Architectural Prototyping: An Approach for Grounding Architectural Design and Learning

Jakob Eyvind Bardram, Henrik Bærbak Christensen and Klaus Marius Hansen
University of Aarhus
Computer Science Department
Aabogade 34, 8200 Aarhus N, Denmark
{bardram,hbc,klaus.m.hansen}@daimi.au.dk

Abstract

A major part of software architecture design is learning how specific architectural designs balance the concerns of stakeholders. We explore the notion of "architectural prototypes", correspondingly architectural prototyping, as a means of using executable prototypes to investigate stakeholders' concerns with respect to a system under development. An architectural prototype is primarily a learning and communication vehicle used to explore and experiment with alternative architectural styles, features, and patterns in order to balance different architectural qualities. The use of architectural prototypes in the development process is discussed, and we argue that such prototypes can play a role throughout the entire process. The use of architectural prototypes are illustrated by three distinct cases of creating software systems. We argue that architectural prototyping can provide key insights that may otherwise be difficult to obtain before a system is built. Furthermore, they define skeleton systems that serve as communication means and knowledge transfer among stakeholders.

1 Introduction

Exploring unknown territory may serve as metaphor for building new software architectures. The planet Mars is a fine example of unknown territory that is presently drawing a lot of attention. Many techniques help us in such an exploration. Satellites like Mars Global Surveyor, Mars Odyssey, and Mars Express orbit the planet and chart the territory using a wide range of different techniques. However, even with the most advanced techniques there is no substitute for "being there", which is the reason why rovers like Mars Spirit and Opportunity are now scouting the surface providing data that would otherwise be impossible to acquire. In a similar vein, methods, techniques, and catalogues have been designed for helping architects in contemplating, evaluating, and constructing architectures. However, abstract as they are, they provide the orbital view; and more often than not, architects need to "be there" to explore issues in sufficient detail to make qualified decisions.

In this paper, we explore *architectural prototypes* as the metaphorical equivalent of "being there", i.e., scouting the architecture-to-be-implemented by actually building low-cost executable systems that reflect essential architectural qualities of a potential target system's architecture.

Our main contribution is to explore the notion of architectural prototype in detail, to discuss its characteristics and to describe how it differs from traditional software prototypes. We hope thereby to initiate a classification and investigation of the concept in order to introduce well-defined terminology and techniques into the vocabulary of every software architect.

Our paper is organized as follows. We present a definition of what makes architectural prototype a particular class of prototypes and outline characteristic properties in Section 2. We then present three cases of defining, constructing, and evaluating architectural prototypes, emerging from three radically different projects in Section 3. Next, in Section 4 and 5, we present an initial attempt at classifying types of architectural prototypes and relate these to the overall process of architecture and system development. Finally, we conclude in Section 6.

2 Architectural Prototyping

The seminal paper by Floyd [14] acknowledges the difficulty in accurately defining "prototyping" in connection to software development. She takes her starting point in a definition of engineering/manufacturing prototypes and then discusses the properties that are different or characteristic for software prototypes.

Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)
0-7695-2172-X/04 $ 20.00 © 2004 IEEE

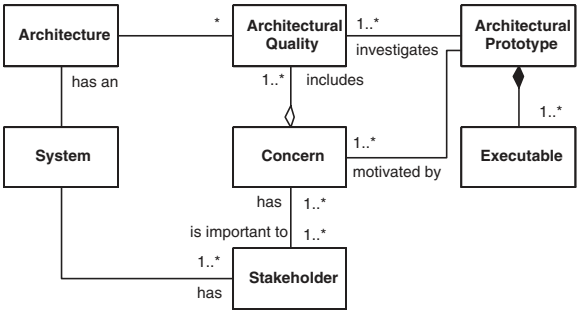

Figure 1. Ontology of Architectural Prototypes

In a similar vein, we may characterize *architectural prototyping* as a special class of software engineering prototyping with a number of distinct properties. Floyd defines prototypes as executable systems that "involve an early practical demonstration of relevant parts of the desired software". Prototypes are primarily interesting as a process rather than the prototype itself as a product—it is used as "a learning vehicle providing more precise ideas about what the target system should be like."

2.1 Definition

We define the concept of architectural prototype as follows:

> An *architectural prototype* consists of a set of executables created to investigate architectural qualities related to concerns raised by stakeholders of a system under development. *Architectural prototyping* is the process of designing, building, and evaluating architectural prototypes.

Figure 1 relates this definition to the software architecture ontology defined by IEEE [23]. The *System* is the entity being built, which may encompass, e.g., single applications, frameworks, and product lines. *Stakeholders* have interests in the System expressed through a number of *Concerns* regarding the development of the system. *Architectural Qualities* are specific Concerns of interest in architectural prototypes and includes qualities such as performance, reliability, or modifiability [2]. The Concerns and their included Architectural Qualities define the reasons for building *Architectural Prototypes* through a set of *Executables*.

Architectural prototypes may investigate Architectural Qualities in a number of ways. We elaborate this below through a focus on defining characteristics of architectural prototypes.

2.2 Characteristics

A number of properties characterize architectural prototypes and set them apart from traditional software engineering prototypes.

First, *architectural prototypes are constructed for exploration and learning of the architectural design space*. Floyd characterized prototypes as "learning vehicles", but though this was a general statement, the focus was (and has traditionally been) on "software intended as a direct support for human work." In contrast, architectural prototypes have the architecture itself as focus, largely ignoring the intent of the system under study. This has interesting relations to what Bass et al. describe as the *architecture business cycle* [2, § 1]. One important influence on architecture, mediated through the architect, is the architect's own experience. Success with a certain architectural style, for instance, inclines the architect to use it again even if inappropriate. Architectural prototyping is a way to break this circle as it allows the architect to learn and experiment at a low cost. That is, the architect's "design vocabulary" is expanded.

Second, *architectural prototyping addresses issues regarding architectural quality attributes* in the target system. It is the primary challenge for the architect to define an architecture that balances conflicting qualities in an appropriate way. Typical issues the architecture must address are: "will it perform adequately?", "is it cost-efficient to maintain?", "can we guaranty high availability?" Architectural prototypes are important vehicles for striking this balance as qualities can be observed and measured directly: performance measurements can be made, experience is gained whether the programming interface is understandable, etc.

In our example section, we outline architectural prototypes developed for the purpose of analyzing the qualities of performance versus maintainability (Section 3.1), modifiability and buildability (Section 3.2), and performance (Section 3.3) respectively.

Third, *architectural prototypes do not provide functionality per se*. As noted by Bass et al. "functionality" (defined as the ability of the system to do the work for which it was intended [2, § 4.1]) is largely non-architectural in nature. Architectural prototypes are skeleton systems that facilitate functionality; they do (usually) not deal with the functionality itself. This is an important property because it is what makes architectural prototyping cost-efficient.

Fourth, *architectural prototypes typically address architectural risks*. Architectural risks may be to find the proper balance among opposing qualities like performance and modifiability but it may also be related to, e.g., buildability concerns. A typical example of the latter is prototyping to acquire knowledge about technologic platforms (e.g., J2EE), programming models (e.g., knowledge-based system), and other aspects that may be new to the organization

or to the architects.

And fifth, *architectural prototypes address the problem of knowledge transfer and architectural conformance*. A major challenge in architecture centric development is ensuring that the architecture "as-built" is identical to the architecture "as-designed". Here the architecture prototype serves as a reference system that demonstrates key architectural decisions, patterns and styles to the development team as concrete source code, and may serve as the backbone system for "growing" the required functionality on top of. Thus architectural prototypes that demonstrate the desired balance of qualities are most likely not thrown away but used as skeleton systems.

Architectural prototypes are executable systems but our definition does not dictate how these are produced. In the case studies described below, the architectural prototypes were build by hand using traditional integrated programming environments. However, stronger tool support, e.g., to produce executables from Unified Modeling Language (UML; [20]) or Architecture Description Language (ADL) specifications, may be available for specific problem domains.

3 Case Studies

The following sections present architectural prototypes in action through three cases of architectural prototyping. The examples have been chosen as to represent a broad spectrum of types of architectural prototypes (see Section 4) and problem domains – viz., real-time embedded software, information systems, and (research in) pervasive computing.

3.1 Closed-loop Process Control

This project dealt with a major revision of a Danish vendor's closed-loop process control system for engine control [7, 9]. The existing system was based on software running on a single, embedded CPU system with fixed connections to sensors (sensing the controlled process) and actuators (controlling the process). As example, the system is used to control elevators, cooling by fans, conveyor belts, etc.

The challenge was to design an extensible, distributed, system that allowed for a flexible, expandable, system by adding "option-cards" having their own processing power and memory. For instance, one may extend the system to handle more sensors simply by adding more A/D converter cards. Thus the software had to be redesigned to handle A) distribution, as data comes from separate embedded cards and B) dynamic reconfigurations. Our focus was primarily on issue A.

The main quality of the new architecture was *hard real time performance*. All measurements of the process (feed-back values) have to be sampled, processed (like averaging), communicated to the control algorithm, and feed to the engine control hardware within a timeframe of about a few milliseconds. The second driving quality was *maintainability* in order to ease maintenance and lower the software error rate. The existing system had had high performance but had turned out to be difficult to maintain.

Thus the challenge was to find a suitable architecture that could provide "as much maintainability as possible" and still perform.

We responded to this challenge by designing four architecture prototypes for further analysis. Three of these were variants of the *pipe-line* architectural style while the last was based on a *repository* style.

The three pipe-line architecture prototypes differed on the dataflow protocols. In the *pull variant* the control algorithm (data sink) requests data from components in a chain all the way to the components responsible for sampling feed-back value input (data sources). In the *push variant* it is instead the input components that publish data to subscribers, and data flows using an Observer pattern protocol [15] through the chain of components to the control algorithm. Finally, a *mixed push/pull variant* was defined where input components publish data to an intermediate cache component (push), while the control algorithm requests data from the cache (pull). Common to all three variants was that data flows through (sometimes remote) methods calls between components.

The repository prototype instead defines a *distributed database component*. Logically, there is one central database storing feed-back values. At the deployment level, each CPU card has a local database component that use the network to guaranty that the database contents is consistent across all nodes; a task that the underlying CAN bus is directly designed for [6]. In this scenario, input components simply write feed-back values to the distributed database, and the control algorithm component reads them. Thus data is communicated essentially through variable reading and writing.

The architecture evaluation phase contained two, interwoven, activities. In the analysis activity, consequences for performance and maintainability were discussed and analyzed in meetings between stakeholders; in the construction activity actual prototypes were constructed, both on hardware similar to the target platform [13] as well as on the Java platform. The stakeholders included lead designers from the company and a university research group, both with architectural and software design insight.

Prototypes were built for the pushed variant pipe-line and for the repository-based architecture. The pulled variant pipe-line was used in the present production system soft-

ware and its characteristics therefore well known; the mixed variant pipe-line was not constructed as sufficient knowledge was acquired during the construction of the other prototypes to deem it less interesting.

The conclusion of the evaluation phase was that both the process as well as the prototypes built were essential in finding a high performance architecture with good maintainability characteristics. The process of building the prototypes and later reviews of the "lessons learned" during meetings spurred a lot of discussion and pin-pointed trouble spots that had not been identified during earlier more abstract discussions. As an example, the pushed variant pipeline seemed promising, due to its low coupling property, but the experiments showed a major drawback about timing issues: each new feed-back value sampled in the input component essentially triggers a new update sequence through to the control algorithm that is thus run several times within each control interval—which clearly exceeded the time slot allotted. One may argue that we should have identified this (in hindsight, obvious) problem without building prototypes. However, it shows the value of prototypes as even seasoned architects may oversee the obvious whereas executing software does not; also there was great value in actually demonstrating the problem to stakeholders.

The outcome was that the repository variant had superior properties: Performance was adequate while the variable access paradigm for data communication provided strong decoupling between components while being easy to understand and maintain. Furthermore, time expenditure calculations for each component were greatly simplified as data read/write is constant time while method call estimates depends on a complete flow analysis. Thus the repository architecture prototype was chosen as skeleton system for the final product.

3.2 The Dragon Architectural Prototypes

The Dragon project developed a series of evolutionary prototypes of a customer service system for a globally distributed container shipping company [10]. The project progressed through two major phases in which an iterative and incremental development strategy was used (see Figure 2):

- *Exploring major functional concerns through horizontal prototyping*. This phase explored uncovered functional requirements through among others prototyping, ethnography [19], and participatory design [16] since a major risk of the project was uncertainty and complexity with regards to functionality and scope. The phase resulted in a series of prototypes which were reviewed and approved by the shipping company.

- *Exploring software architecture through vertical prototyping*. Building on the horizontal prototypes developed in the first phase, the Dragon project explored technical concerns such as integration with legacy systems, creating Computer-Supported Cooperative Work (CSCW) support, and experimenting with binary component technology [11].

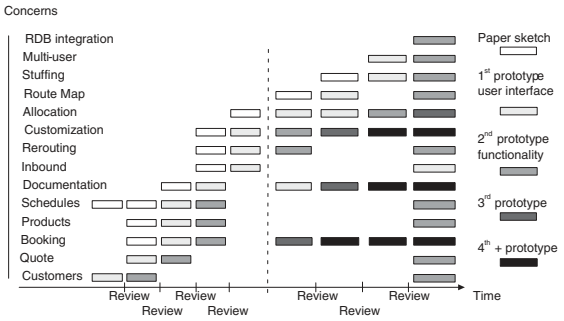

Figure 2. Activities in the Dragon Project

The second phase of the Dragon project created evolutionary architectural prototypes based on horizontal prototypes. The decision to build architectural prototypes on top of a functional prototype was, among others, based on that the realizability of the system needed to be investigated with respect to unknown and conflicting requirements. The first phase, which included numerous presentations to staff, workshops, and reviews, meant that requirements were reasonably complete in terms of use cases as well as user interaction. Achieving such a level of completeness of functional requirements may indeed often be necessary to ground architectural design of interactive system in the advent of high uncertainty of requirements [3]. An example of a user interface design which had architectural implications and was crucial for usability was the possibility for customer service agents to partially specify, e.g., values to a search in a legacy system for products related to a quote, let the search work in the background, and at the same time let the customer service agent continue to work on other parts of the quote.

In the context of this, the architectural prototypes of the Dragon project explored, first, a horizontal, suitable division of responsibilities among components and associated connectors and, secondly, the implications of adding vertical support for, e.g., integration with legacy systems and support for collaborative work.

The main architectural quality that was investigated in the second phase of the Dragon project was buildability. The earlier exploratory activities had shown that the initial architecture was problematic with respect to development time, deployment etc. It was thus investigated through prototypes whether and how these problems could be alleviated. Other architectural qualities explored encompassed modifiability, usability, and integrability. Modifiability and

usability were explored through the creation of architectural prototypes on top of functional prototypes, showing that a system containing components pertaining to major domain abstractions and with the required usability characteristics could indeed be built in a way in which new domain components could be added to the system with little effort [11]. Integrability was investigated through proof-of-concept prototypes simulating integration with legacy systems based on data from the functional prototypes.

The stakeholders of the architectural prototypes were the development team (which included a university research group, external consultants, and company representatives) and decision makers in the company. The development team was concerned with whether and to which degree a software architecture supporting the functional prototypes could be designed in the context of the quality requirements. From the perspective of the decision makers from the company, the purpose was to remove as many risks as possible in a situation characterized by high uncertainty and complexity with respect to functional requirements before a full-scale development of the production system was launched; a development for which the prototypes were to be used as a main requirements specification. The architectural prototypes were instrumental in reaching the goals of these stakeholders.

3.3 Activity-Based Computing Framework

The Activity-Based Computing (ABC) Framework [8, 1] is a research system for pervasive computing. The basic idea in activity-based computing is to help users focus on their work activities (or tasks) rather than low-level issues in contemporary computer systems, like files, applications, network connectivity, or device interoperability. Such higher-level support is essential in a pervasive computing paradigm, because users cannot cope with the details of handling many different devices.

A core part of the ABC framework is an infrastructure (or middleware layer), which stores, manages, and distributes activities on behalf of the users. To explore how to design the architecture for this middleware layer, a couple of architectural styles were explored using architectural prototypes; a call-based client-server style, an event-based system consisting of several independent components, and a peer-to-peer architecture. The purpose of these prototypes was to explore different approaches to an architecture for activity-based computing and to experiment with the feasibility and efficiency of different styles, features, and architecture patterns.

First, the well-known call-based client-server architectural style was applied. The prototype explored how activities could be stored, managed, and distributed to clients using a central 'Activity Server'. Clients would call the server and get a copy of an activity. The server would also handle parallel updates and inconsistencies. This architecture was, however, put to a test when real-time collaboration was introduced to the ABC framework. Two or more users could simultaneously collaborate on an activity, and the ABC framework would then enable them to see what each other was doing and hear each other. This is in the CSCW research tradition often referred to as desktop conferencing or real-time application sharing. The call-based client server architecture turned out not to be well-suited for this kind of real-time updates of e.g. moving scrollbars and other user-interface changes.

Hence, a prototype was built based on an event-based publish/subscribe architecture instead. Each of the clients participating in a real-time activity sharing session would both publish and subscribe to changes in the state of the client applications. This architecture proved to handle the real-time collaborative features of the ABC framework in a much more elegant way.

Further exploration has been done lately, building a prototype that use a peer-to-peer type of architecture where changes to the state of clients in a session is distributed directly between the clients, and is not handled by a central publish/subscribe broker.

Performance in real-time activity sharing is crucial and further examination of how to make the most efficient implementation of this event-based architecture were carried out. In particular an experiment was set up to measure which architecture pattern was the most efficient in terms of response time and CPU and network load; the 'Reactor' architectural pattern or the 'Leader/Follower' architectural pattern [22].

A final example of architectural prototyping done during the development of the ABC framework was to explore the issue of portability, and more precisely the feature of binding time in the control of the system. Portability is a key concern in pervasive computing and also a difficult architecture quality to meet, because there is (or will be) a wide range of heterogeneous devices in a pervasive computing world. In the first architectural prototype of the ABC framework, applications used in the framework were added at compile time. Hence, portability was supported at compile time, by compiling and linking to the applications, that the ABC framework would support on its clients. The prototype was built in Java and could hence be ported across devices that had a Java virtual machine, which could run the J2SE version of Java. This is a traditional way of handling portability, i.e. to link and compile applications at compile time and then have a hardware abstraction in terms of a virtual machine. However, the aim of the ABC framework is to look up applications on run-time and the next version of the prototype was hence built to explore this issue of the framework. To achieve this there is a registry on each device

that can participate in the ABC infrastructure. This registry is used by the framework to look up applications that can implement certain services. For example, an activity dealing with word processing would require a text editor and a spell checker. When activated, this activity would look up a 'text editor' and a 'spell check' service on various devices. Hence, on a Windows PC the text editor might be 'Write' and the spell checker taken from MS Word. On a PDA the text editor might be the default one and the 'spell check' service might not be implemented.

4 Classification of Architectural Prototypes

An architectural prototype can serve several purposes at various stages in the development process. We can distinguish between two basic types of architectural prototypes:

- **Exploratory prototypes** used to clarify requirements to the architecture together with its stakeholders, to explore aspects of the target system, and used to discuss alternative architectural solutions.
- **Experimental prototypes** used to gauge the adequacy of a proposed architecture, or details hereof, before investing in a large-scale implementation of it. Experimental prototypes are typically used to measure software architecture qualities; e.g., to make quantitative measurement of performance or portability.

Exploratory and experimental prototypes can be used to clarify and learn about architectural questions and issues in a broad sense, ranging from questions of the overall architectural style to more detailed questions about a specific architectural pattern. This is illustrated in Table 1. Let us consider these approaches to architectural prototyping in detail.

4.1 Exploratory Architectural Prototypes

Exploratory prototypes are used to focus on basic issues, questions or critical problems of the architecture. Their main purpose is to facilitate communication and learning among stakeholders of an architecture, like the architect(s), the customers, the users, and the developers. Exploratory prototypes are used when there is a high degree of uncertainty about how to build the target system, and is thus often used in the early stages of software development. In such situations, a practical demonstration of an executable architectural prototype can facilitate a discussion of pros and cons, which is grounded in 'real code' and not merely speculations of what can be achieved and what not. Typically, a number of alternative prototypes are built and compared against one another and they are typically thrown away after use.

When planning a new system, a new module, or changes to an existing one, the architect might consider using different architectural styles. In this case a number of architectural prototypes can be build to explore pros and cons of different styles. Such exploration can also be used to help the architect to think beyond the styles/patterns that s/he is usually applying (cf. [2, 5]). For example, in the 'Closed-Loop Process Control' case (Section 3.1) the architect explored two types of architectural styles, each with a different approach to meet the requirement of the system. Architectural qualities like buildability and conceptual integrity are also investigated in such early, overall architectural prototypes. Alternative exploratory prototypes can be judged according to how they meet core quality requirements to the target system. This judgment is typically done by the system's different stakeholders in common and the exploratory prototypes are used to ground the discussions in "real code". For example, in the Dragon case, a proof-of-concept of how to integrate with legacy data was used among the stakeholders of the target system to learn and to reduce risk.

Once an architectural style has been decided upon it can be refined by exploring different features of the architecture like exploring types of components, connectors, or the topology of the architecture. Prototypes can be built to explore how the choice of certain features of the architecture influences the overall target system. For example, in the ABC framework presented in Section 3.3, the feature of synchronicity in the control structure was explored. The motive was to investigate how an asynchronous event system handled real-time collaboration compared to the more traditional call-and-return client-server architecture.

On a more detailed level, exploratory prototypes can be used to survey different strategies in implementation. In this case minor prototypes can be built to investigate how various architecture or design patterns can help meet the required qualities for an architecture. For example, once the asynchronous event system style of architecture was decided upon in the ABC framework, prototypes were developed in order to decide how the event notification scheme should be implemented in the architecture.

4.2 Experimental Architectural Prototypes

Experimental prototypes are used to measure specific aspects of a proposed architecture, or details hereof. This is done before engaging in the development of the target system based on this architecture. Their main purpose is to establish experimentally if the proposed architecture meets the qualitative requirement. For example, measure if the architecture supports the qualities of performance, availability, or portability. The main users of an experimental prototype are the architects themselves in order to prove that the system, when built according to a certain architecture, can

Table 1. Classification of Architectural Prototypes.

	Architectural Style	Architectural Feature	Architectural Pattern
Exploration	Exploring alternative architectural styles of a system	Refining the style by exploring alternative features	Exploring different implementation strategies
Experimental	Assessing how to meet core qualities	Quantitative measurement of specific qualities	Tuning implementation details to improve on qualities

meet e.g. the performance requirements. Other stakeholders, like the customers, might, however, also be interested in an early proof of conformance to quality requirements. One might argue to what degree it is possible to obtain valid measurements for architectural qualities before the system is built. It might, e.g., be difficult to measure performance and response time in a highly distributed system handling terra-bytes of data, before this system is built and deployed on thousands of hosts. However, we would argue that if a certain architecture quality, like in this case performance, is important, then it is imperative to start building for and evaluating performance at an early phase in the development. For this purpose, an architectural prototype can be built of the intended architecture for experimentation under execution conditions simulating the intended deployment environment.

On the most overall level, an experimental prototype can be build to assess how to meet core requirements to software qualities. For example, in the 'Closed-Loop Process Control' case, the qualities of performance and modifiability were key to the system and hence experimental prototyping were applied to judge how these two qualities would play out against each other. Not surprisingly, there is a trade-off between various architecture qualities and experimental prototyping can be used as method to judge exactly how this trade-off is manifested through real code, and help discover solution of how to balance such trade-offs.

On an intermediate level, an experimental prototype can be used to quantitatively measure how well an architecture fulfils requirements to software qualities. This is typically done by experimenting with features of the chosen architecture. For example, the modifiability and reusability of a system and its components are often correlated to its constituting parts, i.e. components and connectors. Thus, an experimental architectural prototype can be constructed to evaluate how a certain blue-print of components and connectors can meet the requirement for modifiability and reusability. Runtime qualities, like performance and availability, can be gauged by experimenting with control and data issues, such as synchronicity, binding time, and continuity. Such qualities can be subject to quantitative requirements, e.g. time measurement of response time in information systems and processing availability in embedded systems. Experimental prototype can help to continously establish if and how the underlying architecture meets such requirements. For example, a simple experiment in the 'Closed-Loop Process Control' case revealed that the pushed pipe-line architecture could not meet the hard real-time performance requirements to the architecture. Other experimental prototypes focusing on architecture feature might work more like a 'proof-of-concept' in the sense that the prototype demonstrated (proves) that certain software qualities are reached. For example, in the ABC Framework binding time for applications were moved from compile time to runtime and an experiment was conducted to establish how portability and reusability played out under these new conditions.

On a detailed level, an experimental prototype can be built to test how the use of architectural and/or design patterns can help accomplish the overall architectural qualities. Such experimentation often reveals some of the internal tensions embedded in an architecture. On the one hand, the use of architectural and design patterns help meet non-runtime qualities. Patterns help to make code modularized, to create loose couplings, to abstract, etc., that all help accomplish qualities like modifiability, reusability, and integrability. On the other hand, there is a tendency that patterns make it harder to meet runtime qualities, like performance, security, and availability. Levels of indirections simply take time and are, if distributed over network layers, vulnerable to security attacks. Experimental architectural prototypes can help understand and settle this basic tension in the use of patterns. For example, in the 'Close-Loop Process Control' case, the use of the Observer pattern was simply abandoned for performance reasons, and replaced with constant time access of data in shared variables.

5 Architectural Prototypes and the Development Process

As the examples in Section 3 and 4 point out, architectural prototypes may play different roles at different stages of a development process. More generally, we may look at a process framework such as the Unified Process [17] which encompasses a number of concrete development processes and relate architectural prototypes and architectural proto-

typing to this[1]. The Unified Process is a process framework which may be tailored to specific development processes through, among others, modelling the development process. As an extreme, consider the "dX" process created based on (R)UP [2] which is actually an instantiation of eXtreme Programming (XP; [4]) on top of UP.

In the context of the Unified Process, a process defines an assignment of tasks and responsibilities within development.

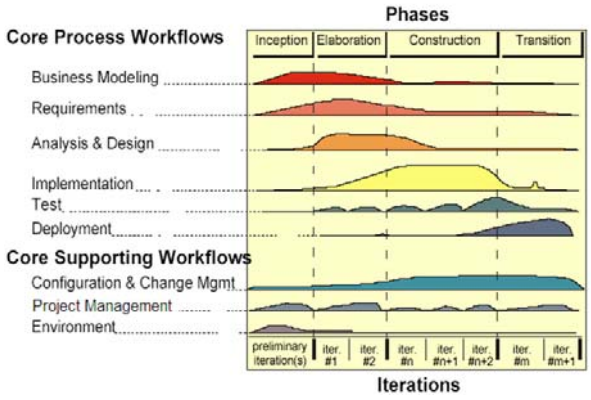

Figure 3. Rational Unified Process (from [21])

Figure 3 shows an overview of an instantiation of UP for a development project in which the dynamic organization of the process (in terms of time) is shown horizontally and the static organization of the process (in terms of workflows) is shown vertically. In general, the UP process operates with four phases in which iterative development is performed:

- *Inception* is concerned with understanding *what to build* by among others establishing a business case and finding and describing major use cases. The Closed-loop Process Control case and the Dragon case were very much concerned with development in this phase. Architectural prototypes built in this phase would focus on major risks and qualities that are essential to whether the system should be built.

- *Elaboration* is concerned with understanding *how to build* what has been identified in the inception phase. This typically includes defining an "executable architectural prototype" [21], "80% complete" use cases, and development plans. These are developed in order to eliminate major risks and uncertainties and on an architectural level, in particular experimental architectural prototypes may play a role. The "architectural prototypes" of RUP are a special kind of architectural prototypes which are built as a skeletal system supporting incremental construction of a system. In contrast, our main emphasis is on *learning* through architectural prototyping, i.e., much of architectural prototyping would come *before* the architectural prototypes of RUP could be defined.

- *Construction* is concerned with incrementally building a functionally complete version of the system (typically a "beta version" of the system). Remaining application features and components are built and integrated into the system. Here, architectural prototypes may be particularly useful for investigating detailed architectural concerns or experimenting with new architectural requirements.

- *Transition* is concerned with building the final version of the product which may be finally shipped to customers. A mature system is installed in the production environment and the final product is created. Although architectural prototypes may not be very useful per se in this phase, most systems go through additional (reduced) phases of inception, elaboration, and construction in which architectural prototyping may be appropriate. The versions of the ABC framework, e.g., has evolved through a number of transition phases and the deployment (albeit in limited form) of the system has been a major driver for the need for architectural prototyping. Moreover, experiments with needed architectural evolution of a deployed system may be necessary, although these may be very high-cost experiments.

Considering the core process and supporting workflows of Figure 3, architectural prototypes may play a role if decisions of architectural significance are to be made in either workflow.

Consider as an example the "requirements" workflow in which actors and use cases are used to elicit, organize, and document required functionality. Here exploratory architectural prototypes may be used to uncover requirements to the architecture (as in the Dragon case in Section 3.2). In particular, this may be useful in extending the vocabulary of the architect and avoiding tunnel vision, e.g., with respect to favoured architectural styles.

During "analysis and design" — in which it is uncovered how the requirements should be realized through implementation — exploratory and experimental prototypes may complement models of architectural designs by providing a grounding of architectural choices. Also, architectural scenarios as created through, e.g., SAAM [18] or ATAM [12], may be used to guide and verify the created

[1] Here we will refer to the Rational Unified Process (RUP; `http://www.ibm.com/software/awdtools/rup/`), which is a widely used variant of the Unified Process, and UP interchangeably

[2] `http://www.objectmentor.com/publications/RUPvsXP.pdf`

architectural prototypes. In particular, specific critical architectural qualities could be investigated as a result of this workflow.

An "architectural prototype" in the sense of RUP is a prerequisite for "Implementation". A sound basis for the RUP architectural prototype may have been provided through exploratory and experimental prototypes. Even though implementation is mainly about realizing functionality, the implementation may provide further validation of experimental architectural prototypes through their realization as full (sub) systems.

The core supporting workflows (test, deployment, configuration and change management, project management, and environment) are workflows in which architectural prototypes may be used to eliminate major concerns with specific architectural quality attribute related to these. Architectural prototypes, may, e.g., be used to explore architectures realizing a desired level of testability such as the ability to verify reliability, functionality, and performance automatically. Another example would be to experiment with buildability through architectural prototypes supporting the environment workflow.

In summary, architectural prototypes may play a role in most phases of development as well in supporting the workflows of the process, although the use of and need for exploratory and experimental prototypes has a different balance throughout development.

6 Conclusion

In this paper we have discussed the concept of *architectural prototypes* and defined it as a set of executables created to investigate architectural qualities related to concerns raised by a system's stakeholders. An architectural prototype is primarily a learning vehicle to explore and experiment with alternative architectural styles, features, and patterns. As opposed to traditional software prototypes, an architectural prototype typically does not provide functionality per se, but addresses non-functional architectural qualities of the target system.

We have proposed a classification of architectural prototypes as *explorative* or *experimental*. Exploratory prototypes help in settling the intrinsic trade-offs between architectural qualities. Experimental prototypes measure certain software qualities at an early stage of development. We have related architectural prototypes to an iterative development process, exemplified by RUP.

We have explored the concept of architectural prototype by discussing how they were used in three distinct cases: (i) investigating how to build a high performance, embedded, distributed control system; (ii) developing prototypes of a globally distributed customer service system supporting containerized shipping; and (iii) a research framework for pervasive computing. In all three cases, we have found architectural prototypes beneficial and cost-effective in demonstrating intrinsic properties of the potential architectures at an early stage of architectural analysis and design, and thereby been essential to reduce risk and demonstrate critical concerns to stakeholders. In one case, the architectural prototype was used as the skeleton system for the final product, and thus served as principal means to communicate architectural design to the development team.

The term 'architectural prototype' has been used before, primarily meaning a 'skeleton system' that is the first version of the target system upon which developers can build the system. We have found this use of the term too restrictive as prototype building serves well in the architecture analysis phase and should not be restricted to the constructive phases.

Our main mission and contribution is to show that the construction of architectural prototypes is a viable and strong technique for architecture analysis and design. Architectural prototypes have characteristics allowing them to identify and analyse architectural concerns and trade-offs that are otherwise difficult to obtain. We thus find the concept important and hope to see more research in the nature of architectural prototypes in order to make it a first-class citizen of any software architects' vocabulary and toolbox.

Acknowledgements

The research presented in this paper has been funded by ISIS Katrinebjerg competence centre, Aarhus, Denmark (`http://www.isis.alexandra.dk`).

References

[1] J. E. Bardram. Activity-Based Support for Mobility and Collaboration in Ubiquitous Computing. In L. Baresi, editor, *Proceedings of the Second International Conference on Ubiquitous Mobile Information and Collaboration Systems*, Lecture Notes in Computer Science, pages 101–115, Riga, Latvia, Sept. 2004. Springer Verlag.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice 2nd Edition*. Addison-Wesley, 2003.

[3] L. Bass and B. John. Supporting usability through software architecture. *IEEE Computer*, 34(10):113–115, 2001.

[4] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture*. John Wiley and Sons, 1996.

[6] CAN. Controller Area Network CAN, an In-Vehicle Serial Communication Protocol. In *SAE Handbook 1992*. SAE Press, 1992.

[7] H. B. Christensen. Using Software Architectures for Designing Distributed Embedded Systems. Technical Report CfPC-2003-PB-55, Center for Pervasive Computing, 2003.

[8] H. B. Christensen and J. E. Bardram. Supporting Human Activities – Exploring Activity-Centered Computing. In G. Borriello and L. E. Holmquist, editors, *Proceedings of Ubicomp 2002: Ubiquitous Computing*, volume 2498 of *Lecture Notes in Computer Science*, pages 107–116, Göteborg, Sweden, Sept. 2002. Springer Verlag.

[9] H. B. Christensen and O. Eriksen. An Architectural Style for Closed-loop Process-Control. Technical Report CfPC-2003-PB-54, Center for Pervasive Computing, 2003.

[10] M. Christensen, A. Crabtree, C. Damm, K. Hansen, O. Madsen, P. Marqvardsen, P. Mogensen, E. Sandvad, L. Sloth, and M. Thomsen. The M.A.D. experience: Multiperspective Application Development in evolutionary prototyping. In E. Jul, editor, *ECOOP'98 – Object-Oriented Programming. Proceedings of the 12th European Conference*, pages 13–40. Springer Verlag, 1998.

[11] M. Christensen, C. Damm, K. Hansen, E. Sandvad, and M. Thomsen. Creation and evolution of software architecture in practice. In *Proceedings of TOOLS Pacific'99*, pages 2–15, 1999.

[12] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2002.

[13] Europe Technologies: EVM-CAN Evaluation Board. http://www.europe-technologies.com/.

[14] C. Floyd. A systematic look at prototyping. In R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllighoven, editors, *Approaches to Prototyping*, pages 1–18. Springer Verlag, 1984.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reuseable Object-Oriented Software*. Addison-Wesley, 1994.

[16] J. Greenbaum and M. Kyng. *Design at Work: Cooperative Design of Computer Systems*. Lawrence Erlbaum Associates, 1991.

[17] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[18] R. Kazman, L. Bass, and G. Abowd. SAAM: A Method for Analyzing the Properties of Software Architectures. In *Proceedings of the Sixteenth International Conference on Software Engineering*. ACM Press, 1994.

[19] F. Kensing and J. Simonsen. Using ethnography in contextual design. *Communications of the ACM*, 40(7):82–88, 1997.

[20] OMG. Unified Modeling Language specification 1.5. Technical Report formal/2003-03-01, Object Management Group, 2003.

[21] Rational Software. Rational Unified Process: Best Practices for Software Development Teams. `http://www.rational.com/media/whitepapers/rup_bestpractices.pdf`, 1998.

[22] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-oriented Software Architecture – Patterns for Concurrent and Networked Objects*, volume 2. John Wiley and Sons, 2000.

[23] Software Engineering Standards Committee. IEEE recommended practice for architectural description of software-intensive systems. Technical Report IEEE Std 1471-2000, IEEE Computer Society, 2000.