Agile Architecture Methodology: Long Term Strategy Interleaved with Short Term Tactics

Ethan Hadar
CA Labs, CA Inc.
Yokneam, Israel
+972-54-4752648
ethan.hadar@ca.com

Gabriel M Silberman
CA Labs, CA Inc.
New York, NY, USA
+1-212-415-6908
gabby.silberman@ca.com

Abstract

Tactical agile development and strategic architectural evolution are viewed as opposite ends of the development spectrum, with the long-term value of applying an architecture-centric approach seemingly at odds with rapid development, featuring its informal documentation activities. The value of a long-term vision, or architecture, is particularly high in the context of product integration and research. However, there is also benefit in rapid and early feedback on incremental deliverables, as featured in agile development.

To extract the main benefits of both worlds we introduce the CA Agile Architecture (C3A) method, targeted for the architecture and design phases of the development life cycle. Its foundation is the Reference and Implementation Architecture, which features a lean one-page per component contract, as well as several abstraction levels. The C3A artifacts are governed by a cyclic process of architectural evaluation and evolution, with accompanying coaching and training activities.

This work-in-progress is being prototyped with three product teams, varying in team size, product maturity and complexity, and geographical location. C3A features a common tactical-focused agenda for the functional and system architectures, with minimally overlapping strategic views.

Categories and Subject Descriptors D.2.7 [Distribution, Maintenance, and Enhancement] Restructuring, reverse engineering, and reengineering. D.2.10 [Design] Methodol gies, Representation. D.2.11 [Software Architectures] Domain-specific architectures. D.2.13 [Reusable Software] Reusable libraries, Reuse models I.6.5 [Model Development] Modeling methodologies

General Terms Management, Design, Documentation

Keywords Reference Architecture, Architecture Centric Evolution, Design Reviews, UML.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.
ACM 978-1-60558-220-7/08/10.

1. Background

The concept of agility is having a considerable impact on the practice of software design and architecture. Misunderstanding and misuse of the lean documentation approach of Agile development results in the lack of adequate architectural information [5] and strategic methodologies [2].

There is no argument that strategic reasoning, as captured in an architectural blueprint, is of value. However, development teams focused on the delivery of tactical short-term solutions feel they cannot afford to be delayed by strategic thinking. Therefore, they opt for quick evaluation tools for their design approach [6] and are satisfied with a high-level roadmap that can provide a tangible, incremental evolution to the next feasible solution [7].

Development teams struggle to create a simplified architecture, one resilient to change, to preserve their critical intellectual property (e.g., in the context of software product lines [8]). Therefore they fail to benefit from the value of the architecture-first approach as detailed by [3]. Furthermore, the diversity of stakeholders and their various perspectives on architecture [4] can also lead to confusion. Add to it the ongoing tactical adjustments triggered by influential customer needs, as advocated by many agile approaches [1], and the process may drift towards continuously changing requirements and instability.

In this work we aim to combine the quick feedback in delivering short incremental minor releases, as featured by an agile process, with the architecture-provided long-term vi-

sion, thus producing an Agile Architecture methodology. To be successful, such a methodology should:

(1) Provide a strategic *reference architecture* to illustrate the vision in technical and functional terms.

(2) Provide an *Implementation Architecture* to scope parts of the Reference Architecture into future releases.

(3) Provide a technique for gap analysis between current and envisioned states, thus constructing the incremental releases.

(4) Use lean, minimal documentation, featuring different levels of abstraction and separate perspectives for each stakeholder, aligned along a single architectural roadmap.

This paper illustrates the CA Agile Architecture (C3A) methodology, as prototyped with three products and their development teams. The prototypes varied from a new release of a mature product, through new product development, to the integration of several existing products. Section 2 details the circumstances leading to the development of this methodology. The C3A dimensions and artifacts are detailed in sections 3. The C3A methodological steps are defined in section 4, closing with the experience gained and conclusions in sections 5 and 6. In the following the terms system and product will be used interchangeably to mean a (complex) software system under development.

2. Genesis – In the beginning

Prior to introducing our applied approach, it is important to understand our Genesis. Why the classical approaches did not work for us, and accordingly, how did we come up with the new methodology?

2.1 The Development Team Perspective and Agile Constraints

Software products are usually delivered to customers in major release cycles. Within these major cycles, incremental minor deliverables are produced, as well as hot fixes, patches, service packs, and so on. Further, it is not unusual for development teams working on the next release to be occasionally directed to work on previous releases. These assignments are either handled by members of the original team that developed the previous release, or by a new and dedicated tiger team. The number and variety of tasks, if not handled carefully, may lead to delays in the next release schedule.

The Agile approach tries to minimize these interferences by shortening the time to value by frequent release cycles, and benefits from the feedback of real customer experience. However, in a large scale organization, aligning the different products released is difficult, not to mention doing it in a short timeframe.

Moreover, one must understand the conditions under which the development teams operate. It is of utmost importance to deliver highest quality working code to QA. The teams are constantly focused on execution. They must not be derailed from their schedule commitments as their fundamental obligation to our customers. It means that any activity labeled as “strategic”, “long term”, “theoretical”, “research”, “visionary”, or similar is immediately suspected as hazardous to the team’s schedule regardless of its importance.

2.2 The Strategic Architectural Perspective

Strategic (architectural) thinking is usually done at the early stages of a project, followed by a tactical implementation. However, in many cases, while executing the tactical tasks one discovers new opportunities, and the question is whether or not to pursue them, putting at risk deliverables of the original project. If some of these opportunities had been exposed early in the process, alternative solutions or different directions may have been chosen.

Business strategists, product managers, chief architects and the company thought leaders practice strategic activities daily. However, when under time and resource constraints, strategy takes a backseat to on-time product delivery. Compromises are made.

2.3 Conflicts of Interests in Existing Methodologies

The time horizons of the strategic and tactical perspectives conflict, namely, one cannot “design everything beforehand” and use the waterfall top-down approach because high uncertainty and instability. Moreover, due to critical task lists and short-term goals, risky transient solutions (which may be eventually refactored) are used.

Modeling exercises done at the beginning of a cycle might be later forgotten simply because they are not needed in every step. The code lives on, interfaces change and mature, and the architecture perspective slowly becomes obsolete knowledge.

On the other hand, when the same product keeps evolving over time, the amount of detail conceals the essence of the application. The number of abstraction levels is usually too small, resulting in a wide gap between the highest (conceptual) and lowest (concrete) layers. Consider, for example, the difficulty in maintaining an architecture, which maps from requirements and use cases directly to application code, while delivering the knowledge from one generation of developers to the next.

The dilemma is obvious. Should the team follow the strategic vision and keep on thinking what to do and where to go while documenting their vision? Or, should they focus on the

daily deliverables and keep on executing the prioritized tasks?

The obvious answer is both. However, how could they accomplish this?

The truth of the matter is that Agile approaches do not emphasize the architectural centric activities, but rather rapid selection of tasks, rapid design sessions, with lean, or even non-formal documentation. The dominant tools are white-board, camera, PowerPoint slides, and the development suite. Validation is done with clients and frequent testing.

On the other hand, Model Driven Architecture (MDA) tools such as Magic Draw [9] from NoMagic and automatic code comprehension such as Structure 101 from Headway [10], are seldom used due to their perceived complexity and lack of understanding of their tangible value. It is simply easier for the developers to work on the low-level development environment they are used to. They choose to do so instead of mastering abstraction techniques, since their fellow developers communicate knowledge the same way. To illustrate this point, complex program examples found on the Web, even recently produced ones, are shown mostly as code, without the use of either UML diagrams or artifacts. Have you ever wondered why?

3. The CA Agile Architecture (C3A) Dimensions

To effectively accommodate an architecture's interested parties, C3A must provide a common strategic vision of the product(s) as well as an agile structure of loosely-coupled components with accurately described interfaces. Moreover, it should provide a perspective of both major and minor releases, serving as the architectural blueprint for the development team.

We believe the above would address the dichotomy between strategic thinkers and tactical implementers, which may be traced to the following factors:

(1) Control over Abstraction Barriers: "how low should we go?" Meaning, what is good enough documentation in terms of granularity?

(2) Minimal Documentations and Artifacts: "where should it be captured and how?" Or, how can we keep this strategic effort to the minimum (documentation) and maximize the impact on our tactical deliverables? Moreover, we must keep our strategic concepts and tactical deliverables constantly aligned.

(3) Synchronization Constraints: "when should we synchronize?" In other words, what is the time horizon of each architectural task?

These factors led us to the main C3A dimensions, namely: (1) levels of granularity; (2) number and nature of the artifacts; and (3) the time horizon of each artifact

3.1 Architecture Granularity and Abstraction Barriers

Maintaining different levels of abstraction is crucial to reducing information overload. Moreover, we want to stop designers from diving into too much detail, so imposing abstraction barriers is essential. This is done by mandating only two architectural levels but allowing additional (optional) levels. The highest level of abstraction, featuring the minimum amount of detail, is the requirements level and is referred to as Level 0. At the other extreme of abstraction is the level representing software creation, namely the code level denoted as Level n. The more abstract the architecture, the less details it contains. Thus, Level 0 marks the software engineering process starting point, and features minimal details.

The above levels, and some in between, are described in the following.

3.1.1 Level 0 – The Modules Level

This level describes a system's logically-separate and self-maintained module, with distinct and disjoint functional responsibilities. It is captured in the Reference Architecture as main modules, and its functionality is usually reflected by one or more of the API (application programming interface), SPI (server provided interfaces) and API for GUI (graphical user interface). In the example shown in figure F-1, the *Statistics* Level 0 component (market as a UML package symbol), provides the capability of statistical reports of monitored information via the *Reports Provider* API.

For each Level 0 module, there is an accompanying one-page document or *contract* (detailed in section3.2.1) that describes the functionality and responsibility of the interfaces, rather than their technical details. The abstraction barriers for the more detailed level are the Level 1 components (market as the UML component symbol), described in general functional terms. In addition, the overall system dataflow may be outlined using Level 0 components, including logical activation dependencies. Thus, the main concern of this level is the system's main modules and their responsibilities.

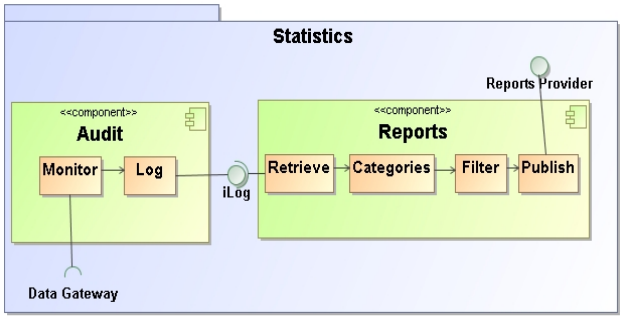

Figure F-1: Example of architecture granularity and abstraction barriers: Level 0 component (Statistics, market by UML package symbol)) is a module that contains self-deployed Level 1 components (Audit and Reports, market as UML

component symbol). The Level 2 internal components (monitor, log, retrieve and such, market as UML Class symbol) are not self- deployed and require a deployable unit.

Level 0 corresponds to the most abstract, and therefore most stable, architecture according to the Stable Abstractions Principle (SAP) in a component-based design methodology [2]. Thus, Level 0, as the name suggests, should not contain any implementation details.

3.1.2 Level 1 – The Components Level

This level focuses on the inner system components of Level 0. It provides the same structure of information as Level 0, but with higher granularity and more transparent abstraction barriers. These barriers are represented by the detailed interfaces, including parameters, protocols and messages, channels and ports. Level 1 focuses on usability, robustness around boundary conditions and fault tolerance, as well as providing a generic and expandable interface. The spotlight of this level is on a good interface design and a clear separation of functional concerns.

The abstraction barriers for this level are deployment capabilities. Each Level 1 component is self-deployed, thus it may be replaced locally without the need to re-install a full Level 0 component.

In figure F-1, there are two main Level 1 components, *Audit* and *Reports*. The *Audit* component defines a SPI called *Data Gateway* that monitors external data, as well as an Internal API called *iLog* consumed by the *Reports* component. The published API *Reports Provider* of the *Reports* component is externalized by the Level 0 *Statistics* component.

3.1.3 Levels 2 to n – The Technology and Design Levels

These optional levels deal mainly with a component's interfaces and their boundary implementation, according to design patterns such as façades, factories, bridges and proxies. In practical terms, Level 2 and beyond effectively represent the iterative construction, up to writing of the code, of the interfaces in Level 1.

Notice that in order to complete the Level 1 interface one usually performs a top-down analysis of its functionality, followed by a bottom-up collection of the interface's parameters**.** You start with "what needs to be done," followed by "what is required to accomplish it." The main concern of this level is with the proper application of design patterns. However, in many cases, it is of importance to understand the internals and flow interactions of the Level 1 component, in order to realize the use cases.

In our running example, the *Audit* Level 1 component contains the *Monitor* and *Log* Level 2 components, market as UML class symbols. The *Reports* Level 1 component contains a sequential dependency between *Retrieve*, *Categories*, *Filter* and *Publish* Level 2 components, aiming at organizing the logged information for specific report consumers.

Within Level 3 and beyond, one will encounter additional interfaces among internal components.

3.2 The Agile Architecture Artifacts

The C3A methodology addresses the different points of view by providing just two architectural perspectives. The first is for product management and integration architects, which deals with "what" to build and for whom. The second is for the system architects (product, frameworks, and common components) focused on "how" to build and what to reuse. The specific perspectives of developers, architects, development and product management, as well as business strategists, are all linked in just a single diagrammatic blueprint, shining the spotlight on their respective architectural concerns, namely: integrations, functional, system and cross concerns. The missing details are simply handled elsewhere, according to stakeholder-specific best practices. Tools, documents or tacit knowledge are all acceptable, as long as the majority of the organization's teams are C3A-aligned.

To fulfill this minimalist approach, the C3A methodology uses the Reference Architecture (RA) diagram to represent the strategic structure and function of the product's major release. A mapped-upon Implementation Architecture (IA) diagram, illustrating the status and scope of the nearest (in time) minor release, complements the RA.

Consider our previous example mapped into RA as it might exist in an enterprise *Reporting* application (figure F-2), such as the one from Business Objects [11]. It requires streaming data (collected by the *Audit* component by the *Data Gateway* Interface), filtering and aggregation engines (within the *Reports* component), a data warehouse and OLAP servers (within the Level 0 *Storage* component), report builders and a report user interfaces portals (within the *Trend Analysis* and *Periodic Reports* Level 1 components), and so on. Some of the functionality is consumed in remote activation using Web Services for Remote Portlets (WSRP) within the *SOA Supported Portlets* Level 0 component. Naturally, the security concerns will be handled by the enterprise level framework of *Access Control* and *Single Sign On* Level 0 components.

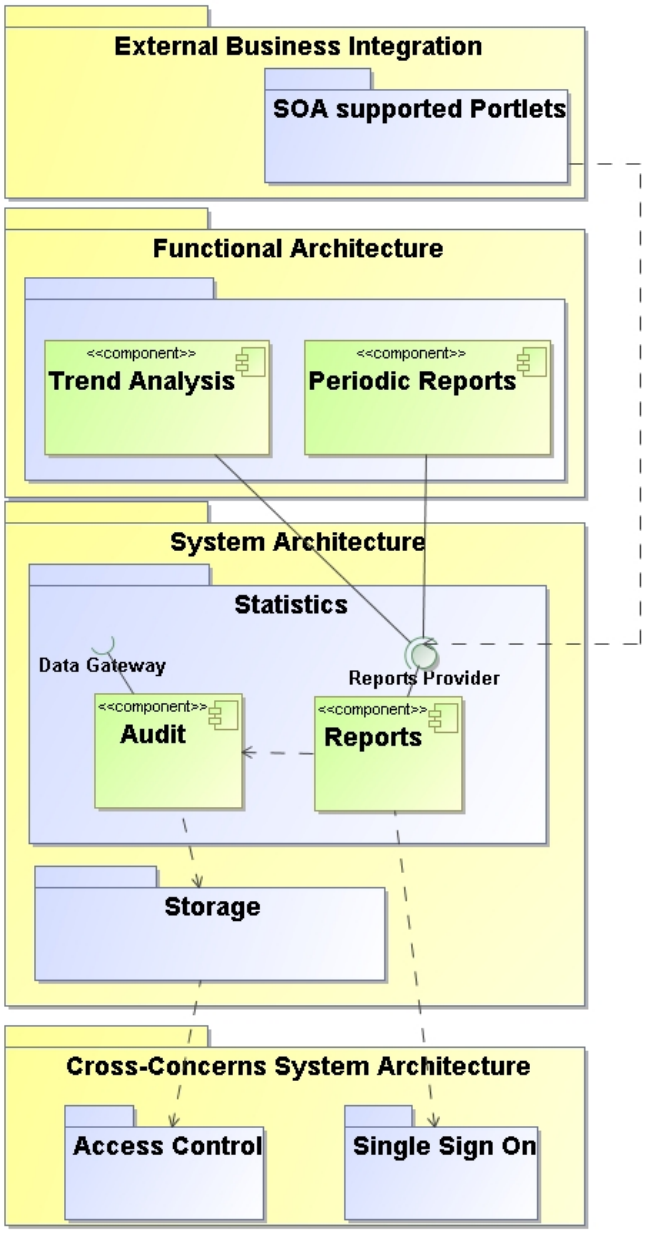

Figure F-2: Example of reference architecture for enterprise reporting system.

An IA for an instance of this example might include the exact selection of third party technology for the OLAP and warehouse servers, and proprietary engines developed in house, which will implement the *Audit* and *Reports*, as well as ready-made portals for the functional architecture layer from an existing solution.

Each of the components in the system is described in a single, one-page document that provides functional and technical information. These two artifacts, a diagram and a set of single page documents, are all the architectural documents provided beyond the actual code and internal code documentation. The RA will have a set of Level 0 component contracts, while the IA will have a similar set of Level 1 contracts. Each such contract is a written agreement between the component owner, its consumers and suppliers, on the type and nature of the component provided and required services, as detailed below. They are built according to requirements documents provided by product management, and assist in the QA testing specifications as well as provide information for technical publications.

The RA and the Level 0 one-page component contracts are living documents between major releases, representing the architecture evolution, and usually are kept constant for the minor revisions. The IA and the Level 1 contracts are snapshots of the evolving architecture and may change between minor revisions.

3.2.1 The One-Page Component Contract

This lean approach to documenting architectures in an agile environment requires avoiding duplicated documentation. One of the possible techniques works by minimizing the amount of information documented outside the code scope. However, since component owners do not own the overall system architecture, they need a simple contract that isolates their component's responsibility. Moreover, the contract provides a quick understanding on all the major issues that could affect integration among components.

If possible, the Level 0 component is handled by a team lead and the team developers handle the inner Level 1 components. For that matter, the team architect (may be the same person as the team lead) owns the full Reference and Implementation Architectures.

The main items to be listed in the one-page contract are:

- Name – the component name
- Ownership – the team member responsible for the component delivery.
- Responsibility – a brief description of the component responsibility. This should be no more that 3-5 sentences long, i.e., the functional perspective.
- Requestors – a list of the external components that request a service from this one using its API/SPI/UI.
- Dependency- a list of the external components activated by this component. It may include a dependency on a common data structure such as an external persistency system.
- Deployment – a short description of the deployment requirements. It can be a link to an external Deployment Description file.

- API – the detailed responsibility of each separate interface, or connectivity and activation protocols, based on the level of granularity (0, 1, or 2).
- Data structure – a description of the component's responsibility manifested by its underlying logical structure, namely, the internal system architecture. Although some methodologies claim this is not good practice, in many cases external users of the component need the detail to understand its functionality. Consider the case in which the component provides workflow services. Knowing the inner workflow steps, could provide an understanding of the underlying process, its invariants, and its capabilities, specifically if one would like to use unpublished (yet) capabilities and APIs.
- Scalability – limitations on the component's capabilities, or mechanisms for overcoming them. For example, is the number of transactions limited, or would replicating the same component add to the overall capabilities.
- Performance requirements –the required criteria, or defacto capability that a single component instance in its loaded condition will provide.
- Alerts and Errors – a list of alerts and error conditions not explicitly linked with the business process as exposed by the API. An example might be writing into a log file or sending a message via an event mechanism.
- Technology – details of technology which affects the implementation capabilities, specifically when using third party utilities and libraries. This section is critical for integrators to understand the technological limitations in connecting to the API/SPI/UI.

3.2.2 Reference Architecture

The Reference Architecture (RA) diagram depicts the aggregated responsibilities of the different stakeholders, the Level 0 and Level 1 components, as well as the dependencies among them.

The RA is a living, evolving document, capturing a wish list of building blocks. It functions as a reference point on "what we should have," not necessarily that we will ever have it all.

The RA elements, illustrated using a conceptual example provided in figure F-3 are (1) the RA layers; (2) Level 0 and Level 1 (RA) components; (3) visionary components; and (4) optional Level 2 components. These are further elaborated in the following.

The Reference Architecture Layers

Each of the four layers, market with UML package symbol, illustrated as light yellow boxes in the figure, encapsulates a different stakeholder perspective.

The External Business Integration layer is important for the business analysts, to leverage the system's integration capabilities with other (external) systems. The Functional Architecture layer represents the SPI/API and UI as presented to the product consumers. This layer, usually within the responsibility of the functional architects or product management, also provides a common understanding with QA, since it encapsulates all the integration points within the product, making it easier to test. The System Architecture layer, as the name suggests, exhibits the system's computational engines, without any external GUI elements. It is this layer's responsibility to implement the functionalities described in the previous layer. The Cross-Concerns System Architecture layer provides internal services to the system which are not part of the direct functional value-added features, but rather a part of its internal infrastructure. This last layer, similarly to the System Architecture layer, is under the responsibility of the system architect.

RA Components

Each of the layers contains components at the highest abstraction level. In figure F-3, the Level 0 components are colored in light blue. At this point, the one-page documents are created for each of the Level 0 components.

The Level 1 components are light green boxes within the different layers of figure F-3. In a new product where the code is generated after the architectural foundations are laid down, these components would describe actual code packages. With existing products, reverse engineering tools and processes may extract this information by harvesting tacit knowledge and examining the product's code.

Visionary or Strategic Research Components

The purple boxes, or darker ones gray levels printing, in some of the layers of figure F-3 represent areas where new, higher risk capabilities might be added. These are considered under research and strategic evaluation. Their presence serves to align the location and value of these new capabilities with the overall product architecture, and fosters debate on the value of a certain new capability, and what area(s) of the architecture may be impacted.

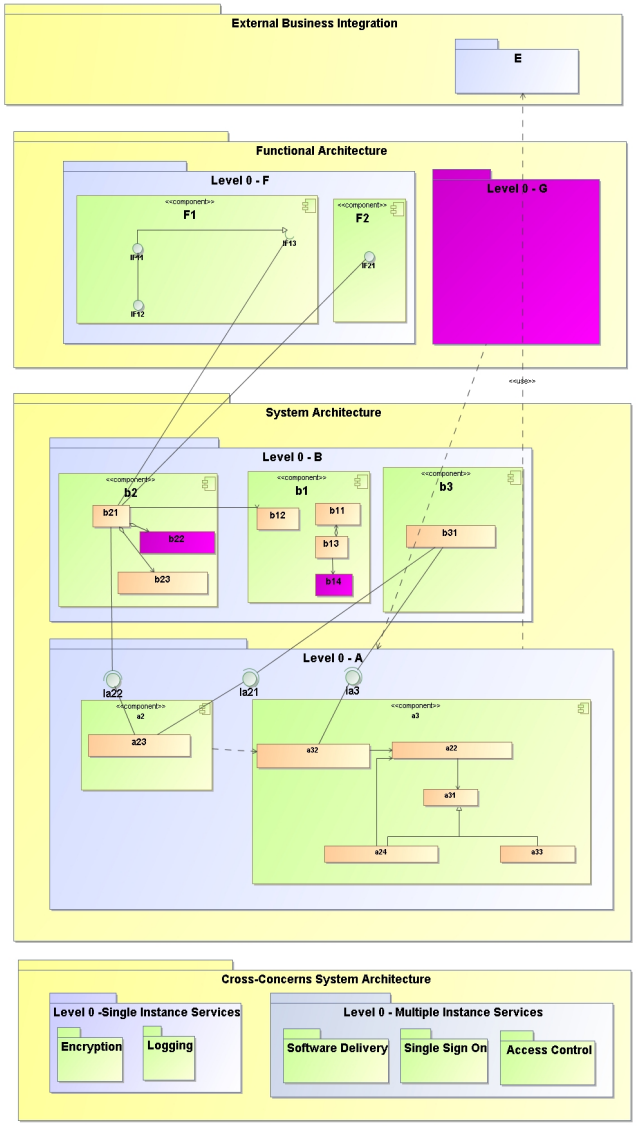

Figure F-3: An Example of a Reference Architecture. The light blue UML package boxes represent the Level 0 components; the green UML components are the Level 1 components; and the orange boxes UML class element, show the optional Level 2 components of the system architecture. Purple darker boxes represent visionary or strategic components.

Having components representing visionary capabilities in the RA does not guarantee their inclusion in the product. They may be deleted as a result of new discoveries, changing priorities, schedule and/or resource constraints. The rest of the RA is tacitly accepted to be on the product roadmap, as approved by product management.

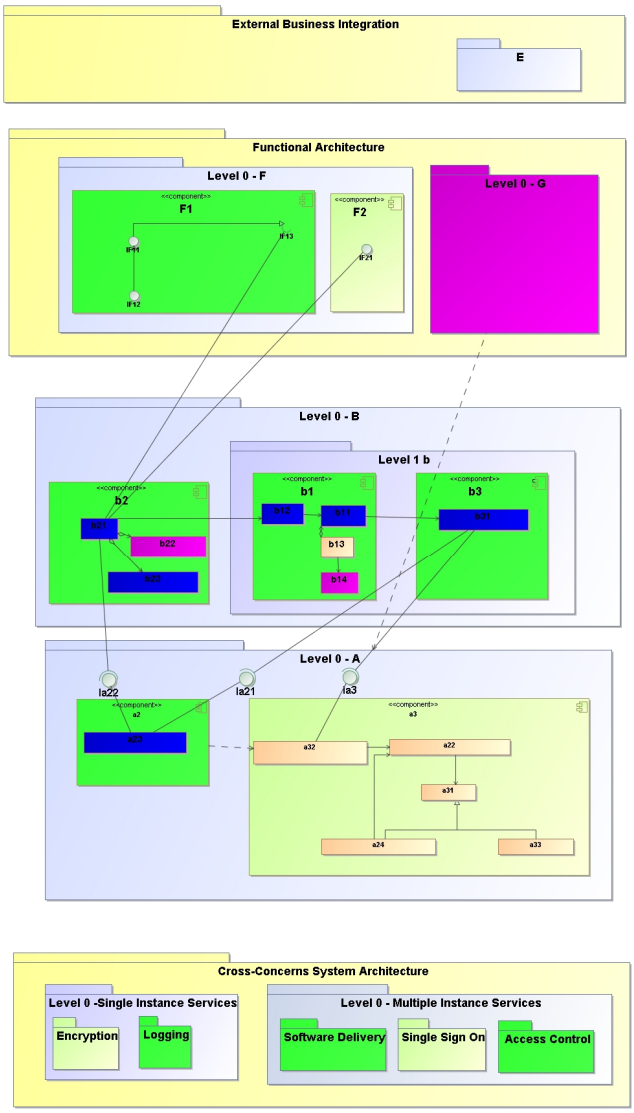

Figure F-4: An example of a scoped IA. The darker green boxes are the Level 1 components involved in a major release, and those in darker blue show optional Level 2 components specific to a minor release.

Requirements change and evolve, as do organizational needs, such as the increasing use of common components and the maturing of visionary research, causing the RA to evolve as well. For example, visionary research components may change their (purple) color when they are judged mature enough to be scheduled for prioritized implementation. Alternatively, they may be considered too risky and removed altogether from the RA blueprint.

Such actions should be considered very carefully, since they shape the future of the product. In the discussion leading to these decisions, all stakeholders should participate and collaborate to reach an agreed-upon common vision.

3.2.3 The Implementation Architecture

Implementation Architecture deals with the current release under construction. Thus, it is important to highlight to the team, what components are undergoing change, or added to the system, e.g. the scope of the immediate release cycle. Accordingly, a different coloring scheme is used, overlaying the RA, to mark the components of the IA. For example, figure F-4 shows the scope of the IA, with dark green marking the participating Level 1 components and the dark blue showing the affected Level 2 components.
Notice that this overlay affects both Level 0 and Level 1 components.

In this example, the IA represents an existing system. Even though the RA (figure F-3) indicates the *b1* and *b2* Level 1 components should be distinct, the scope of the IA does not have the corresponding Level 2 *b13* and *b14* components. Moreover, in figure F-4 *b1* and *b2* in Level 1 are merged, and they are thus aggregated in the IA as one, with a new Level 1 wrapper component termed *Level 1 b*. A simple comparison of the models in figures F-3 and F-4 highlights the gaps between the RA and IA. These gaps will exist until the architectures converge. Ideally this should happen by the IA aligning to the RA, but unfortunately, due to tactical constrains, sometimes the RA will change according to the IA.

It is possible for the IA to contain components not present on the RA simply due to legacy code. This gap between "what we have" and "what is important for the product evolution" should eventually disappear as the result of refactoring activities during successive incremental releases. Moreover, if the existing product features the RA functionality, but it is structured differently, we will map the components in an aggregated intermediate (to be refactored) component. This component represents the difference between the ideal RA and that of the existing implementation.

In an agile development cycle, tasks are scoped and selected from a prioritized "to do" list of features and functionalities. The C3A diagrams clearly highlight any backlog IA features relative to the RA, thus creating a common understanding among the various stakeholders. This enables better tradeoffs in balancing of resources and effort among the Level 1 components, the possible scope for the next minor release, as well as alignment of the product's vision to customers' needs.

3.3 The Time Horizon

Putting the artifacts and the level of granularity on a time scale is important for synchronizing the different activities. The question to be answered here is when should we update and re-align the vision with the tactical needs of the artifacts.

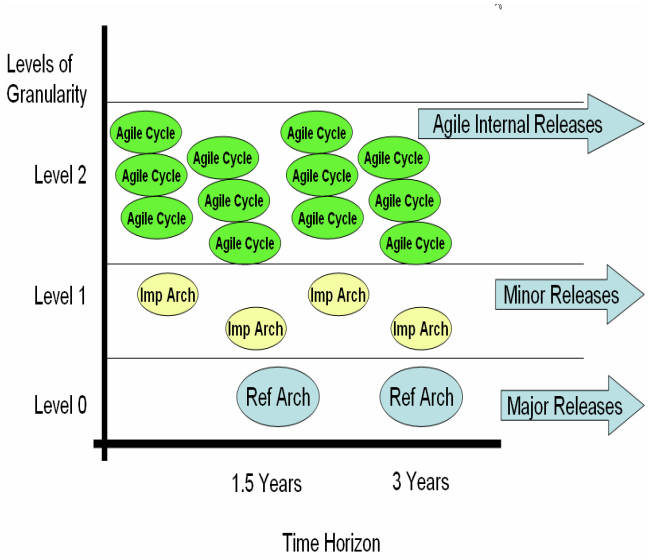

Figure F-5: An example of the C3A dimensions: Artifacts, Granularity levels, and the Time Horizon as related to the product releases and development activities.

Since the granularity of Level 1 components is higher than the granularity of those in Level 0, separate IAs (for different teams) may exist for a given RA. Consequently, for each IA there may be several ongoing agile implementation efforts focused on Level 2 and beyond.

Figure F-5 illustrates the temporal relationship among the Reference and Implementation Architectures, and Agile Cycle sessions. In this example, the RA is updated every 18 months, the IA every 7-8 months, and the agile cycle sessions last 2-3 months. However, it is important to note that any major issues appearing during a Level 2 design session should be propagated to the encompassing Level 1 IA.

4. The Agile Architecture Methodology Steps

Putting it all together in a structured manner requires following step-by-step instructions during product evolution. The methodology steps define what activity should be done when, and what level of detail is needed.
In our methodology, we employ a 7-step "LOW RISK" process, which stands for "**L**isten and **O**bserve, **W**atch, **R**eflect, **I**mprove, **S**crutinize, and **K**ick Start". As illustrated in Figure F-6, they are organized in two connected cyclic phases: Evaluation (**L**isten, **O**bserve and **W**atch), a **R**eflection binding step to close the outer cycle, and an Evolution (**I**mprove, **S**crutinize, and **K**ick Start) inner cycle.

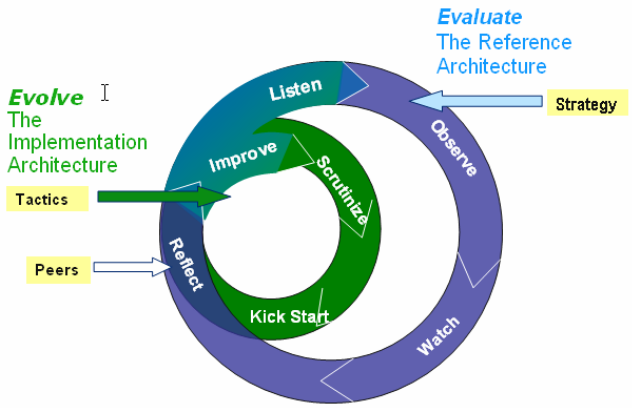

Figure F-6: The C3A LOW RISK process for evaluation of the reference architecture and evolution of the implementation architecture. It is comprised of 6 steps linked with a reflection step.

The LOW RISK steps are:

1. **Evaluate (focusing on the RA)**
   - 1.1. **L**isten and **O**bserve:
     - 1.1.1. Collect architectural documents, technical publications, manuals, PowerPoint presentations as well as tacit knowledge.
     - 1.1.2. Capture the first conceptual architecture into a RA, according to the perspectives layers, and define the Level 0 components.
   - 1.2. **W**atch
     - 1.2.1. Learn what other products are building, as well as new emerging technologies and solutions.
   - 1.3. **R**eflect on your Architecture Design:
     - 1.3.1. Validate the overall functionality and system modules with a steering committee of system and functional architects, practice engineers, support engineers, product management, and development management.
     - 1.3.2. Detail the status of the visionary research modules on the **next** RA cycle, while maintaining the present RA unchanged for the current IA release, thus preventing it from being derailed by uncertain components.
     - 1.3.3. Highlight design patterns recommended for the more detailed levels.
2. **Evolve (focusing on the IA)**
   - 2.1. **I**mprove:
     - 2.1.1. With every Level 0 owner, define and document the Level 1 components. For new products, this will be an abstract design activity, while in existing systems we recommend the use of Model Driven Architecture (MDA) tools such as Magic Draw or RSM.
     - 2.1.2. Repeat 2.1.1 until all Level 1 components are mapped correctly on the RA.
     - 2.1.3. Start implementing the internal IA minor releases.
   - 2.2. **S**crutinize:
     - 2.2.1. Map or provide gap analysis for two different minor IA releases relative to the next major RA release.
     - 2.2.2. Carefully adjust the IA while trying not to break the RA vision.
     - 2.2.3. If not possible, and critical differences between the IA and RA are identified, then pause and mitigate the impact on the RA. Return to step 2.
   - 2.3. **K**ick Start:
     - 2.3.1. If none of the original plans changed, modify the IA minor release plans to match the current RA, and execute on the implementation architecture
   - 2.4. **R**eflect (again)
     - 2.4.1. Estimate the effect of the IA on the next RA release, and propagate any gaps to the latest IA.
     - 2.4.2. If needed run the Evaluation cycle again (adapt the RA)
     - 2.4.3. If there are no major changes, continue with the next Evolution cycle (Evolve the IA)

These conceptual cycles enable us to focus on major and minor releases, while constantly evolving and using the same artifacts as a common base.

5. Experience Gained

The C3A methodology, a work-in-progress, has been prototyped in the context of three diverse products and their development teams. The first prototype targets an altogether new product, the second works on a new release of a mature product, and the third applies to the integration of several existing products. The same methodology applies for all these product categories with slight adjustments.

The new product is intended to produce value in a short period of time, meaning it features an aggressive schedule with rapid agile cycles. For this case we chose a single Implementation Architecture and performed a gap analysis with respect to the Reference Architecture, followed by two agile cycles. The reflection step (fifth in the C3A methodology, as presented in Section 4), dealing with adjustments to the Implementation Architecture, is done in proximity to the next minor release. The RA portion of the visionary research was postponed until the completion of the first IA.

Our work in the context of the new release for a mature product focuses on capturing the tacit knowledge of the RA as present within the development team, and validating it logically against the existing product. However, the level of detail in the IA is limited to the next release scope only.

Meaning, this is not a complete documentation exercise, but rather one focused on evolution. The agile Level 2 design sessions are run in a lean framework thanks to the vast experience and expertise of the team members.

The integration product requires changes to be implemented in other products as well. This mandates a detailed RA to elicit the needs and vision of the product to external stakeholders. Naturally, the IA in this case is more heavily focused on the integration and functionality layers of the product than on the system layers.

We use Magic Draw and UML to capture the architecture documents, and Word files for the one-page contracts. Each component has a separate page for better control. Later on, all separate pages are collected into a release.

The one-page contracts describe each component's current capabilities, not what needs to be done in the next release. This essentially makes their collection a reference document that evolves and matures over time, maintaining the Architecture Knowledge.

We soon found out that in order to manage this minimal level of architecture activity, some basic training was needed. Depending on the number of developers participating and their geographical location, we used one of three approaches, as outlined below.

1. We conducted a practical design workshop for the remote five-member team developing the new product, while providing on-the-job training. An expert captured the observations during the debates, and while doing so, explained the symbolic notation as well as the methodology. The construction of the RA continued in remote design activities, limited to 2 hours each.
2. For the local ten-member team working on the next release of a mature project we asked three experts to build an initial RA which was then the subject of a brief review with an external expert. We then debated the accuracy of the RA during a four-day educational session with the whole team. This was followed by the team building the IAs working in small groups.
3. The integration product team featured the most globally dispersed membership. In this case, we worked with the leading architects, conducting on-the-fly training and basic assistance, dividing the task of constructing the RA among three individuals. One captured the functional perspective, one the conceptual, and one captured the system perspective. All perspectives were combined by the system architect into the single RA.

What we learned from our experience with remote teams is the need to keep the architectures as simple as possible. The higher the number of artifacts and perspectives, the harder it is to make progress. It is difficult enough to lead a strategic agenda, which gets even more so when doing it remotely.

It was interesting to observe how the application of abstraction barriers and the four layers caused the various RA consumers to focus their attention. Because of the separation of concerns, product management leaders were primarily interested in the functional layer. They ensured each of their demands was being considered and conceptually manifested. Development managers were interested in the Level 0 components to assist them in constructing teams and owners, whereas the developers themselves were focused on the Level 1 components and the one-page artifact, detailing the exact needs and constrains of the functional and non-functional requirements of a deployable unit.

However, this narrow focusing could have distracted the individuals from seeing the overall picture. This was mitigated by periodic peer reviews when the system was evaluated, and every time the implementation architecture was changed.

Relevant maturity was reached during different timeframes. After 6 months, the integration team RA was relevantly stable but not mature, due to cross product debates and discussions, as well as additional activities extraneous to the C3A. The new product reached stable maturity of the RA after 4 months while constantly adapting the architecture based on feedback from implementing the first release. The mature product, reached maturity within 2 months, focusing on new areas of the next release, and conducting proof-of-concept projects, aiming at validating our initial assumptions.

The new product refactored its existing IA, based on a better understanding of future value, as elicited by the C3A process. The mature product team materialized three new modules and capabilities based on the analysis conducted for the entire product, focusing on external integrations.

In order to elaborate the methodology, we provided a coaching and training plan, in which the product architect builds the architecture, while being mentored by an external (experienced) architect. The team is provided with tutorials and educational materials such as papers, power point presentations and handbooks. Iterative team discussions in the form of peer reviews facilitate the design activities. While working on the architecture, the teams concurrently suggested means for adjusting the 7 steps of the methodology denoted as LOW RISK (**L**isten and **O**bserve, **W**atch, **R**eflect, **I**mprove, **S**crutinize, and **K**ick Start), thus impacting the evaluation and evolution of their architecture-centric approach.

6. Conclusions

In this paper we presented our CA Agile Architecture (C3A) methodology for bridging the gap between strategic thinking and tactical agile implementations in a development organi-

zation. It is based on Reference and Implementation Architecture diagrams, a set of one-page architecture component contracts, and an encompassing methodology to align the scheduling and granularity of all architectural activities.

C3A enables mapping of visionary research within the system architecture domain, resulting in the clear identification of its functional value and its underlying system impact. By conducting a gap analysis between the evolving Reference Architecture and snapshots of the Implementation Architecture, the stakeholders can track their respective agendas using the same blueprint, understand how far or near they are from their targets, fostering change discussions and mitigating risks.

The structure of the different C3A artifacts imposes abstraction barriers on the architecture's granularity levels, while interweaving visionary and strategic directions.

Beyond expanding this methodology to more products within our organization, as well as to the community in large, the experience we gained gave us some ideas on directions for further exploration. These include the modeling of non-functional requirements affecting the RA architecture, customization constraints regarding product deployment, as well as means to provide inner integrations with wrapped products.

Consequently, based on the modular structure of C3A, its evaluation and evolution process, and the ability to adapt its lean C3A artifacts, the teams can effectively implement an agile architecture without being derailed by changing strategic needs.

7. References

[1] Alexander, L.; Beck K**, "**Point/Counterpoint", Software, IEEE, Volume 24, Issue 2, March-April 2007 Page(s):62 - 65

[2] Albin S.T., "The Art of Software Architecture: Design Methods and Techniques", Wiley; ISBN-10: 0471228869, 2003

[3] Booch, G., The Economics of Architecture-First", Software, IEEE, Volume 24, Issue 5, Sept.-Oct. 2007 Page(s):18 – 20

[4] Booch, G. "The Irrelevance of Architecture", Software, IEEE, Volume 24, Issue 3, May-June 2007 Page(s):10 - 11

[5] Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R., Stafford J., "Documenting Software Architectures: Views and Beyond", Addison-Wesley, ISBN-10: 0201703726, 2002

[6] Hadar E. and Hadar I., "Effective Preparation for Design Review - Using UML Arrow Checklist Leveraged on the Gurus' Knowledge", International Conference on Object Oriented Programming, Systems, Languages and Applications, OOPSLA 2007, Montreal Canada, October 21-25, 2007

[7] Hadar E. and Perreira M., "Web Services Variation Façade – Domain Specific Reference Architecture for Increasing Integration Usability", IEEE International Conference on Web Services (ICWS 2007), Salt-Lake City, July 2007

[8] Northrop L.M., Clements P.C., A Framework for Software Product Line Practice, Version 5.0, from http://www.sei.cmu.edu/productlines/framework.html, extracted on March 17, 2007.

[9] Magic Draw Modeling tool, No Magic software company, http://magicdraw.com/, referenced on July 27, 2008.

[10] Structure 101 Modeling tool, headways software, http://www.headwaysoftware.com/products/structure101/index.php, referenced on July 27, 2008.

[11] Reporting tool from business objects, an SAP company, http://www.businessobjects.com/ referenced on July 27, 2008.