R. Kazman, W. Tracz, and E. Kahane, *Report on Software Architecture Review and Assessment (SARA), Version 1.0*, February 2002.

[26] H. Obbink, J.K. Müller, P. America, R. van Ommering, G. Muller, W. van der Sterren, and J.G. Wijnstra, COPA: A Component-Oriented Platform Architecting Method for Families of Software-Intensive Electronic Products. Tutorial for the First Software Product Line Conference, Denver, Colorado, August 2000. 2000,

[27] A. Ran, "ARES Conceptual Framework for Software Architecture," in M. Jazayeri, A. Ran, and F. van der Linden, ed., *Software Architecture for Product Families Principles and Practice*, Boston: Addison-Wesley, 2000, pp. 1-29.

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

[29] K. Schwaber and M. Beedle, *Agile Software Development with SCRUM*, Upper Saddle River: Prentice-Hall, 2002.

[30] R.C. Sharble and S.S. Cohen, "The object-oriented brewery: a comparison of two object-oriented development methods," *ACM SIGSOFT Software Engineering Notes*, vol. 18, no. 2, 1993, pp. 60-73.

[31] X. Song and L.J. Osterweil, "Experience with an approach to comparing software design methodologies," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, 1994, pp. 364-384.

[32] D. Soni, R. Nord, and C. Hofmeister, "Software Architecture in Industrial Applications," in *Proceedings of 17th International Conference on Software Engineering*, 1995, ACM Press, pp. 196-207.

[33] F. van der Linden, J. Bosch, E. Kamsteries, K. Känsälä, and H. Obbink, "Software Product Family Evaluation," in *Proceedings of Software Product Lines, Third International Conference, SPLC 2004*, Boston, MA, 2004, Springer-Verlag, pp. 110-129.

[34] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva, "Symphony: View-Driven Software Architecture Reconstruction," in *Proceedings of 4th Working IEEE/IFIP Conference on Software Architecture (WICSA-04)*, Oslo, Norway, 2004, IEEE, pp. 122-134.

[35] R. Wieringa, "A Survey of Structured and Object-Oriented Software Specification Methods and Techniques," *ACM Computing Surveys*, vol. 30, no. 4, 1998, pp. 459-527.