The 4x6 Tiered Architecture Method: An Approach to the Design of Enterprise Solutions

Ethan Hadar[1], Irit Hadar[2], Gabriel M. Silberman[1], and John J. Harrison Jr.[1]

[1] CTO Office
CA Technologies
{ethan.hadar,gabriel.silberman,jay.harrison}@ca.com
[2] Department of Information Systems
University of Haifa
{hadari}@is.haifa.ac.il

Abstract. Enterprise architecture software design is all about composing applications to assemble value-added solutions rather than standalone products. Yet, each product and technology may have been designed and developed separately because of software engineering practices, management control over the deliverables, or technology acquisitions. To promote efficient assembly, solutions must be architected in a similar style, adhering to fundamental design principles while leveraging capabilities available in modern environments and relevant platforms. Furthermore, business agility and cost requirements dictate the identification of common capabilities and their development as reusable components across products and solutions. The 4x6 Tiered Architecture Method presented in this paper imposes a structured design, in terms of steps to follow, structure and documentation, for the logical view of an enterprise solution. Application of the 4x6 method to the analysis of an enterprise solution yields a six-tiered architecture structure and an abstract architecture specification. This specification expresses the various components, dependencies and design patterns using a graph-based data model (or "architecture catalog") and blueprint, the latter expressed as both a diagram and XML document. The 4x6 Method has been applied in practice; this experience indicates that this method results in higher quality architecture and requires lower effort for both constructing and reviewing the architecture and its documentation.

Keywords: Design Tools and Techniques; Software Architectures; Domain-specific architectures; Patterns.

1 Introduction

The architecture design for an enterprise solution involves a number of challenging decisions, including the expected transaction load, response times, volume of users, number of integrated systems, options to deploy as a stand-alone, on-premise, or as a Software-as-a-Service (SaaS) solution. To address these issues and produce a good design, architects employ design patterns [1] and, in many instances, reuse existing components and integrated services to solve known problems with known solutions

M. Bajec and J. Eder (Eds.): CAiSE 2012 Workshops, LNBIP 112, pp. 180–191, 2012.
© Springer-Verlag Berlin Heidelberg 2012

[2]. The design intent is to invest the most (new) effort while reusing existing assets as much as possible. To further enable the combination of existing solutions, and their replacement as future technologies emerge, requires us to characterization them in terms of overall capabilities and non-functional characteristics from the customer perspective [3].

In short, as long as the product or solution is not an isolated instance, most design activities will deal with adding capabilities to existing modules, integrations with external technologies and services, and/or refactoring and evolution of the architecture structure. The challenge for the solution architect is to provide a quality design for separate structures resulting in easy to understand, out-of-the-box components, with a set of (estimated) characteristics for the combined result.

To tackle these challenges, we developed a model-driven architecture method, called the *CA Four Architecture (C4A)*, an extension of the C3A approach described in [4]. As in other Model Driven Architecture (MDA) approaches, multiple views over a single model enable the capture of design intent and multi-dimensional module characteristics, while documenting architecture decisions. The proposed methodology uses four diagrams, representing four views, iteratively developed and refined through a series of analysis, design and delivery steps, as shown in Figure 1.

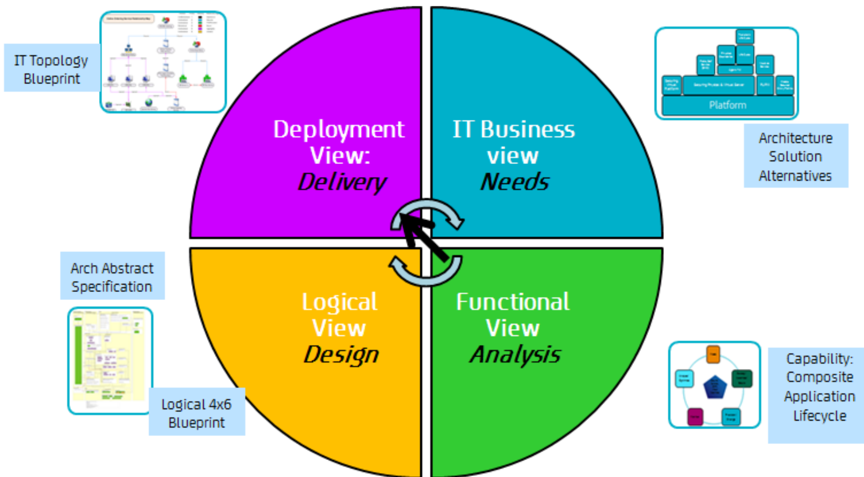

Fig. 1. C4A Architecture Views and Process

Although similar to the 4+1 view model by Kruchten [1], C4A employs fewer diagrams than the eight suggested by Kruchten and addresses a larger portion of the full software development life cycle, including a target reference architecture and evolution plan. C4A also integrates the design process via systematic analysis and design flow, which in the 4+1 approach is managed through integration with the Rational Unified Process [1].

As depicted in Figure 1, the C4A views include 1) an *IT Business View*, focused on the business rationale and “go-to-market” needs; 2) a *Functional View* to capture the

"jobs-to-be-done" (JTBD) [5] capabilities using the customer's own business language and taxonomy; 3) a *Logical View* of the architecture software components as the entry point for a detailed design for the R&D team; and 4) a *Deployment View* to support issues such as configuration and deployment of components using on-premise hosting, virtualization and /or leveraging of external (e.g., Public Cloud) environments.

The integrated analysis and design process, introduced briefly in the following section, defines the architecture activities and artifacts, regardless of the software development lifecycle method (e.g., Agile, Waterfall or Incremental) it is intended to support. Its detailed description is beyond the scope of this paper. The remainder of this paper focuses on the Logical View as the fundamental pillar of the design phase, its design goals, and introduces the *4x6 Tiered Architecture Method* (or *4x6 Method,* for short) approach to logically architecting an enterprise solution.

Application of the 4x6 Method to an enterprise solution imposes a structured design, in terms of steps to follow, structure and documentation, and yields a six-tiered architecture structure and an abstract architecture specification. This specification expresses the various components, dependencies and design patterns using a graph-based data model (or "architecture catalog") and blueprint, the latter expressed as both a diagram and XML document. This structured approach enables a software product line assembly [6][7] of reusable patterns and components, with increasing development efficiency over time, and applicable to a whole domain, solution, sub-system, product, or a single component.

2 Analysis and Design Process

The C4A methodology is an attempt, based on best practices, to address the five architectural concerns presented in [6][7], namely economy, visibility, spacing, symmetry, and emergence.

The *Economy* design goals are to define and detect usable, unique IT modules and services and evolve toward value-added solutions built on top of common technologies and services. The *Visibility* design goals are to employ a unified service and data language across the various IT domains, and to implement systematic architecture taxonomy, symbolic representations and patterns across the solution's various levels of documentation. The *Spacing* design goals are to provide replaceable modules which are loosely-coupled and based on a services model as well as produce granular pre-packed smaller offerings of dedicated (composite) services yielding reusable commodities. Finally, The *symmetric* main design goal is to support a service abstraction and orientation approach, such as the virtual IT services found in a Software-as-a-Service (SaaS) delivery modality, regardless of the actual delivery mode. Such a decoupling between a service and its actual implementation enables symmetric provisioning and consumption of the service, thus supporting the construction of a composite application in a supply-chain manner [8]. Additional goals are to standardize over time by using common technology and IT services to embed within the IT integration framework. The *Emergence* design goals are to detect

and adjust to changes in underlying commodities, and explore new ways of interactions among systems and users (e.g. via mobile devices, virtual appliances, etc.).

To address the architectural concerns listed above, the analysis and design process embedded in the C4A methodology includes two analysis and two design phases, to correspond with the four views shown in Figure 1. These are briefly outlined below.

The *IT Business Analysis Phase* operates within the IT business view of the subject IT solution. During this phase the solution architect creates a business value proposition according to high-level business expectations. Various graphic means may be used to represent the result of this phase, including an architecture stack, flow or PERT charts, and the like. For our example, shown in Figure 2, we chose a stack representation. In our "hello world" solution, its value proposition consists of mobile interaction in a multi-lingual, user-role-sensitive scenario. During the business analysis phase we separate our offerings into two basic blocks, Sentence Building and Multilingual Adjustment. The former composes a structure of English strings, while the latter has replacement capabilities of these strings with corresponding sentences in other languages. Only after successfully building these two blocks, one can address the need to limit access to the functionally by the user, depending on their identity and corresponding role(s) within the enterprise. Once this intermediate block is properly addressed, attention can shift to the question of where the solution is accessed from, a desktop or mobile device. Thus, the full solution features a gradual implementation roadmap for continuously measured progress.

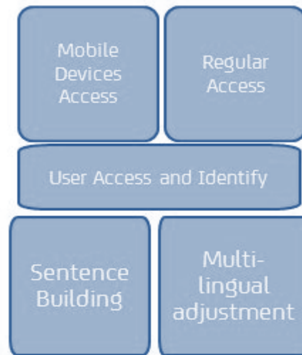

Fig. 2. Stack representing a mobile interaction in a multi-lingual, user-role-sensitive scenario

The *Functional Analysis Phase* of the process works inside the functional view. This analysis captures the solution's capabilities organized as an IT service lifecycle. These capabilities are structured as jobs-to-be-done (JTBDs) defining the action verb, the object of the action, and the contextual classifier [5], driving technological integrations and ultimately defining the customer architecture assimilation roadmap(s). Following our example, the capabilities of the multilingual adjustment offering will be: (JTBD1) replacing English sentences with the corresponding text in a different language, and (JTBD2) on demand provisioning of a number of language libraries. Each JTBD is tagged and capabilities are collected according to these tags, to be later mapped to an implementation component in the logical design view. In our

example, JTBD1 is tagged as “change,” and JTBD2 as “model”. Although any tagging is possible, in C4A our IT service lifecycle phases/tags are model, assemble, change, monitor, and optimize.

The *Logical Design Phase* corresponds to the design view and focuses on integration (via APIs) and the functional layer (GUI, if applicable), mapping structures to the components in the functional capability view created by the previous phase. Further, the architect examines possible mappings of available design patterns across the logical layers, and plans alternative roadmaps for evolving the architecture. Understanding the logical and physical dependencies among the various components is critical to correctly estimate the overall quality attributes and performance characteristics of the solution being built. For example, a component depending on a lower reliability module needs to account for that exposure when its overall score is calculated. The detailed application of this phase to our “hello world” solution is provided as part of the detailed discussion in the next section.

The *Deployment Design Phase* acts within the design view and its objective is to gather the various components making up a particular instance of a solution. The main activity in this phase is the definition of computing resource requirements and constraints on their nature (physical, virtual, or Cloud) to fulfill the needs of the logical components. In the context of our “hello world” sample solution, the design could prescribe the multi-language components to be consumed as a service and thus would not require deployment. Also, the design could require the mobile device component to be highly scalable and use cache memory for improved performance, while the sentence building component needs to be restricted to a maximum memory footprint, yet can be run anywhere as a stateless server.

The overall design intent of a logical architecture and its blueprinting implementation must:

- enable managers to leverage resources across their portfolio [6][7][9];
- separate non-unique technologies into interchangeable consumable commodities [6][7];
- enable the composition of a technology from underlying patterns [4];
- provide a modular architecture and appropriate evolution roadmap [10];
- increase the overall quality attributes of the architecture structure [11][12][13]; and
- enable the structured estimation of aggregated system quality attributes [8].

The challenge is thus to create an architecture focused on consolidation and optimization of component and service (re-)use, while increasing the overall solution’s quality attributes. We must do so while bundling all of the design requirements, constraints, principles and directives into a single architecture blueprint reflecting a good enterprise solution design.

In the following we examine the 4x6 Logical View and its capacity to address the above challenge, by enabling an architect to systematically model and superimpose existing and future technologies, and design the architecture evolution of an individual product or enterprise solution.

3 The 4x6 Logical View

The structure of the architecture obtained by applying the 4x6 Method consists of 4 conceptual tiers, or stereotypes, that underlie 6 logical tiers, yielding the “4x6” designation. The four conceptual tiers (the “4” in 4x6), are defined in the C3A approach [4], namely: 1) business integration; 2) functional architecture; 3) system architecture; and 4) cross-concerns. As for the six logical tiers (the “6” in 4x6), they now include a mapping of the classical three-tier architecture pattern (Presentation, Business, and Storage), plus three additional tiers. The resulting tiers are: 1) virtual IT services (a business integration stereotype); 2) views (corresponding to the classical Presentation tier, a functional architecture stereotype); 3) business logic (corresponding to Business, a system architecture stereotype); 4) data (corresponding to Storage, and also a system architecture stereotype); 5) integrated services (another business integration stereotype); and 6) common components (a cross-concerns stereotype). The 4x6 logical view layout is depicted in Figure 3.

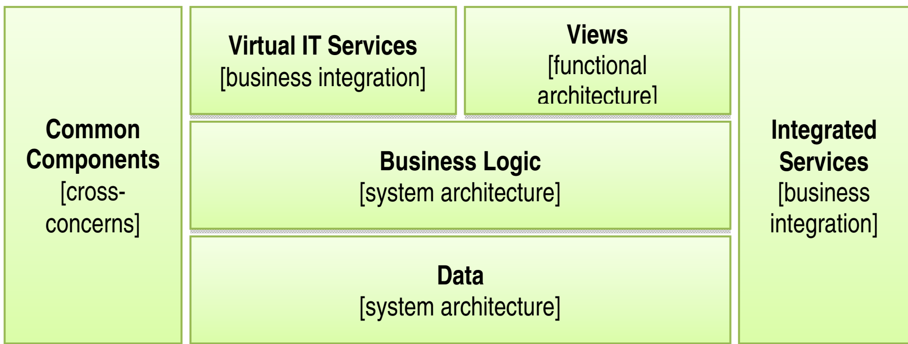

Fig. 3. The 4x6 logical view layout; square brackets indicate the kind of conceptual tier, or stereotype

It is worth noting C4A uses the four stereotypes (business integration, functional architecture, system architecture, and cross-concerns) in its 4x6 view in order to cater to a different stakeholder, namely external integrators, functional architects, system architects, and common components managers, respectively.

A hypothetical layout for our “hello world” solution is depicted in Figure 4. In it we see the four stereotypes (outer rectangles, colored yellow in the modeling tool) and three abstraction levels. Level 0 (middle rectangles, colored blue in the tool) represents high-level modules or sub-systems. Below it, Level 1 (inner rectangles, colored green) contains deployable components, which may be removed and replaced with similar components without affecting the rest of the system or requiring the replacement of a full Level 0 module. Level 2 (colored orange, not used in this example) contains an internally cohesive set of components that is usually deployed or managed as a unit.

Color is also used to provide the state of components, either existing or future, in a single view. Those components intended for future development (or modification) are left white by default (e.g., support for right-to-left languages in Figure 4) and can be set by the architect to any color in order to reflect the timing (or extent) of the implementation.

A summary description explicitly calling out the scope of new product release(s) within the overall solution, as well as their long-term architecture roadmap, are also produced during the building of the solution's layered architecture. In our "hello world" example (Figure 4), the blueprint suggests that most of the components existed and were implemented, due to their color-coding as either blue (Level 0) or green (Level 1). The additional capability being added (in white), to translate languages read/written from right to left, is limited to string building. The purple color (rectangle *Language format* in this example) is used to signal the timing (e.g., next release) for the additional capability to appear in the solution. These new components are owned by the development team, since they reside at the middle business tier.

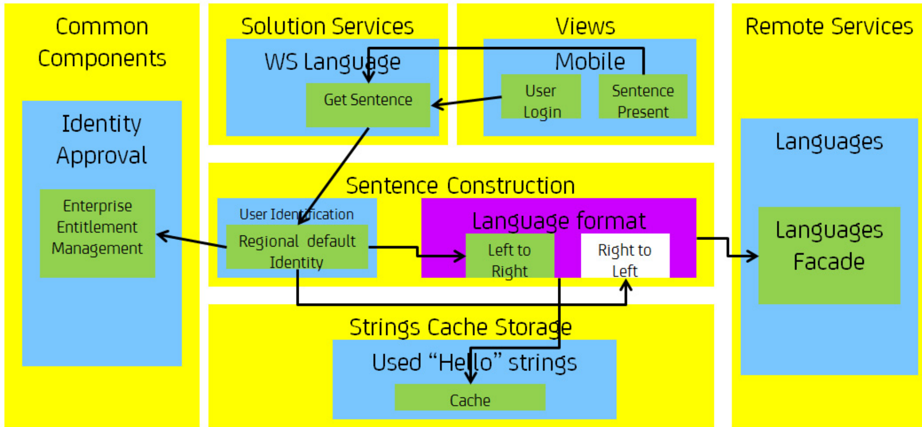

Fig. 4. The 4x6 layout for the "hello world" solution

The blueprint also shows (in the External Services tier) the usage of external language sources, provided via a façade design pattern. On the other hand, the association of a geographical region with a user and the corresponding selection of a default language are done through querying a common component of the enterprise. By definition, common components are not part of the architecture development, but are instead treated as if they came from a third party, but with a known implementation. Thus the placement of components among the six tiers and their color-coding enable the architect to provide implicit information about the characteristics of a solution to other designers and observers. Naturally, the simple example presented here may evolve into many components, depending on the granularity and complexity of the solution. Yet, they all adhere to the same design principles presented in this paper.

This logical blueprint, which contains components properties and attributes, may also be used to create a top level (abstract) design specification document for a product or solution planned release, as well as documentation for the product/solution family as a whole. Still, the blueprint's most valuable use is in assisting with the definition of architecture evolution roadmaps, from an existing state to new releases over a given period of time.

In the next sub-sections we explore ways for an architect to most effectively use the 4x6 logical view to convey architectural intent, followed by an examination of the six logical layers. We wrap up this section with a brief description of a tool we developed to assist the architect with the creation of the 4x6 logical view.

The 4x6 logical view captures the software elements of a solution, including components and their logical interactions. We found architects benefit by considering four design value propositions as they create the 4x6 logical view. These are *components coupling* for obtaining loosely-coupled structures for future assistance with system development and deployment, *components cohesion* for separating the solution's intellectual property from needed functionally, *meaningful architecture naming* for conveying the business value of the design elements, and *composition of aggregated tiers* for dealing with solutions composed of a single technology.

When constructing an architecture using the 4x6 Method, the loose coupling between its tiers hints at their potential separate deployment. The distribution onto six tiers is also meant to reflect logical constrains or concerns for the system design, captured as six logical layers. Logical layers do not impose coupling restriction, but rather a division of responsibility, defining a weak cohesion relation or grouping of components based on type, not cohesiveness based on the same functionality. These are important for any design, but more so for existing solutions and products not featuring six logical layers built as six separate tiers. These solutions have the components to fulfill the intent of the logical structures, and will gradually refactor their structure into corresponding tiers, enabling rapid integration with other products to form a larger solution.

Our aim is to work with or towards tiers, but most current product and solution designs feature only layers. Therefore, in the reminder of this section we shall use the term "layer" to replace "tier," keeping in mind the logical rather than the physical intent, as explained above.

We start with the *Views Layer*, which addresses the visibility, spacing and emergent concerns of an architecture. This layer contains the user interface(s) and presentation rendering adjustments, which are involved in human interaction. The rendering adjustments adapt the interface to a particular device type or presentation technology, such as mobile, tablet or desktop, browser, Linux or Windows, flash or AJAX, etc. Therefore, this view supports the "View" in the Model-View-Controller design pattern [6][7], whereas "Model" and "Controller" are part of the Business Logic Layer (see below). User interface and human factors design are part of this layer's best practices. As a result, the three main Level 0 components in this layer should be "Multi-device UI façade," "Reports Publisher," and "Content Synchronization controller."

Next is the *Virtual IT Services Layer*, to address all five architectural concerns, namely economy, spacing, emergent, visibility, and symmetry. The main purpose of this layer is to encapsulate programming namespaces, and in many cases it merely holds the system software development kit (SDK) in several interoperability formats to support the external activation of business transactions. Consequently, its two main Level 0 components are a regular SDK, called "Published Service," and one employing a domain unified language, called "Published Canonical Data."

The *Business Logic Layer*, addresses spacing, economy, symmetry and visibility concerns. This layer encapsulates the intellectual property of the offered technology. Any interaction with other layers should be surrounded by boundary components, model interfaces, or the data layer. The boundary components maintain a clean design-to-test approach, consumed via the "Internal Domains Specific Model" Level 0 component, and the Level 1 "Web Interfaces" component. The third component is the "Data Access Layer" or DAL. The DAL abstracts the persistency technology and exposes the CRUD operations (Create, Read, Update, Delete) for translating data into the object format understood by the Business Logic Layer. An example of such a translation is needed for stream-based non-persistent data retrieved from agents and monitors

The *Data Layer* addresses the concerns of visibility, spacing and emergent. Its focus is on increasing data resiliency, improving I/O performance, and providing ease of content scaling and expansion.

Focusing our attention on the bracketing layers in Figure 3, we first look at the *Integrated Services Layer*, which addresses the economy, emergent and symmetry concerns. This layer abstracts remote activation of (Web-) services, presumably commoditized ones, and presents a single gateway to general services such as reporting, logging, identity management, message bus, and more. In a cloud-based era, these services may be consumed from a remote vendor, and paid on a per-use basis. By encapsulating remote activations and delegating local calls, directly or indirectly, this layer supports future advances in technologies instead of rigid deployment and coupling. Therefore, this layer deals with how to design issues such as a good API, Generic versus Specific API, Web Services Variation Façade, Canonical Data Model, Hub-and-Spoke, Broker, Observer, and Publish/Subscribe patterns.

The second bracket in Figure 3 is the *Common Components Layer*. This layer addresses economic design goals, containing embedded and deployed components that are considered part of the compound solution, although not constructed by the development team. The first option for using a common component is to enforce its installation regardless of the number of instances already deployed at the customer's site. The second is to conditionally install components only when they are not already present in the deployment environment. Notice that Common Components are part of the provider solution and must not be replaceable by the customer.

Recall that the 4x6 components proposed within each layer should be given archetypical names, hinting at the design patterns used to define its technological purpose. When the design is instantiated as a real blueprint, the business (value) name is added giving the component a (composite) meaningful name.

A modeling Tool: The 4x6 logical blueprint is implemented in a modeling tool called *CAM Logical Architecture* (CAM stands for CA Architecture Management) displayed in Figure 5. Each of the model's components has the following set of properties: interfaces, functional description, non-functional level of enterprise compliance (the so-called "ities," such as reliability, scalability, security, etc.), and resource requirements. The component blueprint, component properties and dependency information, and overall blueprint properties are stored in a component catalog. This catalog is used to produce reports and can translate content to other formats outside of the tool's technology. Also provided is a centralized library of captured information at the individual component level and blueprints, enabling information sharing and design reuse among CA's architects community.

The component catalog can effectively produce an Architecture Abstract Specification [11], automatically providing about 70% of the information needed for examination by the internal CA architecture review board. The modeling tool itself, which operates in either centralized or local mode, is based on the Eclipse Modeling Framework and the ECORE model as a standalone logical editor.

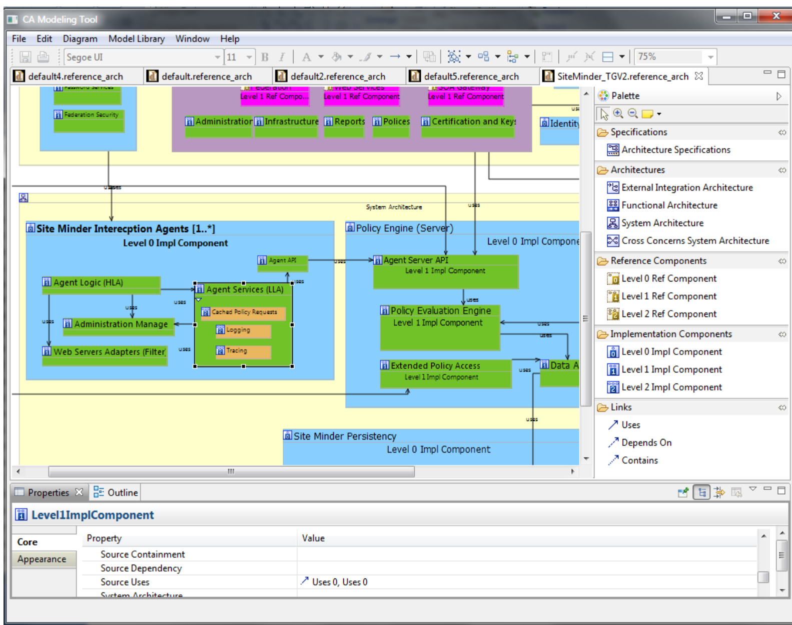

Fig. 5. The CAM Modeling tool for the 4x6 logical architecture

4 Applying the 4x6 Method in Practice

On January 2011 the 4x6 Method and CAM tool were presented at the CA corporate architects' conference in front of more than 300 architects, as part of a new method of modeling and capturing architecture information in an agile manner. It was

enthusiastically received by many of the architects, who requested guidance to use the method and tool. Following the conference, educational material was recorded on video and made available as part of the delivery process.

Training on the tool and 4x6 approach takes less than an hour, assuming basic architecture knowledge in modeling, while the practical implementation of a design depends on its complexity and the architect's familiarity with design patterns and techniques. The structure of the 4x6 Method forces the architect to systematically think about the design intent and concerns, fostering gradual design implementation. To quickly introduce the tool and 4x6 concepts to the architects' community, coaching is done on a train-the-trainer manner, with mentoring and support by the central CA architecture team.

Starting in March 2011, a large number of architecture teams have applied the 4x6 Method and associated CAM tool. During the period of March 1, 2011 to October 21, 2011, a total of 53 distributed enterprise-grade products were involved in architecture reviews. Of them, 20 projects were *Versions* (implying a potentially more significant architectural effort) and the other 33 were *Releases* (large development efforts extending a current architecture).

Of the 53 projects aforementioned, 27 adopted the 4x6 Method. Of these, seven were Versions and the remaining 20 were Releases. From the start of a project to its review checkpoint took an average of 5.2 effort-months for a Version (11 month maximum – 2 month minimum), and a slightly higher average of 6.0 effort-months for a Release (12 month maximum – 2 month minimum).

24 projects kept the old style of architecture documentation. Of these, 12 were Versions and 12 were Releases. The corresponding efforts to reach the review checkpoints for these projects were in average 6.5 (13 Month Maximum – 1 month minimum, for an exceptionally small effort) and 7.7 effort-months for a Version and Release, respectively. The remaining two projects produced each a unique document. Both of these were Releases and took 8 and 9 months.

Of the 27 projects using the 4x6 Method as the foundation for their architecture, 12 were told no formal review was required because the reviewers understood the approach as documented. 11 were Releases and 1 was a small Version. The 15 projects formally reviewed had an average of three significant comments as a result of the review. Among those projects using the old style for documenting their architecture, 12 projects formally reviewed in a meeting had an average of 7 significant comments.

Reviewers reported less time was needed to prepare for the review session when dealing with architecture documents based on the 4x6 Method, and a better understanding of the designs was possible, when compared with the old approach to architecture modeling and documentation. The reviewers also remarked on the consistency of the models across projects, and how this aided comprehension and identified areas that might have not been discussed otherwise.

Project review meetings, on average, were 90 minutes long for a 4x6 Method based project, compared to 120 minutes (or more, in a few cases) for the old method.

5 Conclusion

In this paper we presented the 4x6 Tiered Architecture Method, the core of the CA Four Architecture views model-driven architecture methodology. Its use enables the

capture of business, functional, logical and deployment views to maintain control over architecture evolution. In particular, design intent and architecture directives are captured within the 4x6 logical view. The resulting view can be used to state design goals and process, evolution steps, design rationale, as well as recommend best practices for structural composition of an enterprise product.

The ability to superimpose or integrate architecture elements requires them to be structured using identical format, templates and tools that foster collaboration and content reuse. In our case, the six logical layers and the use of design patterns are the methods, the CAM modeling tool is the means, and the component catalog serves to maintain and share information.

Both the 4x6 Method and CAM tool were tested and verified by practitioners on the design of real products, passing the corporate architecture review board with flying colors. By employing the systematic thinking and formal modeling of the architecture concerns, as highlighted by the supporting methodology and tools, the 4x6 approach forced architects to consider the use of design patterns, change the structural layout, and consider component implementation by third-party technology.

References

1. Kruchten, P.: The rational unified process: An introduction, 3rd edn. Addison-Wesley Professional, Boston (2003)
2. Rising, L.: The benefit of patterns. IEEE Software 27(5), 15–17 (2010)
3. Maiden, N.: Service design: It's all in the brand. IEEE Software 27(5), 18–19 (2010)
4. Hadar, E., Silberman, G.M.: Agile architecture methodology: Long term strategy interleaved with short term tactics. In: Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, pp. 641–652. ACM, Nashville (2008)
5. Silverstein, D., Samuel, P., DeCarlo, N.: The innovator's toolkit: 50+ techniques for predictable and sustainable organic growth. Wiley, Hoboken (2008)
6. Buschmann, F., Henney, K.: Five considerations for software architecture, Part 1. IEEE Software 27(3), 63–65 (2010a)
7. Buschmann, F., Henney, K.: Five considerations for software architecture, Part 2. IEEE Software 27(4), 12–14 (2010b)
8. Burbeck, S.: Application programming in Smalltalk-80: How to use model-view-controller (MVC) (1992), `http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html` (retrieved)
9. Bosch, J.: Toward compositional software product lines. IEEE Software 27(3), 29–34 (2010)
10. Ferguson, D.F., Hadar, E.: Constructing and evaluating supply-chain systems in cloud-connected enterprise. In: Moinhos Cordeiro, J., Virvou, M., Shishkov, B. (eds.) Proceedings of the International Conference on Software and Data Technologies, ICSOFT 2010, pp. 69–76. SciTePress, Athens (2010)
11. Sherman, S., Hadar, I., Hadar, E., Harrison, J.: Architecture documentation for agile development. Paper Presented at SATURN 2011, San Francisco, California (May 2011)
12. Booch, G.: An architectural oxymoron. IEEE Software 27(5), 96 (2010)
13. Buschmann, F.: On architecture styles and paradigms. IEEE Software 27(5), 92–94 (2010)
14. Hadar, E., Perreira, M.: Web services variation façade – Domain specific reference architecture for increasing integration usability. In: IEEE International Conference on Web Services, pp. 1207–1211. IEEE Computer Society, Salt Lake City (2007)