Towards an Operational Framework for Architectural Prototyping

Henrik Bærbak Christensen
Department of Computer Science, University of Aarhus
Aabogade 34, 8200 Aarhus N, Denmark
hbc@daimi.au.dk

Abstract

In this paper we present a case study of using architectural prototyping to explore an architectural design space. The case study will be treated as a data point representing one way of using architectural prototypes. Based on an analysis of the processes involved we present a first, tentative, framework that describes key concepts and their relationships.

1 Introduction

Bardram et al. [5] has defined architectural prototyping and an architectural prototype as:

> 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.

They outline a number of characteristics of architectural prototyping: It is a viable and cost-efficient technique for exploring an architectural design space and for addressing issues regarding quality attributes for the target system. Architectural prototypes implement only an architectural skeleton without business-oriented and user-oriented behavior making them cost-efficient to produce. Architectural prototypes typically address architectural risks, and finally, they address issues of knowledge transfer and architectural conformance.

The contribution of the present paper is to describe a concrete case study of using architectural prototyping and by abstracting over this to provide a first tentative operational framework for describing architectural prototyping processes. Our focus has been on the "process in the small" i.e. on how a competent architect in a small set of sessions has produced a number of architectural prototypes for a number of different purposes and how they are interrelated, feed each other, and each contribute to the learning process.

2 Case Study: ABC Programming Model

2.1 Activity Based Computing

Activity based and task oriented computing has been drawing research attention for a number of years [4, 7, 3, 2]. At University of Aarhus, the ABC project [1] has been exploring aspects of the concept. The concrete case study deals with a specific problem that we have been working on lately. Activity based computing's central theme is the ability to group all the computational services used by a user for a certain activity/task into a first-class object, the *activity*. The computing infrastructure allows the user to *suspend* the activity and later *resume* it in its entirety on another computational device i.e. restart all services and force then into the same state as when the activity was suspended.

2.2 Architectural Design Problem

A central architectural challenge is *state management of services*. So far the ABC infrastructure basically assumes that ABC enabled services implement an interface, `Statefull`, containing only two methods:

```
interface : Statefull {
   public String getState();
   public void setState(String aml);
}
```

where the strings are the service's state information described in a special XML format called AML: Activity Markup Language.

This makes state management simple from the ABC runtime's point of view: invoking the ABC middleware's suspend method involves invoking `getState` on all registered services to build a large AML data structure and store it in a central server. The resume is just the opposite: fetch the AML, instantiate appropriate services, and invoke their respective `setState` methods with the appropriate part of the AML.

However, it provides very little help for the service developer that has to do all the tedious marshalling and unmar-

shalling of AML, the translation from and to the service's internal run-time object structure, and potentially updating the GUI—a process that has turned out to be both tedious and prone to errors. Thus, one branch of research in the ABC project looks into improved the programming model.

2.3 A Proposal for an Improved Programming Model

One proposal that is currently investigated is using the meta-data annotation techniques of modern languages like C# and Java 1.5. The idea is to annotate fields that must be part of the state information and thereby automate the marshalling and unmarshalling as much as possible. For example, a text service that needs to reestablish textual contents and the cursor position when resumed may mark the x, y, and $theText$ fields:

```
class TextEditorService : Statefull
{
  [StateAttribute("contents")]
    private String theText;
  [StateAttribute("x-coord")]
    private int x;
  [StateAttribute("y-coord")]
    private int y;
  ...
```

The interpretation is that the string and two integers are marked as fields that must be handled by the state management facilities of ABC with the given XML tag names, and thereby provide at least a naive implementation of `getState` and `setState`, leading to an improved programming model for ABC service developers.

3 Case Study

One of architectural prototyping's main aspects is as a vehicle for learning and exploring an architectural design space. In our case study we have followed an architect in his process of exploring aspects of the proposed programming model for state handling in ABC. The architect is a deep understanding of the ABC architecture and is an expert Java programmer. He has, however, no experience with C# and only an abstract understanding of C#'s annotation and reflection facilities. Thus the architectural challenge facing him is three fold: exploring how field annotations may be used to improve the quality attributes of modifiability and reliability; climbing the learning curves of concrete technology; and assessing the technology's support for the architectural vision of an improved programming model.

The discussion to follow will be in the form of a short essay describing a small set of architectural prototyping sessions. We will focus on the actual prototypes constructed, the problems they address, and how the relate. Figure 1 sketch the architectural prototypes and their relationships made during the sessions.

Henceforth we will use the short form "AP" to denote "architectural prototype".

3.1 Architectural Prototypes in Action

The present release 4.2 of ABC comprises about 164 classes and 37.500 LOC in C# forming the ABC middleware on the client side and about 26 classes / 3.500 LOC in Java forming the server side. While the programming model issues only concerns the client side, there is still a lot of code to consider. Another problematic aspect with regards to a fast exploratory phase is that testing the client's ability to reliably resume and suspend is hinged on manually setting up applications and comparing their state visually before and after a suspend/resume, and the need to ensure that the server does not cache any old state information that may interfere with ones observations. Moreover the design exploration cycle is slowed down considerable because a networked system including a running server must be set up for every session.

Iteration 1. Objectives: Harvesting; C# Learning. It was therefore decided to extract architecturally relevant code for the problem at hand into an AP, TestSetGetABC, that is then used for experimentation instead of the full system. This extraction process we will refer to as *harvesting*, inspired by a similar concept used in software reuse.

An important aspect in the harvesting process is to extract the minimal executable that faithfully reflects the original architecture with respect to the problem at hand. The stated architectural problem may be rephrased more technically as:

> Minimize the manually written code required to transfer a service's state into XML and vice versa.

This specific focus allows a large set of aspects in ABC to be either completely ignored or abstracted:

- No need for end-user functionality. We simply want to test the transformations between XML and run-time object state, and unit tests or visual inspection in a shell is sufficient.
- No need for a server and a network. Our problem is about transformations in the client side's services, once they can produce proper AML, we are done as the ABC already supports all the steps involved in storage and server communication.
- Service abstraction. We will be experimenting with a new programming model for implementing services, thus we take existing ABC services only as inspiration.

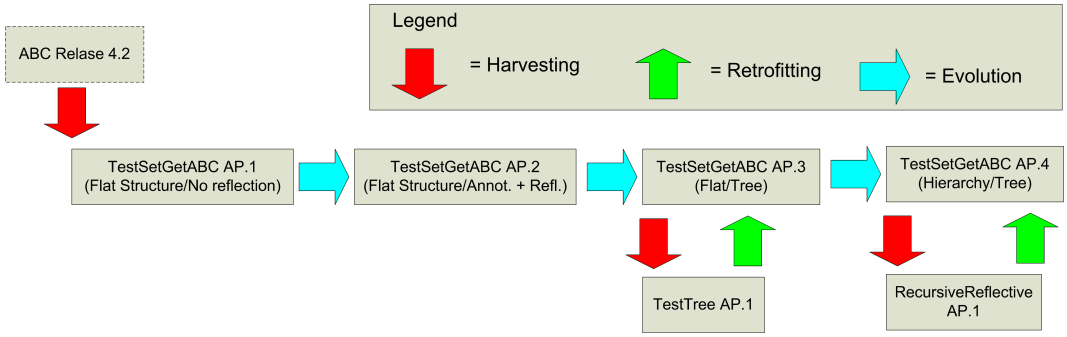

Figure 1. Case study prototypes

There are some requirements though. The architectural prototype must be in C# to make realistic experiments with the concrete annotation and reflection technology. The service abstractions must be as defined as complex as possible to stress test the ideas for the programming model; in essence making them more complex than any of the existing ABC services; but only with respect to the complexity of the run-time object structure.

An iterative and incremental process is natural: make simple state-management work first, and then progress through more and more complex object-structures.

Based upon the observations above a very simple first AP was written. It contained:

- The `Statefull` interface
- An abstraction, `ABCClientMiddleware`, over the real ABC client side middleware. It is simplified as much as possible while being architecturally consistent with the real component. The abstraction simply maintain a list of running services and contain the `suspend` and `resume` methods that mimics ABC's ability to suspend and resume activities.
- A representative service, `TextService`, implementing the `Statefull` interface. The choice of a text service is arbitrary but chosen as a text edtior service is already present in ABC. Again only architecturally significant properties are haversted, thus the text service has no GUI and has a flat run-time object structure containing a text contents and a cursor position. It contains only a `change` method simulating changes to the text service's state, and a `ToString` method which is the closest the service gets to having a user interface.
- A driver/main that instantiates a TextService, manipulates its state, invokes the ABCClientMiddleware's suspend, and thereafter its resume. Visual inspection in the shell is used to determine that the text service's state is properly handled as well as output of the resulting XML code.

In the first iteration, no annotation is made of the text service's fields and the set and get methods are written manually as in the original ABC.

Other measures were taken as well to lower the AP development cost. The architect was used to the Eclipse environment but not Visual C#. He therefore decided to avoid the latter's learning curve (and its habit of inserting large amounts of comments) and use a code-editor and a command-line compiler.

The working 1st milestone AP.1 consists of 175 LOC, took about one man day, and met the objectives: The ABC architecture was harvested in a minimal configuration that allows experimentation while being architecturally consistent with the original; and the architect had set up a simple C# environment and got exposed to basic C# class library functionality.

Iteration 2. Objectives: Buildability—Learning annotation and reflection. The harvested architecture now serves for the first experiments with C#'s annotation language mechanism. The text editor's attributes are now decorated as described in section 2.3. This requires another class to be added to the TestSetGetABC AP:

- The `StateAttribute` class inheriting C#'s `System.Attribute`. Objects of this class represent the C# run-time information of the fields decorated with the StateAttribute annotation.

and experiments are started with automatic state management.

During this process several milestones are made, each adding an increment of functionality and exploring annotation, reflection, and class library support (for instance XML parsing) and their use for the defined goal. It culminates in an AP that defines an delegation-based architecture where services delegate state management to a special component tailored for the service. The ABC middleware provides this

component to the service upon registration. This AP.2 is less than 350 LOC, and took about one additional man day.

Iteration 3. Objectives: Buildability—Hierarchical object-structure. Next it must be explored how a service with hierarchical object-structure can be handled. The service's structure is refined into:

```
class MyPoint {
  [StateAttribute("int", "x")]
  public int x;
  [StateAttribute("int", "y")]
  public int y;
  ...
}
class TextEditorService : Statefull {
  [StateAttribute("text","contents")]
    private String theText;
  [StateAttribute("ref", "cursor")]
  private MyPoint cursor;
  ...
}
```

and of course AP.2's state management fails as it does not handle recursive descend into the object structure and has no provision for generating hierarchical XML.

From the AP process point of view, this iteration brings several architectural challenges. First, the state-management component needs to understand its associated service's object-structure and this understanding is best coded in a tree. However, C#'s standard library has no tree data structure. Second, the AP only use reflection to set primitive types, now it must be able to set values in a deeper object structure (with the possibility of having null references).

The first challenge is addressed by finding a public domain tree data structure in C# on the net. Thus a new learning curve is facing the architect as the data structure has a large set of manipulation methods, scarse documentation, and its characteristics suchs as reliability and suitability is unknown.

Iteration 3.1. Objectives: Buildability—Tree data structure. This challenge is assessed by a new architectural prototype: TestTree. This isolated prototype addresses again the buildability quality: learning the API and assessing the component's viability for the problem at hand. The reason for addressing this problem with a separate AP instead of learning within the TestSetGetABC AP is similar in argumentation: the AP is clearly focused on a single goal and there is simply less code to disturb.

Still, a sort of harvesting takes place between the TestSetGetABC and the TestTree AP as the tree data structure built and experimented with in TestTree is identical to the one the state management component must handle for the text service.

The process ends in a milestone—the TestTree AP.1. The conclusions drawn is that the tree data structure is usable but it lacked support for a deep first traversal and its build-in XML generation support is not flexible enough for generating AML.

Iteration 3.2. Objectives: Retrofitting. Iteration 3.1 made the architect confident in the tree data structure, therefore the next step is to replace the existing flat data structure used in the state management component with the tree. As the TestTree AP.1 contains code fragments that build the type of tree suitable for the TestSetGetABC AP, these fragments can be transferred back into the TestSetGetABC AP.2 and refactored to work there. We denote this process *retrofitting*, again inspired by a term from the software reuse community.

To ensure an incremental process, the TestSetGetABC's TextService was first reverted back into a flat object-structure—and made to work as at the end of iteration 2. In essence, it was a refactoring step: ensure same behavior but with an improved design. Again this was considered a milestone: TestSetGetABC AP.3.

Iteration 4. Objectives: Hierarchical object graph. Next, the complex object graph was introduced into TextService again, and work began on the state management component's handling of first getState and next setState. It quickly turned out that reflection over object references in C# during a recursive descend of both the meta information in the tree structure as well as in the real object graph was not trivial. This problem was studied in a new architectural prototype.

Iteration 4.1. Objectives: Buildability—Recursive reflection. The RecursiveReflective AP harvested the TextEditor service code from the TestSetGetABC AP.3 and experimented with a recursive descent into the object graph based upon C# reflection. This relatively simple prototype was simply a vehicle for finding a viable way to code this behavior. The result was RecursiveReflective AP.1.

Iteration 4.2. Objectives: Retrofitting. The lessons learned from the RecursiveReflective AP.1 could now be retrofitted into the TestSetGetABC AP.3. During this retrofitting no actual code was moved between prototypes; instead it was only the structure of the recursive descent and the knowledge of the proper C# reflection primitives to use that was retrofitted. Eventually this work will lead to a new milestone, the TestSetGetABC AP.4.

3.2 Summary

We will break from the process here as the purpose is not to present a new programming model for ABC but just to present a concrete case of using architectural prototypes to explore and learn an architectural design space.

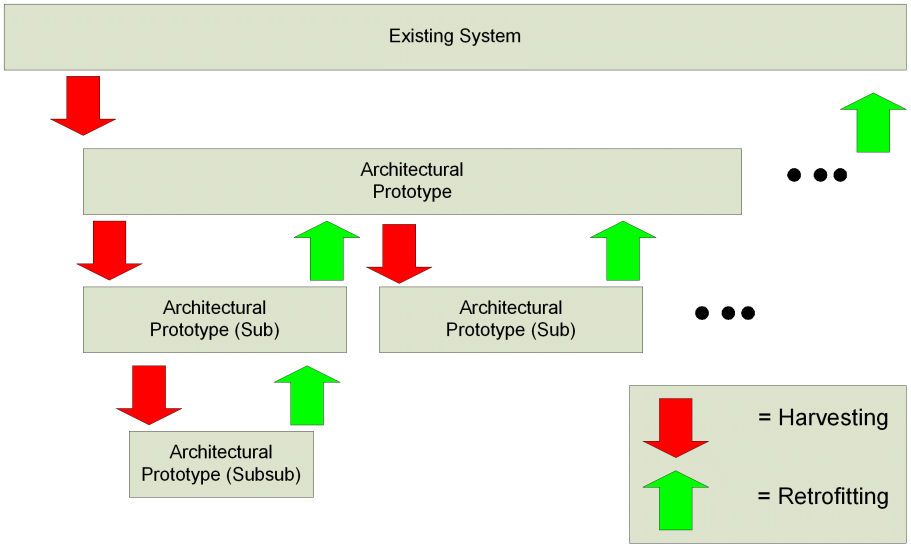

Figure 2. Architectural Prototype Processes

4 A Framework for Architectural Prototyping

The case study above has been presented in somewhat detail to give the readers concrete insight in the process of using architectural prototypes, and as we find this small example representative of the process and the subprocesses involved. We will use this case study as well as our experience to present a first attempt at classifying and analyzing an operational framework for how architectural prototyping is carried out.

Architectural prototype can be used in most phases of a software development project as argued by Bardram [5]. Their case studies are mostly about defining architectures from scratch, but in our case study the onset is an existing architecture that must be refactored.

First, *architectural prototyping is defined as the process of designing, building, and evaluating architectural prototypes.* A prototyping process starts with some formulated architectural challenge and ends by establishing validated proposals to the challenge. Reflecting upon the presented case study we conclude that the prototyping process is *recursive* where a complex problem set is divided and conquered by formulating simpler subproblems, each to be solved using the architectural prototyping process.

This recursive application of the process naturally lead to a *hierarchy of APs* where identified architectural challenges in a "super AP" is explored in a set of "sub APs". Whether a subproblem warrants building a special AP to investigate it is of course a matter of cost-benefit. The cost is in terms of defining a new AP (new program/project), in terms of the amount of code (and potential support code) that must be transferred to the AP, as well as the cost of making the AP reflect the real setting and problem in the superAP. The benefit is "sand-boxing" i.e. no code unrelated to the problem obscures the focus. Another benefit is the very existence of the subAP as a self-contained program. Thus the code and the lessons learned are available in the design's later phases—it is not buried and eroded inside a much larger AP or in the system itself.

The hierarchy of APs is connected by relationships. We identify two relations:

- **Harvesting.** Extraction of architecturally significant code/behavior or knowledge from a super AP to a sub AP. The borderline between what is significant and what not is defined by the architectural problem to be investigated.

- **Retrofitting.** Insertion of architecturally significant code/behavior or knowledge from a sub AP into a super AP. A sub AP has clarified some problem statement and the proposal that is evaluated to be correct is transferred to the super AP. Retrofitting comes in two flavors. In the case of *code retrofitting* it is physical source code/design artifacts from one AP that is transferred and refactored into another AP. In the case of *knowledge retrofitting* it is only the new learned knowledge of the architect that is transferred in the form of a changed design or refactored code.

Both "relations" are of course really processes where code is copied, modified, and refactored, or knowledge applied in the target AP. Care must of course be taken that the simplifications and adoptions made in the processes do not invalidate the architectural conclusions made in the APs.

The terms are borrowed from the reuse community, see e.g. [8, 6], where the process of extracting reusable assets from a software product is denoted harvesting, and retrofitting denotes refactoring the product to use the harvested, reusable, assets.

Note that at the outermost level in the hierarchy of processes, the real product system may serve the role as AP. In the case study, iteration 1, was a harvesting from the real ABC, and if the envisioned state management support is evaluated positive, it will eventually be retrofitted into ABC.

5 Conclusion

In this paper, we have observed a couple of short sessions in which an experienced architect used architectural prototyping for exploring a design space. The exploration was both in terms of development of new architectural means as well as for learning concrete technology and assessing their ability to support the architectural vision. The problem studied was aspects of improving the programming model for developing services for an existing activity based infrastructure, ABC.

Based upon our general knowledge of architectural prototyping and the case study in particular, we propose a first attempt at defining a framework for the process of doing architectural prototyping. We have identified special processes, denoted *harvesting* and *retrofitting*, and described their relation to the individual crafted architectural prototypes as well as their purpose and process contents.

The continued effort to further refine this process framework is driven by two main objectives. First, it serves to provide a common terminology and conceptual framework for architectural prototyping which is important from a research perspective, Second, it serves as an operational framework that defines and exemplifies concrete steps to perform for practitioners in their quest to tackle architectural and learning challenges.

We hope that we will get the opportunity to discuss the process framework at a WICSA working session in order to evaluate it and define directions of further research.

Acknowledgements

We would like to thank Jakob Eyvind Bardram and Klaus Marius Hansen for the inspiring joint work on ABC and architectural prototyping. This research was partly funded by the ABC project that is funded by the Danish Research Council under the NABIIT program.

References

[1] ABC – Activity Based Computing. http://www.cfpc.dk/abc.

[2] J. E. Bardram, H. B. Christensen, D. Garlan, and J. Sousa, editors. *Proceedings of First International Workshop on Computer Support for Human Tasks and Activities*, Vienna, Austria, 2004. Pervasive 2004.

[3] H. B. Christensen and J. E. Bardram. Supporting Human Activities — Exploring Activity-Centered Computing. In *Proceedings of Fourth International Conference on Ubiquitous Computing, UbiComp 2002*, Göteborg, Sweden, 2002.

[4] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project Aura: Towards Distraction-Free Pervasive Computing. *IEEE Pervasive Computing*, 1(2):22–31, 2002.

[5] Jakob E. Bardram, Henrik Bærbak Christensen, and Klaus Marius Hansen. Architectural Prototyping: An Approach for Grounding Architectural Design. In *Proceedings of Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA4)*, pages 15–24, Oslo, Norway, June 2004.

[6] T. Ravichandran and M. A. Rothenberger. Software reuse strategies and component markets. *Commun. ACM*, 46(8):109–114, 2003.

[7] J. P. Sousa and D. Garlan. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. In *WICSA*, 2002.

[8] Ted J. Biggerstaff. Design Recovery for Maintenance and Reuse . *IEEE Computer*, 22(7):36–49, 1989.