Architectural Requirements Engineering: Theory *vs*. Practice

Robert W. Schwanke
Siemens Corporate Research, Inc.
robert.schwanke@scr.siemens.com

Abstract

This paper discusses how architectural requirements engineering fits into an overall software development process in the concept and definition phases of a project. It defines a reference process identifying the "ideal" artifacts and their interrelationships, describes some key technical activities that are useful for producing these artifacts, and captures some practical experience in commercial projects.

1. Introduction

Theory and practice are generally the same, in theory.
– Anonymous

This paper is an attempt to reduce the wide gap that so often occurs between the theory and practice of architecture requirements engineering in real software development projects. Too frequently, an organization fails to capitalize on a good software architecture, for reasons such as: the development process is not aligned to profit from it; the key stakeholders do not buy into it; or, it simply solves the wrong problem.

The "theory" aspect of this paper offers a reference process for architecture requirements engineering and related activities. The artifacts and dependencies are foremost in the process definition, because (in practice) most software analysis and design activities are artifact-driven and opportunistically scheduled, so modeling the data of the process gives more insight than trying to model control. These artifacts are sequenced within a simple phase-and-gate framework that shows the phases and decision points where the project can be cancelled, sent for rework, or approved to enter the next phase.

The central artifact, for the purposes of this paper, is the Global Analysis document, first introduced by Hofmeister, Nord and Soni [1]. The software architects at Siemens Corporate Research have used Global Analysis in half a dozen projects since the book was written. This paper gives a brief review of the approach, updated based on our experiences.

The "practice" aspect of this paper offers hints on doing software architecture effectively and efficiently. Doing it effectively means building stakeholder consensus and buy-in for both the technical design and the development plan, by obtaining agreement on the requirements and other constraints that they must satisfy, and convincing people that the design and the plan do satisfy the requirements and constraints. Doing it efficiently means focusing attention and other resources on the important issues, at the right times, while tracking, but living with, a large number of less-important inconsistencies, unsatisfied constraints, and other unknowns.

The "theory" and "practice" aspects are intermingled in the presentation, in hopes of reducing that gap.

2. An Architecture-Centered Process

The architecture group at Siemens Corporate Research provides technical and project management consulting services to a wide variety of software development groups within Siemens (primarily in the United States but occasionally in Europe). The process described here is our starting point: how we would like to do architecture if we could. Naturally, every real project has constraints that prevent this, such as the legacy process used previously, the legacy artifacts providing input to the project, and the skills and comfort zones of the key players. After presenting this idealized process we will discuss some of the adaptations that may be necessary to use it in a real project. (Hereafter, the word "we" usually refers either to the SCR architecture team or to the team and the readers of this paper, depending on context. "I" refers to the author.)

The process definition has four major parts: the artifacts produced, the dependencies between artifacts, the phases of the project, and the rules for coordinating artifacts. It does not specify any activities separately from artifacts, other than reviews, because most activities are artifact driven, anyway, and best discussed in the context of the artifacts they produce. This process definition also

does not describe how to assign artifacts to teams and how to coordinate teams; that would take another whole paper.

For brevity, this process description only covers the parts of the process most related to architecture requirements engineering. It assumes that the project has already completed its "idea phase", and sufficient resources have been allocated to carry out the concept phase. It also assumes that the project is predominantly a software development project, and therefore does not address hardware design, manufacturing, or separate "system" artifacts. The principles presented here certainly apply to such systems, but would require a longer treatment.

3. Artifacts, dependencies, and activities

Figure 1 shows the artifacts and dependencies of the process. Each arrow represents a dependency: "X → Y" means "information in artifact X depends on (or is justified by) information in artifact Y." Typically, each individual item in X, such as a specification, is annotated with references to specific items in Y, such as requirements.

Although many of the artifacts are familiar to the reader, a few comments are in order.

3.1. Stakeholder list

A stakeholder is an *accessible* person who *represents* a class of persons who will be significantly affected by architectural decisions. The stakeholder must be accessible to the project team to answer questions and review artifacts. Sometimes the stakeholder is a member of the class (e.g. a testing manager can speak for all testers), but sometimes he is appointed to represent the class from outside (e.g. a marketing analyst who speaks for the end user.) Every such class should be considered for representation, including such diverse classes as salespersons, buyers, end users (could be multiple classes), software testers, installers, trainers, and help desk attendants.

The stakeholder list clarifies exactly who cares about the project, why they care, and why that matters. As the list develops, it may go through several refinement steps. The first draft might identify all the candidate stakeholder classes, with stakeholder names, where known, and explanations of why each class is important. As the list stabilizes, the classes without named stakeholders become action items, either to find stakeholders or to explain why the class is not important enough to be represented. Later on, the list may also prioritize stakeholders or define different groups of stakeholders, typically for allocating stakeholders to artifact reviews.

If an organization has a well-developed business process model, showing all the actors in the product's target business domain, many of these actors will require stakeholders to represent them. However, since such business process models are still uncommon in current practice, this software process does not assume that such an artifact exists.

3.2. Stakeholder requests

Stakeholder requests document the concerns of stakeholders. Some stakeholders produce artifacts that are defined in the company software process; others just write white papers, send e-mails, and attend reviews. For the purpose of this process definition, we assume that any input from a stakeholder can be documented as a stakeholder request. Since a stakeholder could request almost anything, we are usually only interested in *qualified* stakeholder requests, which have been reviewed and approved as being worth addressing.

Most stakeholders are "outside" the architecture team. The chief architect and the project manager are often also stakeholders. However, their requests should be qualified by someone outside the project, so that they do not appear to abuse their right to write requests.

3.3. Features

Requirements, in general, define properties of the product, in terms that external (outside the development project) stakeholders recognize and understand. Features are requirements at a coarse granularity, suitable for use in sales presentations and for allocating to product releases (in the Build/Release Plan). A feature could be a specific service that the product provides, but it could also be an attribute of the whole product, such as "fault-tolerant."

The Features artifact should specify which stakeholders' interests it represents. Often, it is limited to customer stakeholders, and becomes the "voice of the customer." Eventually, each feature should be annotated with references to qualified stakeholder requests that justify the feature.

This process avoids using the terms "functional" and "non-functional" to characterize requirements and specifications, because these terms mean different things to different people.

3.4. Detailed requirements

Detailed requirements spell out what the feature level requirements mean in terms that are testable, but still in the stakeholders' language. We often find that 15-30 detailed requirements are needed per feature, to be

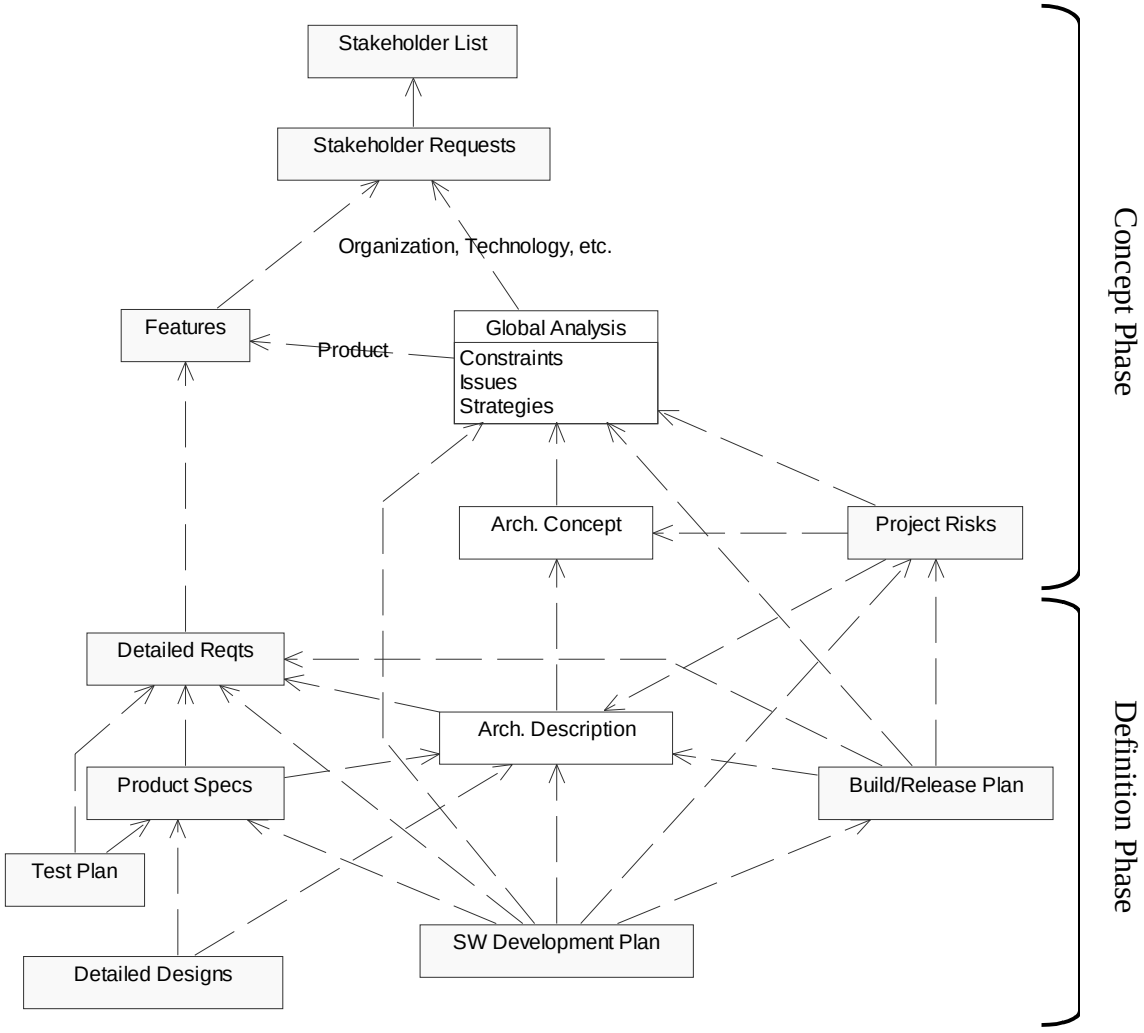

Figure 1. An architecture-centered process

complete and unambiguous. A single detailed requirement can support several features.

For user interface requirements, a UI prototype is strongly recommended, to capture both the intent and the details of features, particularly aesthetic features like "easy to use" and "common look and feel". Eventually, the prototype can be captured in a conventional requirements artifact by copying screen shots into it, with accompanying text and models (e.g. state transition diagrams or message sequence charts) to nail down exactly what the product should do.

Some projects do not need both detailed requirements and product specifications. (Cf. Section 6.4)

3.5. Product specifications

Specifications define properties of the product and its parts, in technical terms that the developers, testers, documenters, project engineers, maintainers, and other "downstream" stakeholders understand. We often observe an expansion factor of 2-5 between detailed requirements and product specifications. A key, *theoretical* difference between detailed requirements and product specifications is that a requirement should state what the product should do, without reference to any particular implementation, whereas the product specification describes the externally-visible properties of the externally-visible interfaces identified in the architecture description.

When requirements and specifications are written by different teams, the product specifications may represent "push back" by the development team, conveying the message, "We heard what you said you wanted, but this is what we think we can build."

3.6. Feasibility and global analysis

The architecture team has the responsibility to analyze everything that may affect the success of the project, determine what the critical issues are, propose strategies to address those issues, and then develop an architecture consistent with this analysis. It is called "global" analysis because it is looks at the project from all directions (from the perspectives of all the stakeholders), and because the critical issues and strategies are typically also global, cutting across subsystem boundaries and appearing in more than one view of the architecture. The global analysis artifact contains three kinds of items: factors, issues, and strategies.

3.6.1. Factors. A *factor* is any fact that is likely to constrain or otherwise influence the architecture. Some factors can be written as requirements, but others cannot be so rigorously stated. Normally, we expect requirements to state properties of the product, and to be correct, unambiguous, and testable, whereas a factor is often unverified, ambiguous, or uncertain, and may describe something other than the product itself. Furthermore, we usually use a stylized language to write requirements, e.g. "The product shall cost no more than $300 per floating license per year." Imposing stylized language on factors would interfere with communication. For example, is it clearer to write, "The product shall be developed using programmers whose previous experience does not include ASP technology" or "Our programmers don't know ASP"? The first alternative is verbose, passive voice, vague, and might actually be incorrect, if there is an option to hire a few ASP developers. The second alternative succinctly captures one fact that constrains the architecture.

Factors can come from anywhere. For convenience they are grouped into three categories: product factors (typically derived from features); technology factors, which involve the technologies available to implement the product; and, organizational factors, which involve properties of the company or other organization that is developing the product. These categories are further grouped into sub-categories, such as product performance, services provided, programming tools, technical standards, staff skills, schedule constraints, and so on. These categories and sub-categories should not be considered exhaustive; any stakeholder request might draw attention to a significant factor, whether or not it fits neatly into one of the categories.

A factor should have a standard structure. We typically record the following properties:

Category and sub-category
Name
Unique ID
Brief statement of the factor.
Flexibility (what sort of "wiggle room" is there in the factor today?)
Changeability (how might it change later on?)
Impact (how does it affect the architecture?)
Authority (what or who justifies this constraint?)
Owner (person responsible for text of this factor)
Status
Priority

Previously, we have tried to capture each factor as a row in a table of factors, but found several practical problems: the columns became too narrow, there was lots of white space, cross-referencing the factor name or number was awkward, and our usual word processing tool didn't handle word-level change tracking very well in tables. So, I recommend organizing the factors into categories and subcategories, giving each constraint its own sub-sub-section within its sub-category, and using a standard text structure within the subsection. Figure 2 suggests a format.

1. Organizational Constraints

1.3 Management

1.3.5 Buy reporting subsystem

(Factor-37)

The reporting subsystem should be based on a commercial product, e.g. Crystal Reports

Flexibility. Previous reporting system was implemented in-house, so buying COTS is not a rigid requirement. But competitors are already doing this.

Changeability. Reporting features may become more specialized, making the "buy" option less advantageous.

Impact. Buying the market leading product has low development cost, risk, and time to market, but introduces licensing costs and reduces product differentiation.

Authority: Features 135, 136, and 139, and SR 174 from Jim Smith, who has interviewed customers concerning reporting features.

Figure 2. Textual presentation of a constraint

Although storing factors in an ordinary text document is often practical, we are also considering using a

requirements management tool to manage architecture factors, but have no experience yet.

Even when a factor is written in the form of an architecture requirement, there are two important differences between a marketing feature and an architecture factor: range and uncertainty. The range of the architecture factor is a way of capturing a set of similar requirements that vary only in certain dimensions. For example, "The architecture must support customers with transaction rates between 1 million and 100 million per day." This factor does not say that any particular customer installation has to perform well across the whole range, nor does it even say that any particular release of the product has to handle the whole range. For example, there could be two variants of the product for low-volume and for high-volume customers.

The dimensions that factors frequently span include numerical ranges, members of a product family, successive generations of a product or family, and ranges of calendar time. For example, "In four years, the architecture must support GUIs on handheld devices." This allows the architect to choose between designing the infrastructure for handheld GUIs now, or leaving a placeholder for them and designing them in two years, by which time the technology will have changed anyway. Note that variation over calendar time is different from the "stability" of a feature or a factor. In the example above, the factor is very stable, but is chronologically positioned four years in the future.

Allowing uncertainty in an architecture factor allows the architect to document the problem before the uncertainty is resolved. For example, now that Internet services are beginning to be offered aboard airplanes, the marketing department might envision the day when radiologists, traveling on airplanes, want to download medical images to their laptops. A factor might be written, "the product must be evolvable to support medical image viewing over low-bandwidth, high-latency Internet connections." Such a statement would normally be disqualified as a requirement, because "evolvable" is a vague word. But, such statements are valuable to the architect, despite their vagueness. Note that "uncertainty" is also somewhat different from "stability", because the statement of the factor explicitly captures the sense in which it is uncertain, whereas calling a requirement "unstable" has more to do with its status.

It may be useful to describe range and uncertainty as separate attributes of a factor, but we haven't tried it yet.

Unlike requirements catalogs, the collection of architecture factors does not have to be complete. Global analysis prioritizes them, finds conflicts and tradeoffs between them, and finally reduces them to a set of key issues that shape the architecture. The less important factors will likely be ignored, for most purposes, so missing a few of them is not important.

3.6.2. Issues. An architectural issue is a potential conflict or tradeoff between two or more factors – usually many more! For example, the issue "Aggressive Schedule" might be stated as, "The project probably can't be completed in 14 months if we have to train our programmers in Java, add new tools to our development environment, and implement all 75 major features, 7 of which require exploratory prototyping." Normally, there are many potentially significant issues, but certain ones rapidly emerge as the most critical. Fortunately, because of the inherent uncertainty of many factors, it is not necessary to satisfy all of them. The architects must identify and prioritize the issues, so that the architecture and the project development plan can be designed to address the most critical ones. The others are managed as project risks, to be addressed later.

3.6.3. Strategies. A strategy is a decision that addresses one or more significant issues. The strategy may be technical, managerial, or a combination. For example, if the issue is "ASP programming is best done in Java, but our programmers only know C++", the architect and project manager could choose to "retrain our programmers in JSP", "buy an ASP development environment for C++", or "use some C++ programmers to write C++ applets, and retrain others to write JSP."

3.6.4. Putting it all together. The original description of Global Analysis[1] suggested using "Issue Cards", where each card defines and discusses one issue, then defines and discusses strategies for addressing it. This approach doesn't work well when a strategy addresses several issues – which many of them do. Instead, I recommend documenting issues and strategies by embedding them in a coherent presentation of the rationale for the architecture. The first part of the Global Analysis artifact should be a catalog of factors, as described above. The second part should present the significant issues and strategies for resolving them. Each issue should be documented in a format that is partly structured and partly informal. The structured part includes backward references to the most relevant factors and references to the most important strategies for dealing with the issue. Each strategy could be defined at the first place it is referenced in the text, perhaps in a sidebar or an inset box. The informal part of the issue description discusses how the factors interact to shape the issue, and how the proposed strategies would help to resolve the issue. The third part of Global Analysis should be a free-flowing, coherent rationale for the proposed architectural approach. This presentation technique emphasizes coherent argumentation more than cataloging and cross-referencing the issues and strategies, as we have sometimes done in the past.

3.7. Architecture concept

This artifact should not be confused with the conceptual view of the architecture. The architecture concept artifact is written for external stakeholders, is informal, and presents the essential concepts of the architecture in notations and words that are comfortable for the stakeholders. It is typically based on a paper "proof-of-concept", which describes a slice of the system using the proposed architecture approach. It then uses portions of this system slice to illustrate the concepts it presents, depending on what is needed to educate and convince the stakeholders

3.8. Architecture description

This artifact is the complete description of the architecture, typically following the IEEE standard 1471-2000. Note that the architecture description depends on the detailed requirements, but the architecture concept does not. This is so because (a) the architecture concept should not be sensitive to small changes in requirements, and (b) the architecture concept usually needs to be relatively complete, reviewed, and approved before authorizing the expense of developing detailed requirements.

3.9. Project risks

This process does not specify how project risks are described and managed, but many risks are identified in the course of global analysis and architecture design. Any key issues that are not fully resolved by the strategies, as well as any major assumptions made while drafting the architecture description, become risks that must be managed.

3.10. Build Plan and Release Plan

The build plan defines a sequence of internal development milestones, or builds, with each module, product specification, and detailed requirement to be implemented in a specified build. We typically recommend that the individual builds be scheduled about 6 weeks apart, to provide rapid feedback on the effectiveness of the design and maintain a common understanding of the system across the development team. Some of the builds are designated as releases that the customer will see (although perhaps only as a demo).

3.11. Software development plan

The software development plan depends on the global analysis artifact for strategies and on the architecture description as the basis for a bottom-up cost estimate. At SCR we use an estimation methodology that annotates the module view of the architecture with development cost estimates, collecting the assumptions needed to make those estimates. The modules become tasks in the plan; the assumptions become risks to be managed. For more on architecture-centric software project management, see Paulish's book of that title [2].

4. Project phases

Figure 1 divides the artifacts into two phases: the concept phase and the definition phase. This division signifies the phase in which each artifact receives its first critical review and sign-off. Of course, each artifact is revised in subsequent phases, as needed.

5. Coordinating artifacts and activities

Other than at the end of each phase, the process does not specify an order in which the artifacts are finished and reviewed, because this ordering varies widely between projects, depending on many "soft" factors. Instead, we expect that the artifacts will be written by different people, and will therefore evolve concurrently. In order to manage this efficiently, it is important to identify where information provided in one artifact is used in another, and to cross-review artifacts between teams. It is equally important to allow, but document and manage, incompleteness and inconsistency between artifacts.

5.1. Incompleteness and inconsistency

Recording incomplete links is especially valuable in the global analysis artifact. It is true that, *eventually*, every issue should be based on factors, and that those factors should derive their authority from other artifacts. However, global analysis frequently identifies potentially significant factors long before the relevant stakeholders have raised concerns about them. Rather than waiting to document the factor until the stakeholder writes a request, the architect should put a note in the authority field of the factor, describing where he expects the authority will come from. The note could even include a shortened draft of the item (e.g. a feature) that he would like to see added to some other artifact. (If necessary, the architect might have to write his own stakeholder request.) Similar techniques should be used wherever links between artifacts may appear. (Incomplete links are very much like

the “fat references” used in the Pattern Languages of Programming community.)

5.2. Cross-reviewing artifacts between teams

One of the most important heuristics for effective artifact review is, “Choose reviewers who depend on the information they are reviewing.” In this process, the dependency links between artifacts are an excellent guide for identifying reviewers. Consider, for example, the detailed requirements. The people who wrote the features (if different) will want to be sure that the detailed requirements accurately define the features. The people who will be writing product specifications will want to make sure they receive good-quality detailed requirements, to make their job easier. The people who have to write tests against the detailed requirements will want to be sure the requirements are testable.

Using the dependency links to identify reviewers also reduces the chances of “disconnect” in a project. Many of us have experienced projects where artifacts were “thrown over the wall” from one group to another, leaving both groups dissatisfied. Having such a wall between requirements engineering and development, for example, tempts developers to ignore the requirements they don’t understand or don’t like. By using cross-reviewing to strengthen communication and buy-in between teams, such problems can be reduced.

5.3. Reviewing links between artifacts

Whenever an artifact is formally reviewed, the links between it and other artifacts should also be reviewed. This includes both the artifacts on which it depends, and the artifacts that depend on it. *This is very important for building consensus!* When a requirements engineer signs off on the global analysis artifact, his signature should mean that, except for noted defects, (a) all relevant, previously documented features have been referenced in the right places in the analysis, (b) any relevant, not-yet-documented features have been discussed and given incomplete references in the analysis, and (c) *he agrees with the analysis of these features*. On the other side, when the global analyst signs off on the Features artifact, his signature means that every feature needed to justify significant factors, whether or not they have been published yet, appears either in the artifact itself or the review notes.

The review notes then become action items for resolving incomplete and inconsistent links. However, the resolution does not necessarily need to happen immediately. Some of the items may be very low priority, some may require further investigation, and some may not be resolvable until a later stage of the work.

5.4. Validation and Consistency

Each significant item in each artifact, such as a feature, an issue, or a specification, is subject to validation in the course of review. Part of the definition of consistency between artifacts is that a link from an item in artifact X to an item in artifact Y is only fully consistent when the item in artifact Y has been validated. Sometimes the validation is simply a yes/no decision on whether the item should be included in the artifact; in other cases, included items are further assigned to “buckets” that represent different development/release cycles. In the latter case, of course, the bucket assignments of X and Y must be compatible.

5.5. Phase reviews

At the end of each phase there is a review, often called a *gate*, whereby managers outside the project determine whether to continue funding the project. There are actually two separate questions to answer: “Is the project ready to move into the next phase?” and “Is the company ready to pay for it?” Some organizations actually have two separate reviews, because some of the decision-makers are different for these two questions.

Each phase review specifies the artifacts that will be considered at the review. In this process, each artifact is considered at each phase review after its introduction, if it is relevant to the decision. Naturally, these artifacts must have been reviewed individually prior to the phase review. However, they don’t have to be absolutely complete and consistent, as long as there is an action plan for resolving the inconsistencies.

This approach to handling incompleteness and inconsistency is especially valuable when the development organization is undergoing change to adapt to new or improved development processes. Often artifacts cannot be completed and reviewed in the same order as the chain of dependencies. Because the show must go on, explicitly documenting incompleteness and inconsistency for later resolution is often the best approach.

6. Merging the Processes

Because there is little standardization of software development processes across organizations, the process defined above will normally have to be adapted for use in the context of an organization’s existing process. This section describes some of the adaptations that are likely to be necessary, and some of the issues that may need resolving.

6.1. Enriching the concept phase

Many existing processes focus mainly on defining product features in the concept phase. If possible, one should insist on doing some feasibility analysis in the concept phase, before committing the resources necessary to do a complete high-level design. This feasibility analysis would then include global analysis and the architecture concept, as well as a UI prototype if the product has a user interface.

6.2. Regrouping information in artifacts

Sometimes it is necessary to combine logically separate artifacts into a single artifact, or, for reasons of scale, to divide a single logical artifact into a main artifact and several subsidiary artifacts. However, it can also be necessary to redefine an existing artifact so that it carries more architecture information than it has in the past.

For example, a process may define a "System Concept" artifact, typically due at the end of the concept phase, which has historically been a very informal document. This might be a good place to put the Architecture Concept.

6.3. Caring for stakeholders

Many existing processes do not address all the important stakeholders. For example, a Market Requirements artifact might be limited to addressing the logical functionality of the product, ignoring non-functional features. This typically arises from a focus on end-users, ignoring the needs of other stakeholders like system administrators, buyers, and commissioning engineers. The remedy might be to add another artifact to carry non-functional features, or to address quality attributes in the Global Analysis artifact.

More generally, the process should be adapted so that every important stakeholder has a "voice" in some artifact – in Global Analysis, if not elsewhere.

6.4. Detailed requirements vs. specifications

Although in theory there is a clear logical distinction between a detailed requirement and a product specification, in practice the two artifacts are frequently combined. We have found several reasons for this:

- Cost pressure: maintaining two descriptions of strongly related information is more expensive than maintaining one.
- Skill shortage: good requirements engineers are under-appreciated, and therefore in short supply!
- Process: without an architecture description, the only input to the product specification is the detailed requirements, anyway, so why not combine them?
- Disconnect: because of inadequate communication between those who write features and those who write specifications, it is not obvious that the detailed requirements are missing.
- Difficulty: it is actually quite difficult, in many instances, to write a good set of detailed requirements without referring to implementations, especially early in the definition phase when so many questions are unsettled.

One way often suggested to overcome these difficulties is to introduce a prototype, often as a controlled process artifact, whose purpose is to facilitate consensus-building between requirements analysts and developers. The most common types, of course, are the UI prototype and the proof-of-concept prototype. The detailed requirements and product specifications do not need to be written down until the prototype stabilizes and is reviewed. Then, both artifacts can be derived from it, if both are needed.

7. Future Work

We are currently investigating how to extend our process to effectively use rigorous models for domain analysis, requirements analysis, design, and testing.

Acknowledgements

The process diagram in Figure 1 was produced by rapid iteration based on feedback from my colleagues, each of whom contributed different expertise: project management (Dan Paulish), problem statements (David Laurance), architectural concept and consensus building (Dilip Soni), global analysis (Bill Sherman and Rod Nord), requirements engineering (Brian Berenbach), and user interface design (Nuray Aykin).

References

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

[2] Paulish, D. *Architecture Centric Software Project Management: A Practical Guide*, Boston: Addison-Wesley, 2002.