Software Architecture Education Whitepaper

George Fairbanks
Rhino Research

Draft: Version 0.3
16 November 2009

1 Introduction

Many organizations struggle to choose the most appropriate courses to advance their people. They struggle because, while it is relatively easy to describe their desired end state, the path to get to it is unclear.

Software architecture education is particularly tricky. If you take a class on Java or Struts, you do so because you do not yet know that language or framework. But everyone has some ability to design programs already, so it is not clear what benefit will come from a general class on architecture and design. If you are considering an architecture class, you suspect that your team would benefit from software architecture and design education, but are having a hard time putting your finger on exactly what is missing. As a consequence, it's hard to be sure if any particular course is what the team needs.

This whitepaper describes a structured approach to thinking about your architecture and design needs. This approach, in brief, consists of the following activities:

1. Grouping people into personas
2. Identifying problems and connect them to personas
3. Connecting education offerings to problems
4. Evaluating or developing courses

At the end of this, you should have a better idea of your organization's needs and therefore be in a good position to decide what kind of education options will be most helpful.

This whitepaper makes three contributions. First, the overall structured approach that aids decision making. Second, the use of personas instead of the obvious choice, job titles/roles. And third, the sets of prototypical personas, problems, and education offerings.

2 Personas

Each person in your organization is unique. However, to get a handle on the problems of education, it is useful to identify several stereotypes that group similar people according to how they will benefit from and respond to education. This technique is borrowed from the User Centered Design community. We call these stereotypes *personas.*

We could simply use existing job titles, but not every person with the title *developer* or *architect* will need or respond to education in the same ways. This paper instead defines personas based on the breadth of a person's view. Some people are code-focused craftsmen; some design single applications; others look across applications to make

the system of systems work most effectively. Notice that each job differs in its needs to balance the understanding of details versus abstractions.

It is useful to seed the discussion with a few common personas that are seen across the industry. You will need to refine these and add additional personas that are specific to your organization.

Module developer. Module developers focus on writing good code. They pride themselves on mastery of programming languages and environments, including hard-won knowledge about how various technologies work in practice. The best module developers have an enormous knowledge and ability to master details which enables them to avoid using abstractions to simplify problems. They may disdain abstractions and believe that the best solutions are found by digging in rather than abstracting away details.

System developer. System developers focus on building a good system, which requires both good code and an understanding of how that system fits in with others. System developers look for opportunities to use abstractions but may use their own notations and concepts rather than standardized ones. System developers know architectural styles from their domain. System developers are comfortable with architectural ideas like quality attributes and standardized connectors when others mention them, but they do not think about them daily.

System designer/architect. System designers/architects focus on how a system accomplishes its functional and quality attribute requirements. They dwell less on how individual modules are constructed so long as the modules are reasonable and fit into the system's architecture. They concern themselves with the suitability of the design and architecture to the functional and quality attribute requirements. They seek to solve or avoid problems by appropriate design decisions.

Enterprise designer/architect. Enterprise designers/architects work like movie producers to exert influence indirectly since they do not write code nor are they responsible for any single application. They look for design and architecture principles that will guide the organizations systems to work together smoothly and to accomplish functional and quality attribute requirements today and in the future.

As you can see, the descriptions of these personas describe how they view and use architecture. Most organizations want more people who can function in the system designer/architect persona since these skills have great influence on the success of individual projects and there always seem to be more projects than top designer/architects.

3 Problems

You need to decide how the state of your organization's design / architecture skills could be causing trouble or inefficiency. It is surprisingly difficult to disentangle how general design / architecture skills contribute to success or failure at building good software.

This section identifies some common problems then connects those problems back to the personas. For example, we are quite worried about transforming module developers into effective designers / architects, but this is not a worry for existing designers / architects.

3.1 Common problems

Here are some common problems seen in many organizations:

Slow advancement. Slow progression to system designer/architecture persona. Not all module developers will progress, and those that do progress slower than we would prefer.

Irreproducible factors. Strength of design based on non-reproducible factors (e.g., individual designers). It has long been reported that putting "ringers" (also called "ten-x-ers" in reference to their being ten timess more productive) on projects will help them succeed.

Low best practice use. Industry best practices not consistently applied. As engineers, we understand that software development entails risks, but we are dismayed when we look back at failed projects and find preventable problems.

Communication friction. Communication inefficiency / friction because of differing vocabulary and concepts. Different teams and pockets within the organization have their own names and abstractions for similar ideas.

Focus on functionality. Focus on functionality, failure to consider quality attributes. Quality attributes such as performance, security, usability, modifiability are decided implicitly rather than deliberately engineered into the software.

3.2 Problems and personas

Not every persona is affected by the problems equally. For example, it is OK when two module developers fail to communicate using architectural terms, but not OK when two system architects fail to do so.

You need to understand which personas are subject to which problems so that you can choose education offerings appropriate for each persona. The following table is an example based on the above personas and problems, and should be calibrated and extended with your organization's details.

	Module Developer	System Developer	System designer / architect	Enterprise designer / architect	...
Slow advancement	High	High	Med	Low	
Irreproducible factors	Med	High	High	Med	
Low best practice use	Med	High	Med	Med	
Communication friction	Med	High	High	High	
Focus on functionality	High	Med	Low	Low	
...					

4 Education offerings

At this point you should have an idea of the kinds of people in your organization, the problems in your organization, and which people experience which problems. What you need to know next is what can be ordered from the menu – that is, what software architecture can offer. What follows is a list of items commonly available in architecture education courses.

4.1 Declarative design knowledge

In 1903, Georges Escoffier published Le Guide Culinaire on the art of French cooking. It was targeted at educating the younger generation of cooks, so it organized and explained what the older cooks had learned and inferred for themselves. It was remarkable because it stated declaratively what the best chefs were doing so that others could do it too.

Every company today wishes that it could simply clone its best designers and architects. While we cannot do that, we can do what Escoffier did, and help them express their design and architectural knowledge declaratively so that others can build like the best do.

Experienced designers and architects may design good systems, but often they cannot say declaratively how they do it. If you have ever tried to learn how your grandmother cooks, perhaps you may have already experienced this problem. She adds an ingredient "until the batter looks right" – which is perfectly repeatable for her, but you wish she could state her knowledge declaratively, perhaps in tablespoons.

Interestingly, this is the opposite of the college-hire problem. New employees straight from college can usually declaratively state facts like "a good module decomposition exhibits high cohesion but low coupling," but they have yet to learn how to apply this knowledge on projects.

4.2 Architecture models

Software architecture uses models to handle the scale of large systems. Humans are not built to grovel through the details of millions of lines of code, so we use models of that code to reason abstractly. Software architecture also uses models to promote important details. When we do story problems like "two trains 50 miles apart are traveling at 10 mph; when do they meet?" we omit many details, like the color of the trains, because those details are a distraction when answering the question. Similarly, when we want to reason about the throughput or security of a system, we want to elide details that distract us.

Some models are created ad hoc, but for standard tasks we already have standard modeling concepts and terminology. Using standard models saves time and aids in communication. When developers use models, their desire is to amplify their reasoning ability. A well-constructed model makes reasoning about hard problems easier.

The use of standard models also establishes a reasoning framework. On software projects, new facts and ideas are always popping up. Having a reasoning framework means that developers can immediately make sense of them – is it a domain fact, an architectural style to consider, a quality attribute tradeoff, or something else?

When a coach and a rookie watch the same game, their experience of it differs greatly because the coach has a sophisticated reasoning framework that allows him to make sense of what he is seeing. He can, for example, categorize it as a success of an offensive strategy rather than as a bunch of guys running around on the field. He can see more because he has a reasoning framework. In the domain of physics, the idea of analyzing problems using free body diagrams is a reasoning framework. In software architecture, the reasoning framework includes a canonical stack of models (the domain, the design, and the code) plus relationships between models (views, refinement, and designation).

4.3 Patterns and styles

Architectural patterns and styles help developers by describing and naming solutions to recurring problems. Examples of architectural styles include client-server, pipe and filter, and map-reduce. Learning about patterns and styles is an easy win educationally because they can be expressed as factual reference material rather than nuanced design techniques, and because developers have likely seen many of them before and just need to associate a name with what they already know.

4.4 Design principles

Some software design principles for modules are well established: minimize coupling, maximize cohesion, encapsulate designs, separate concerns. To these, newer principles can be added, such as creating a *story at many levels* and that *form liberates*. All of the principles should be refreshed and revisited in the context of the standard architectural models and quality attributes, because they are often considered only in the context of modules (not components or allocation nodes) and assume that code modifiability is most important, when it is just one of many quality attributes that may trade off against others.

4.5 Design techniques

Separate from the knowledge of standard models is the set of techniques used to build effective models. For example, the UML community has honed techniques for building information and behavior models; the architecture community has the same for component and connector models. Furthermore, developers can choose to apply styles or patterns to enable quality attributes (architecture-centric design) or even create runtime infrastructure to ensure qualities (architecture hoisting).

4.6 Analysis techniques

Once a design exists, developers want to analyze it to decide if it meets the functional and quality attribute requirements. Many quality attributes have specific analyses that can be performed, such as throughput, latency, and security analysis. Broader analyses exist including the Architectural Tradeoff and Analysis Method (ATAM) from the SEI. Architectural checklists help remind and reinforce quality checks. Architecture reviews enable more experienced developers to provide feedback on designs.

4.7 Processes

Several processes have been documented to guide the use of architecture on projects. The SEI has contributed the Architectural Tradeoff and Analysis Method (ATAM), the Quality Attribute Workshop (QAW), and Attribute Driven Design (ADD) processes. The Risk-Centric Model of software architecture can be used to decide how much architecture to do on projects.

4.8 Case studies

Case studies are descriptions of real or hypothetical examples and how architectural knowledge and techniques are applied to solve problems. For architecture, case studies can be used to demonstrate the implications of different design decisions or explain how forces outside the software make some architectures suitable or unsuitable.

4.9 Local organization knowledge

Every organization has in-house technology, terminology, and patterns. Off-the-shelf courses cannot contain this knowledge without being customized. Most of this knowledge represents specializations of the standard body of knowledge, and should be presented as such. For example, an organization may use proprietary hardware configurations, but these can be modeled using the standard models and described as architectural styles in the allocation viewtype.

5 Evaluation

Now that we have identified which problems each persona struggles with and we have listed some education offerings, we can evaluate which education offerings are most effective for each persona. We start by deciding how well each education offering addresses each problem. Since we have already related personas and problems (in Section 3.2), we can now connect personas to education offerings.

5.1 Problems and offerings

Given the problems identified earlier, we need to choose education offerings that may help each problem. Each education offering addresses some problems better than others. For example, teaching a process works against the problem that best practices are not used (assuming the process is indeed a best practice), but processes do little to address communication friction between developers. The following table shows which problems are helped by which education offerings.

	Declarative Design Knowledge	Architecture Models	Patterns and Styles	Design Principles	Design Techniques	Analysis Techniques	Processes	Case Studies	Local Knowledge
Slow advancement	High	Med	Med	Med	High	High	Low	High	Med
Irreproducible factors	High	Med	Med	High	High	High	High	Med	High
Low best practice use	Med	Med	High	Med	Med	Med	High	Med	High
Communication friction	Med	High	High	Med	Med	Med	Low	Low	Med
Focus on functionality	Low	Med	High	High	Med	Low	Med	Low	Low
...									

5.2 Personas and offerings

Now that we have decided how well each education offering addresses each problem, we can mechanically trace this back to the list of personas. In the table below, the fist two columns are taken from the table in Section 3.2. Based on the problems and offerings table immediately above, we can mechanically populate the third column.

For example, the module developer persona struggles with focusing on functionality, which is a problem that is addressed by education about design principles and patterns and styles. The problems faced by the system developer and system designer/architect personas map to all, or essentially all, of the education offerings.

	Highest priority problems	Most effective education
Module developer	Slow advancement	Declarative design knowledge Design techniques Analysis techniques Case studies
	Focus on functionality	Patterns and styles Design principles
System developer	Slow advancement Irreproducible factors	All
System designer/architect	Irreproducible factors Communication friction	All
Enterprise designer/architect	Communication friction	Architecture models Patterns and styles

6 Conclusion

The big-picture outcome of this exercise is what we would expect: different personas will benefit from different kinds of software architecture education. Personas are a kind of caricature, so people in your organization will not line up exactly, but some generalities should endure. For example, many developers are head-down in the code and strive to master more details rather than use abstractions.

It is possible to tune the results of this exercise to closer align with your organization. There are three primary ways to tune the exercise: the list of personas, the list of problems, and the list of education offerings. Additionally, there are subjective evaluations that can be tuned, such as which education offerings are effective at solving which problems.

Being fluent in architecture technologies and skills will broaden the way that developers understand code and systems, enabling them to work on larger systems and be more effective. Organizations seem to always have a shortage of such developers, so it is beneficial to choose the kind of education that will be most effective.

