Abstract

Maintenance of complex business applications is challenging for software services industry. The maintenance team inherits the software with little design and implementation knowledge. The client-facing team gathers an ad-hoc architectural description of some sort and communicates the same to the geographically distributed maintenance team through informal box and line diagrams. This information is poorly understood, and the underlying architectural constraints are never enforced. This paper proposes a type system to model the architecture of a complex enterprise IT system using Acme architecture description language and reports a modeling approach to capture various architectural design decisions architects perform as a part of the architecture review. An initial field-study to evaluate the usefulness of such modeling has been encouraging.

1. Introduction

Software service industry is constantly faced with the challenge of managing software development and maintenance across geographies. In order to develop and maintain applications using a distributed team, a small client-facing team understands the application details and communicates the information to a larger group who are typically not collocated with the client facing team. An important part of the information which is architecture of the application is described using informal box and line diagrams. In many cases, there are multiple ad-hoc documents and spreadsheets to capture application knowledge. The maintenance team continues to maintain and enhance the application over the years but seldom updates the associated documents- thereby making these documents out-of-sync very soon. The maintenance process framework mostly ensures on-time scheduling and completion of activities- leaving the verification of completeness and accuracy of these activities to the human experts. To illustrate the point, we consider two scenarios. An important maintenance project activity is the on-boarding of a new (typically inexperienced) staff who needs to understand the system within a specified period of time. In most of the cases, the system understanding is poor and inaccurate due to incompleteness, inaccuracy and ambiguity in the documents that the new staff uses. To address this problem, the process enforces the experienced architects to be tightly engaged with the newcomers during on-boarding.

In another scenario, when the system undergoes a change, it is left to the team-members to review whether the design for the change has followed the architectural decisions and constraints. The process compliance report can hardly identify accuracy and completeness of this review.

Specifically we observe the following problems:

a) **Wastage of time**: While communicating the architecture, an ad-hoc and ambiguous representation (spread across multiple documents) is followed. As a result the system is poorly understood and the mentor wastes a lot of time due to a repeated dissemination of the application knowledge.

b) **Loss of knowledge**: There is a constant flow of people without any provision of capturing the application knowledge learnt by the outgoing person.

c) **Lack of enforcement**: Due to lack of formal framework, architects are not forced to describe the design decisions and constraints. Thus this information mostly remains tacit. Architectural constraints assumed in the initial design are eventually not enforced as the system evolves.

d) **Poor reuse**: The underlying technical issues or design decisions for business applications are often repetitive. The ad-hoc and ambiguous documentation often leads to poor reuse and every project team reinvents the approach/design constraints.

Intrigued with this observation we have undertaken a task to formally model the architecture of several complex business applications which are beset with the above problems using Acme architecture description language. In this paper we describe the modeling approach. Contributions of our paper are threefold. First, we have proposed a new type system to characterize various architectural characteristics of a business application using *Acme architectural style*. Since most of the business applications often have similar architectural characteristics, such a type system would possibly be reusable across application domains. Second, we have collected a set of design constraints that architects typically enforces in an ad-hoc manner and formally modeled them as a rules using a first order predicate
logic. Finally, we provide architecture model of a suite of business applications under maintenance to illustrate the usage of the type system and the design rules.

The paper has been organized as follows. Section 2 analyses a few reasons behind the practitioners not using a formal modeling approach. Section 3 deals with the existing modeling approaches for software architecture. Section 4 describes the new type system and design constraints using the notion of Acme family. Using the new family, we then describe a model of a family of large business applications in Section 5. Finally Section 6 concludes the paper.

2. Problem Analysis

In spite of significant research in the area of software architecture modeling, practitioners are yet to adopt such formalism in the mainstream development or maintenance activity. Informal interviews with about 15 technical architects across different projects have revealed the following difficulties in adopting the architecture modeling:

1. **Lack of standard architecture modeling tools** – The architects involved in application development are rarely aware of other modeling languages except UML.

2. **Lack of Architecture Modeling Support in UML** – Even though UML is a de-facto standard as a modeling tool, it lacks the support for architecture representation as discussed later in this paper. Consequently, the architects rarely use this tool to model architecture.

3. **Knowledge Transfer Process** - The guidelines for practitioners to create architecture documents are very abstract. The process does not emphasize on the rigor necessary for documenting software architecture. Consequently practitioners a) do not appreciate the need for documenting the software architecture and b) choose to create the architecture document at a sufficiently high level so that there is hardly any need to change the document even when the underlying code changes. As a result, interpretation of the architecture is largely left to the individual maintenance person.

Even though the application architecture is not captured in detail due to the aforesaid difficulties, the maintenance team needs this information many times for on-boarding of new members, performing impact analysis, reverse engineering, fault detection and regression testing. Specifically, the maintenance team faces the following challenges all the time:

1. How to reduce dependence on expert architect and get a new member familiar with the architecture quickly.

2. Given a change made to a component, how quickly one can identify the optimal set of component-level test cases that must be rerun.

3. If a change request comes, what is the minimal set of components that must be inspected during the impact analysis?

These challenges are addressed today by leveraging the years of experiential knowledge of the architects. A modeling of the architecture with sufficient details would essentially document the expert architect’s tacit knowledge formally, accurately and unambiguously. We believe that the modeling can be an extremely valuable artifact to help addressing all of the above challenges.

3. Software Architecture Modeling- Existing works

Generally speaking, there have been two approaches to model system architecture. The first approach uses a class of architecture description languages (ADL). The second approach uses UML notations for architecture description. An ADL can be thought of as a type of domain specific languages (DSL) to model various architectural characteristics of a software system. The modeling language supports five core entities namely component, connector, system, properties and style [10]. The well known ADLs such as C2 [1], Wright [12], Darwin [2], Acme [8] and xADL [17] support modeling of these entities at different levels of granularity.

Though all the ADLs subscribe to the generic philosophy of component-connector based modeling, there are wide variations among ADLs [10]. For instance, C2[1] describes the system dynamics based on event notification. Wright [12] proposes a CSP based formalism to define the architecture behavior. Furthermore, these ADLs are monolithic and static. This means that most of these ADLs (except Acme and xADL) can’t be extended if required. However, if one has to provide tool support for architectural design and analysis, it is absolute essential to tailor or extend the language/tool capability for the application domain.

UML [9], which can also be thought of as another DSL, has become the de-facto language to describe design aspects. UML tools provide the visual notations to capture the design which in turn improves the ease of modeling to a great extent. While UML is very successful in modeling the implementation details, questions remain if UML is the right language for architecture description. It has been argued in [10][15][7] that UML, even UML 2.0 do not provide adequate support to describe an architecture as interconnection of coarse grained components where it is
possible to model hierarchical decomposition, various communication aspects, architecture styles and so on.

4. Proposed Modeling Approach

There has been a renewed interest in resurrecting ADL for architecture modeling. Specifically, Acme, the ADL developed at Carnegie Mellon University; has become popular to define the architecture description information. We have used Acme as the language for our purpose. Before getting into the details of the proposed modeling approach using Acme, we briefly summarize the core concepts in Acme for brevity. A more detailed description of the language may be found elsewhere [8].

4.1 Acme Core Concepts

Like any other ADL, component, connector, system, hierarchical decomposition, properties and style are first class modeling entities in Acme. Components in Acme model computational elements at any level of granularity- such as functional components, databases, GUI element and so on. A connector models communication among these components. Connectors may be binary, such as pipes and client-server interactions, or N-ary. Each component has a set of ports. A port models the interface of that component, through which it interacts with other components (via connectors). A system in Acme is a connected graph of components and connectors that describes architectural structure of the system. Acme allows a component or a connector to be hierarchically decomposed into their constituent subcomponents. The hierarchical decomposition is called "Representation" in Acme.

The property of a component or a connector is an additional attribute. An attribute can be of a built-in type or a user-defined type.

Lastly, one can define a new Family which is synonymous to architecture style. A Family is a mechanism of characterizing a class of software architectures- e.g. multi-tier architecture, pipe-and-filter and so on. In Acme, a Family allows one to define a new semantically rich type system with rules. The notion of Family in Acme makes the language lightweight and extensible- domain specific variability can be addressed by creating a new Family for the particular domain and then describing the architecture of software using the Family. It is also possible to define Family specific analysis and other automation tools. Acme also allows heuristics and invariants linked to a Family using predicate logic. This observation has led us to use Acme for modeling the business application described in the paper.

4.2 Proposed Architecture Family

A complex business application typically implements one or more business processes and lives in an environment with other existing applications. The application often follows a web based, multi-tiered distributed system architecture with business information stored in data stores. The application needs to be associated with several non-functional requirements namely performance, scalability, security and so on. Interactions among the constituent components of the application and external system interactions are typically multi-protocol, not just an in-memory simple API calls. To model these characteristics we define a new architecture family- called BizAppFamily as shown in Figure 1.

![Figure 1 BizAppFamily: Component and Port Types](image)

4.2.1. Component and Port Types

We have defined a new component type called "BizSubSystem" to model an application (or a subsystem or a fine grained component) that implements one more business processes. This component type is also used to model an application (or a system) that is external with respect to the current scope of work. We have defined two properties namely scenarioList and isExternal to capture these characteristics.

A BizSubSystem type component has at least two ports, namely provided and reqd. The notion of required and provided ports is well known in ADL community[10]. In our model we have added an additional property for a provided type port- which is a set of services (described subsequently) that this component exposes. A reqd type port can consume one service (from other components).
Furthermore, we have defined two special component types called BizSource and BizSink as shown in Figure 2.

![Figure 2 BizAppFamily: Special Component Types](image)

A BizSink type component requires services (from components within our domain of interest) but does not provide any example of such a component could be a GUI component. A BizSource type component provides services but does not require any such service from other components (within the domain of interest). An example of a BizSource component could be a utility service. This component can of course require services from other components, but those components are of no interest to us.

In the following subsection we describe the notion of a service.

4.2.2. Model of a Service

![Figure 3 BizAppFamily: Service Model](image)

We have defined a new property called Service to model a functional service (such as “Manage customer” by a Customer component) implemented by a component as shown in Figure 3. Essential parts of a “Service” type are:

i) Name of the service, Input parameter list, Output parameter list.

ii) ScenarioList - List of business scenario names that are associated with this service.

4.2.3. Connector Types

As shown in Figure 4, we have defined two types of connectors- one for client-server type connection and the other for peer to peer connection.

![Figure 4 BizAppFamily: New Connector types](image)

These connectors have the following properties:

- Name of Protocol Used – Remote Method Invocation, Web Service, Local Procedure Call
- Minimum Network Bandwidth
- Data Interchange format name—XML, ebXML

4.3 Proposed Architectural Constraints

While designing a system, an architect typically warns the developers not to get trapped into certain bad design practices- known as anti-patterns. One can define these anti-patterns using the Acme rules. In a maintenance scenario, one can consider these rules as constraints and check whether the architecture derived from the inherited code has violated these constraints. In this paper we illustrate some of these constraints using a generic predicate logic and set-theoretic notations instead of Acme specific notations for brevity.

Let $C$ be the set of all business components in a system and let $\text{connected}(c)$ be the set of components connected to a component $c$. Let $\text{Ports}(c)$ be the set of ports of the component $c$ and let $\text{Representations}(c)$ be a set of representations this component has. Further, let $\text{Typeof}(e)$ be a function that returns the type of a Acme modeling element $e$.

a) Godly Component – If most of the business components are dependent on the services provided by one component, this component is commonly known as a Godly component. Obviously, it is not advisable to have a Godly component in a system. An Acme rule to describe this constraint for each component is as follows:

$$\text{notGodly}(c) \equiv |\text{connected}(c)| \leq \psi_1|C|$$

where $0 \leq \psi_1 \leq 1$ is the expert defined threshold parameter.
b. Spaghetti Code – If any business component communicates with more than a given number of components, the entire system becomes unmanageable beyond certain point. At the system level one can detect a spaghetti design as follows:

\[
\text{NotSpagetti(System)} = \left|\{c \in \mathcal{C} | \text{notGodly}(c) = true\}\right| \leq \psi_2 \cdot |\mathcal{C}|
\]

where \(0 \leq \psi_2 \leq 1\) is the expert defined threshold parameter.

c. Not self-connected – For a business component, it does not make sense to consume its own service through an external connector. If a component has to invoke any of its own service, the invocation should be internal to the component. Thus, one can define an invariant rule for a component as follows:

\[
\text{notSelfConnected}(c) = c \notin \text{connected}(c)
\]

5 Architecture modeling using Proposed Style

With the help of this architecture family, we have modeled the architecture of a set of business applications which are currently under maintenance for a long time. The complete portfolio consists of more than 800 applications catering to several business domains, built over time using application middleware like J2EE and .NET and integration middleware like MQSeries and web services. This paper describes architecture model for a set of 12 applications catering to one domain. Error! Reference source not found. shows the context diagram illustrating these 12 applications.

5.1 Business Scenarios

This suite of applications implements a variety of coarse grained business scenarios as follows:

a. Identifying the right product
b. Order Management of the products
c. Billing for the orders placed
d. Provisioning of sold products
e. Service Request and Delivery of sold products
f. Customer Relationship and Account Management
g. Promotion and Product Marketing
h. Integration with third party systems

The ApplicationServices application in Error! Reference source not found. implements these scenarios and it consists of close to 3500 classes and 780KLOC. Obviously, this application (and 11 other applications) is much too large for anyone to comprehend. Therefore, it is essential to further decompose each of these components to more fine-grained subsystems. In collaboration with the maintenance team we decomposed...
Application Services - which is undergoing changes and enhancements continuously.

5.2 Hierarchical Decomposition

The Application Services component has been hierarchically decomposed into three components namely Business Services, Data Services, and Application Front End as shown in Figure 6. Each of these components has been further decomposed to help an inexperienced maintainer study the architecture at different levels of abstractions.

Figure 6 Decomposition of Application Services Component

Figure 7 illustrates decomposition of Business Service into finest grained components.

Figure 7 Decomposition of Business Service Component

5.3 Services and Connectors

The fine-grained services provided by the components in Figure 7 have been obtained by analyzing the interfaces of the Java classes, remote interfaces of EJBs and web services definition. Then these services are connected to the provided ports of the finest grained components.

The connectors were used to model various protocols. Recall that the protocol description is an important property of a connector as described in Section 4.2.3. In the present application, protocols like local procedure call, RMI, JRMP, SOAP over HTTP and JMS have been used.

As a part of the architecture the performance and scalability attributes of the services were rarely described to begin with. In the model we have augmented the relevant components with the performance and scalability attributes based on the observations on the live system.

6. Conclusion

This paper explores the use of Formal Architecture models for application maintenance and enhancement teams. The preliminary investigations on the use of formal architecture models for application maintenance teams indicates that the maintenance team can improve the overall on-boarding time for the newcomers and reduce the dependence on human experts. The architecture model can bring standardization for the client team while capturing the architecture. This in turn can help in quantifying the completeness of the architecture. Use of architecture family and associated rules as constraints can help in architecture enforcement in a more formal way.

The directions for future research include automated architecture recovery from codebase, automated impact analysis for change requests, use case execution on formal models and analysis of non functional attributes like performance and scalability while designing enhancements.

References


