# Architecture and Design Intent in Component & COTS Based Systems

***Dewayne E Perry and Paul S Grisham***
*Empirical Software Engineering Laboratory (ESEL)*
*ECE, The University of Texas at Austin*
{perry,grisham}@ece.utexas.edu

## Abstract

*Architecture and design intent are critical elements in the development and evolution of software systems. They are critical in two ways. First, there must be a shared understanding of them to adequately and effectively build and evolve our systems. Second, this shared understanding is needed to coordinate the various developers and teams of developers, especially in evolving our systems. The lack of access to internal implementation details makes the issue of architecture and design intent even more critical in COTS and component based systems. We explore the issues involved in supporting the reification and use of architecture and design intent, discuss a selection of approaches, and present some ideas we have about its use in both planned and agile contexts.*

## 1. Introduction

In creating software systems we make choices throughout the entire development process with certain intent in mind. We select objects and processes from the problem domain and exclude others because we have a certain intent as to the focus of the problem we want to solve. In our specification of the requirements we try to capture the intent we have in mind relative to the problem we want to solve. We choose one architecture over another because of some intent that favors the one over the other. We use specific algorithms and data structures because of specific design intent; we use specific representations for similar intentional reasons.

When we create a product or a component, we also have an idea of how we intend it to be used and express that intent in the documentation as well as in the interface descriptions. Our intent may be specific or general relative to the product or component. Certainly when we use a specific product or component we do so intentionally. Moreover, our intent is often directed to only a part of what the product or component provides. Seldom do we rely on the entire functionality of that component or product. The mere fact of use only conveys a small part of our intent.

This problem of intent is compounded when we use COTS or other components where the internal details are unavailable for use in the discovery process. In these cases, architectural mismatch [11] poses a significant problem. There are often conflicting intents relative to the control and use of resources, how interactions are controlled and managed, what the nature of the global architecture is, and how things are to be constructed and in what order.

And, of course, in evolving our systems and components we are dependent on understanding the original intentions that led to the current state that we need to change. Indeed, we spend as much as 80% of our time in discovery or rediscovery in legacy systems [9]. Much of this time is spent trying to determine the original intent of the architecture, design and code.

In one person or very small group developments, understanding issues of intent rarely pose much of a problem. However, even there problems can arise from when the underlying intent is forgotten or misunderstood. Most of our developments, however, are not of these very-small group-variety but range from tens to hundreds to thousands (in extremely large projects) of developers.

Coordination of multiple developers, then, is a fundamental problem we face when creating or maintaining a software system. Software engineering involves developing correctly functioning software, despite uncertain requirements, languages and tools that do not always relate to the problem domain, and an environment with conflicting priorities, policies, viewpoints, and expectations for the product. Technology, market forces, and the problem domain constantly evolve, leading to changes in the underlying assumptions behind the software development.

Software developers and managers rely on many technologies and processes to manage change. However, the core problem is how to capture, express, and utilize intent. Intent is critical in coordinating a team of developers so that team development is choreographed effectively and economically. Intent is critical in evolving a software system from one release to another so that the original intent is maintained while new intentions are added, or so that only the appropriate intent changes where the system needs to be corrected or improved. Intent also facilitates reuse of system assets, such as code modules or components, by allowing developers to compare the context of the original intent of the asset to the current problem. Without this necessary coordinating intent, we set ourselves up for failure in system creation and evolution.

Traditionally, intention has been conveyed by means of documentation artifacts, such as system requirements, architecture, design, code, test documentation, and user documentation. Documentation is often voluminous, ambiguous, incomplete and out of date. For instance, Brooks' report [8] on the OS360 project described how 6 months of the project workbook measured 5 feet and daily change distributions averaged 2 inches. In one release of AT&T/Lucent's 5ESS system [28] 11.8% of the design and implementation faults were due to ambiguous requirements and design. An additional 30.6% of the design and implementation faults were due to incomplete or omitted requirements or design. That is, 42.4% of the root causes of design and implementation faults were ambiguity and incompleteness – traditional problems of documentation.

The primary benefit of documentation is that it provides a shared model of intent. Requirements documents provide a shared model of the problem [16] to be solved and what the customer wants. Architecture documents provide a shared model of the basic structure of the solution (the machine), the constraints on the various components and their interactions, etc. Design and code documentation provide shared models of the machine in greater detail. These shared models are what provide the coordination mechanism in building and evolving software systems.In creating software systems we make choices throughout the entire development process with certain intent in mind. We select objects and processes from the problem domain and exclude others because we have a certain intent as to the focus of the problem we want to solve. In our specification of the requirements we try to capture the intent we have in mind relative to the problem we want to solve. We choose one architecture over another because of some intent that favors the one over the other. We use specific algorithms and data structures because of specific design intent; we use specific representations for similar intentional reasons.

When we create a product or a component, we also have an idea of how we intend it to be used and express that intent in the documentation as well as in the interface descriptions. Our intent may be specific or general relative to the product or component. Certainly when we use a specific product or component we do so intentionally. Moreover, our intent is often directed to only a part of what the product or component provides. Seldom do we rely on the entire functionality of that component or product. The mere fact of use only conveys a small part of our intent.

This problem of intent is compounded when we use COTS or other components where the internal details are unavailable for use in the discovery process. In these cases, architectural mismatch [11] poses a significant problem. There are often conflicting intents relative to the control and use of resources, how interactions are controlled and managed, what the nature of the global architecture is, and how things are to be constructed and in what order.

And, of course, in evolving our systems and components we are dependent on understanding the original intentions that led to the current state that we need to change. Indeed, we spend as much as 80% of our time in discovery or rediscovery in legacy systems [9]. Much of this time is spent trying to determine the original intent of the architecture, design and code.

In one person or very small group developments, understanding issues of intent rarely pose much of a problem. However, even there problems can arise from when the underlying intent is forgotten or misunderstood. Most of our developments, however, are not of these very-small group-variety but range from tens to hundreds to thousands (in extremely large projects) of developers.

Coordination of multiple developers, then, is a fundamental problem we face when creating or maintaining a software system. Software engineering involves developing correctly functioning software, despite uncertain requirements, languages and tools that do not always relate to the problem domain, and an environment with conflicting priorities, policies, viewpoints, and expectations for the product. Technology, market forces, and the problem domain constantly evolve, leading to changes in the underlying assumptions behind the software development.

Software developers and managers rely on many technologies and processes to manage change.

However, the core problem is how to capture, express, and utilize intent. Intent is critical in coordinating a team of developers so that team development is choreographed effectively and economically. Intent is critical in evolving a software system from one release to another so that the original intent is maintained while new intentions are added, or so that only the appropriate intent changes where the system needs to be corrected or improved. Intent also facilitates reuse of system assets, such as code modules or components, by allowing developers to compare the context of the original intent of the asset to the current problem. Without this necessary coordinating intent, we set ourselves up for failure in system creation and evolution.

Traditionally, intention has been conveyed by means of documentation artifacts, such as system requirements, architecture, design, code, test documentation, and user documentation. Documentation is often voluminous, ambiguous, incomplete and out of date. For instance, Brooks' report [8] on the OS360 project described how 6 months of the project workbook measured 5 feet and daily change distributions averaged 2 inches. In one release of AT&T/Lucent's 5ESS system [28] 11.8% of the design and implementation faults were due to ambiguous requirements and design. An additional 30.6% of the design and implementation faults were due to incomplete or omitted requirements or design. That is, 42.4% of the root causes of design and implementation faults were ambiguity and incompleteness – traditional problems of documentation.

The primary benefit of documentation is that it provides a shared model of intent. Requirements documents provide a shared model of the problem [16] to be solved and what the customer wants. Architecture documents provide a shared model of the basic structure of the solution (the machine), the constraints on the various components and their interactions, etc. Design and code documentation provide shared models of the machine in greater detail. These shared models are what provide the coordination mechanism in building and evolving software systems.

## 2. Managing Evolutionary Systems

The kinds of systems we are generally interested in are evolutionary systems. Lehman & Belady [21] tell us that these systems are in a constant state of change. Requirements change, technologies change, and the systems themselves change the operating environments they were originally designed to operate within change. For any reasonably sized problem, the system has evolved at least once before the initial version of the system can be deployed.

During the traditional software development cycle, requirements are reified into an architecture which influences the design and eventually guides the coding process. In practice, it is often only the code that is kept current with respect to the state of the problem domain. Code is the desiccated relic of a long decision-making process during which various design trade-offs and decisions about functional and non-functional requirements are made. Unfortunately, in the process of abstracting actual requirements into models and converting those models into code, only the end result of the design process is reflected in the code, leaving intent to be represented in the external documentation. It is difficult, if not impossible, to reconstruct the thought process which generated the resulting system from the requirements. As a consequence it is not clear how requirements changes impact the system. As system requirements become more complex, we look for ways to manage these changes and deliver high-quality, high-value systems to our customers.

## 3. Modeling Design Rationale and Intent

An early approach to capturing design rationale and intent is to be found in the Potts and Bruns [30] generic model for delineating the generic elements of a design rationale. These elements include artifacts, issues, alternatives, justifications and the relationships among them. A design deliberation is represented by an issue, a set of alternatives and a justification for the determined decision. This deliberation process begins with an initial design represented as an artifact, which raises one or more issues about the evolving design. These issues lead to a discussion of various alternatives, one of which is selected on the basis of a justification and which yields a new or evolved artifact. The process iterates until all issues have been resolved and all the necessary design artifacts have been created and reached a stable state. The result is a design history that can be used as the basis for evolving the design as needed by changing requirements, etc.

While the Perry/Wolf [29] software architecture model explicitly (first in 1989, about the same time as the work of Potts and Bruns mentioned above) called for rationale in addition to elements and form, the focus of research has been primarily architecture description languages to describe components and form.

An exception to this is the work of Gruenbacher, Egyed and Medovitch [12, 13]. Support for architecture decisions is introduced in their CBSP (Component, Bus, System, Property) model to bridge the gap between requirements and architectures using intermediate models. The intermediate CBSP model captures architectural decision in terms of CBSP dimensions: components (C), bus/connectors (B), system-wide features (S), component properties (CP), bus properties (BP), and system or subsystem properties (SP). CBSP provides a lightweight way of transforming requirements into architectures "using a small but extensible set of key architectural concepts" as well as a high degree of control over this transformation process. For example, bus properties include: synchronous, asynchronous, local, distributed, and secure. Given these bus properties we can characterize their usefulness with respect to various well-known styles. For example, synchronous connectors provide extensive support for a client service style but virtually no support for a pipe and filter style.

More recently, Bosch [4] has lamented the general lack of support for architecture rationale. Among the problems he sees that need to be solved are the following: design decisions are not first class entities; design decision are often cross cutting and intertwined; design rules are easily violated; obsolete design decisions and their artifacts are rarely removed; and high maintenance costs result because of these problems. He then claims that we should, as a community, "take the next step and adopt the perspective that that a software architecture is, fundamentally, a composition of architectural design decisions." Dueñas and Capilla [10] responded to this exhortation by proposing a "set of elements, information and graphical notations to record the decisions during the modeling process" and thus "detail the idea of considering the architecture as a composition of architectural design decisions."

## 4. Intent in the Face of Change and Uncertainty

One of the major problems in trying to capture and maintain architecture and design decisions and intent is the context of uncertainty. No matter how hard we try to keep our development context constant and without change, change and the resulting uncertainty is a fundamental fact of development life. Indeed, change and uncertainty are interdependent, each causing the other.

Requirements uncertainty and change often have far reaching effects, especially if they occur in or persist until the later stages of a development project. Technology changes may have significant impact on developments, at times rendering them obsolete while at other times making their development simpler by an order of magnitude. Environmental and business changes can create significant uncertainty and further change as well.

While late binding is a superb technique to create dynamically adaptable systems, the uncertainty of deferred design decisions can cause significant problems if not handled and managed well. And it is unfortunately the case that we often have to delay decisions until we find a useful rationale for resolving those deferred decisions.

Thus, we as architects and designers face significant uncertainty and change in the process of attempting to build and evolve a stable product. The methods, techniques, processes and tools needed to support our design decisions and convey our architectural and design intent need to be robust and usable in the face of constant change and uncertainty. It is in this rich problem context that we want to find useful and practical solutions to ease the job of the software engineer in creating and evolving complex software systems. We believe that intent is a means that will provide this ease.

## 5. Applying Intent to Manage Evolution

We are currently looking at how to create formal and semi-formal representations of intent as a means of documenting requirements and their relationships to the resulting software system, and testing these representations in two domain areas: a traditional, planned development with architectural design, and an agile software development based on Extreme Programming. We believe that by demonstrating the efficacy of intent-based models in such disparate development domains, we can generalize our results to many other software engineering domains and processes.

We are developing a design approach called *Rationale Reification* [14] that utilizes formal models of architectural rationale to represent the architect's intent in transforming requirements into system architectures. The approach will support iterative modeling of requirements, reified rationale, and abstract (that is, prescriptive) architectures. We are using ontological models of intent and developing tool support for visualizing and configuring architectural components with intent. This approach allows

designers to identify emergent changes and reuse elements from requirements, rationale, and architectures.

We believe that agile software development methods can benefit greatly from intent-based requirements modeling. Our approach is a technique called *Intent-First Design* that is analogous to the approach known as *Test First Design*. Intent First design provides developers with a means of embedding comprehensible, maintainable, and lightweight requirements models into the source code. The approach uses semi-formal models of intent to capture problem domain requirements in terms of goals. Code is enriched with links between models and code assets, and tool support facilitates maintenance and validation of these links. Version management of both code and requirements is classified and organized with respect to rationale and intent changes.

# 6. Initial Approaches

## 6.1 Perry/Wolf Architecture Model

The Perry/Wolf [29] model of software architecture focused on capturing basic structural intention. The Perry/Wolf model defines a software architecture as elements, form, and rationale. Components and connectors are the basic architectural elements. Form prescribes the properties of the elements, their relationships with each other, and constraints on the elements and relationships. Rationale provides the justification for both the elements and the form. Architectural styles are the implicit mechanism by which basic aspects of intent are captured. Rationale and style are critical in managing evolutionary systems.

## 6.2 Inscape

The Inscape Environment [23-25] bases its constructive approach to managing the relationship between implementations and their interfaces on formal interface specifications and a propagation logic [26]. We use the term constructive in the sense that as each piece of the implementation is constructed, Inscape maintains the semantic interconnections based on constraint satisfaction. The formal interface specifications can be viewed as pre-conditions, post-conditions, and obligations (constraints that must be eventually satisfied to guarantee correct functionality). The basic rule about pre-conditions and obligations is that they must be satisfied within a specific implementation scope or propagated to the interface of that scope. There are scoping rules that define the construct granularity for which the propagation rules apply. Thus, intent is expressed constructively and is then reified in the satisfied and propagated preconditions and obligations.

One of Inscape's useful extensions to interface specifications was the possibility of multiple results (that is, multiple sets of post-conditions and obligations) that are often useful in representing multiple normal as well as exceptional results. Inscape incorporated a set of rules for handling these multiple results. For practical systems, exceptions are necessary in building fault tolerant and reliable systems. Implementation intent is expressed constructively by choosing the exception handling technique for the individual exception results.

Another feature of Inscape was constraint-based retrieval of components [27]. One is able to retrieve single as well as multiple components on the basis of their constraints (typically on the basis of their post-conditions since one usually is trying to satisfy unpropagated constraints in an implementation). Unification was added to the basic theorem proving substrate as a mechanism for accomplishing this task of intent-based retrieval.

## 6.3 Architectural Prescriptions

In our work on transforming software requirements into architectural prescriptions, the intent of the methods and techniques [5-7, 17-19, 32] used is to create a constraint-based architectural specification. The starting point for transforming requirements into architectural prescriptions is van Lamsweerde's KAOS goal-oriented requirements specification language [20]. KAOS's stratified goals provide a useful way of expressing multiple levels of requirements intent. We use the KAOS logical language to specify architectural intent by means of constraints in the architectural prescription language Preskiptor. An architectural prescription then is a means of expressing architectural intent by means of a set of constraints about the architecture components and connectors as well as its structure and form.

Architectural styles are a particularly important form of constraint codification. Architectural styles (as defined by [29]) are incomplete architectural prescriptions that focus on some specific components, structures, and/or constraints. These styles are then applied as constraints to components, connectors and structures. They may be applied to specific elements, collections of elements or the entire system. In summary, they capture specific architectural intent.

### 6.4 Intent-based Architectures

In our WOSS'04 paper [15], we extended current requirements engineering and prescriptive architectural approaches by introducing architectural intent as the key concept that enables the creation of abstract intent-based architectures. The intent of an architectural element encapsulates its functional purpose in unambiguous terms, so that any architectural element with a given intent may play a given role in the architecture. Intent-based architectures enable the system to be defined at higher levels of abstraction than current approaches, using a requirements-domain, or problem-domain, language, providing a direct link from the requirements to the system architecture, and enabling the same architecture to be reified by one or more functionally equivalent implementations.

Intent-based architectural prescriptions provide the basis for our design of a prototype self-configuring adaptive system that is able to respond to changing environmental or operational conditions or failures by reconfiguring itself on the basis of a high-level abstract understanding of the functional goals and non-functional constraints of the system., By utilizing the intent of each available architectural element, we build new architectural configurations that will enable the system to perform its required functionality, while conforming to any non-functional constraints (e.g., performance, security, dependability).

## 7. Using Rationale to Transform Requirements into Architectures

A key problem with current system architecture and design practices is that there is no direct connection between the requirements, the high-level design, and the implementation. Transforming a set of requirements into the architecture and design of a system is basically fundamentally a creative process. Requirements are usually captured in problem domain terms using human language, while architecture and design are defined using implementation domain constructs (e.g., classes, components, connectors, etc.). The lack of direct connection between requirements and architecture not only makes verification of architectures difficult, but also makes it difficult to incorporate requirement evolution into existing architectural designs.

The technical goal of requirements gathering is to convey to the architects, designers and developers the intent of the system under development -- that is, to define the functional purpose of the system by describing the set of real-world problems that are to be solved by the system. For the purposes this discussion, we assume that all the various system drivers such as user needs, corporate business strategies, etc., are incorporated into the requirements. However, no matter how accurately the requirements express the functional and non-functional intent of the system, the process of mapping or translating this intent into a system architecture and design continues to be problematic.

To date, software architecture research has largely focused on various aspects of elements and form; the limited research related to the role of rationale in architecture has tended toward general, informal treatments. Rationale is intended to capture the relationship between requirements and architectural prescriptions. Reifying the rationale is a critical element in realizing our general goal of intent-based systems and intent-based architecture in particular. Rationale captures the refinements and transformations used by the architects to transform requirements specifications into architectural prescriptions. They provide the formal link between requirements and architectural specifications.

This intent model of rationale reification provides the basis for systematic requirements-based evolution, where changing the requirements (i.e., the intent of the system) lead to changes to the architectural rationale, and associated changes in the system architecture. This means that in rationale-based architecture, the requirements are a directly connected and primary source for the system architecture (along with the rationale derived from the requirements), throughout the useful life of the system. This approach is different from the usual situation where the requirements are usually maintained (if at all) separately from the system itself.

Rationale reification, then, is an approach that uses architectural rationale to transform requirements into an architecture. The rationale determines the mapping from a set of functional and non-functional requirements to an abstract architecture taking into account functional requirements and their interrelations, non-functional requirements, and relations between non-functional and functional requirements. The abstract architecture is defined in terms of the problem domain terminology of the requirements, and is a model of the functional intent of the system as expressed by the requirements. The abstract elements in the architecture are implemented by one or more concrete architectural elements. Like the requirements themselves, this concrete architecture is related to the abstract architecture according to intent.

A rationale model represents a formalization of the mapping from requirements to system architecture. Rationale reification depends on requirements analysis, in which the system requirements are divided into functional goals and non-functional constraints, and iteratively refined into discrete units of functionality [33]. The refinement process results in stratified goal hierarchies, in which higher-level (coarser-grained) goals are decomposed into lower-level (finer-grained) goals. Rationale reification relies on this kind of refinement approach to transform high-level requirements into functional units that are at the right level of functional granularity to be mapped across to existing or newly designed components.

In rationale-based architecture, the rationale model encapsulates both the semantics and conditions for all the mappings and transformations from requirements to architecture, and the reasons for making each transformation. This enables every functional and non-functional requirement to be traced to one or more architectural elements, and also enables every architectural element to be traced backwards to one or more requirements. And all the rules and reasons for architectural mappings and transformations may also be viewed and refined as needed.

Tool support is an important factor in enabling software architects and engineers to fully leverage the benefits of using rationale-based architectures on a daily basis to design and evolve software systems. Specific tools include a requirements modeler, that will enable developers to create, view, and refine the requirements goals, constraints, and interrelations among requirements; a rationale modeler that will enable developers to create, view, and refine mappings and transformations from the requirements model to a system architecture; and an architecture modeler, enabling developers to edit, view, and refine the architectural model. Finally, an intent modeling and visualization tool will enable developers to view, edit, and extend ontologically rich models of functional intent. Ultimately, to enhance support for flexible cross-platform implementation configurations based on abstract rationale-based architectures, as well as provide better support for self-configuring systems, additional tools such as a component intent classification tool would also be desirable to make it easier for developers to build the necessary metadata models to support implementation configurations for rationale-based architectures.

## 8. Agile Software Development Environments

Agile software development techniques provide means for developing software systems in the presence of changing or uncertain requirements [1]. We consider agility to refer to *how* a development activity responds to change and evolution. There are many strategies for responding to such changes: feature-oriented milestones, short iterations with frequent deliveries, close interactions with the customer, and deferring design decisions as late as possible. Instead of emphasizing process-supporting activities, developers are encouraged to actively resolve requirements uncertainties through working portions of software. Studies suggest that that the sooner in the development process the customer can provide feedback, the better the product will be in terms of both customer satisfaction and quality [22].

In one of the most popular agile software development approaches, Extreme Programming (XP) [3], requirements are captured in terms of acceptance and unit test cases that are written by, or with the assistance of, the customer representative, an approach referred to as Test-First Design [2]. Test cases are written before code in order to ensure that the new requirement is not already satisfied by the current state of the implementation. Code is written to meet the minimum needs of the new requirement. As requirements change, test cases are added or modified. As code and design change, these test cases can ensure that the original requirements are met.

These test cases provide a view into the requirements of an ongoing, evolving project. Unlike requirements captured formally in a notebook somewhere, these requirements are living, active artifacts of the requirements gathering and development process. Unfortunately, maintaining test cases over an evolving project in the absence of information about design intent is just as difficult as maintaining any other artifact. Unit tests are not semantically rich enough to capture these design decisions. Perhaps even more importantly to an agile development effort, the relationship between implementation artifacts and rationale can inform developers about which goals and intentions are still valid, and which have been abandoned or changed as the project evolves.

We believe that agile approaches are an ideal environment for a semi-formal intent annotation method because they require lightweight, maintainable documentation of requirements that offer long-lasting benefit without burdening developers with extra process tasks. Because many agile software developers are already incorporating principles of Test-First Design, we believe that if the intentional model is sufficiently comprehensible and the tools sufficiently

usable, then Intent-First Design should meld easily with any agile process.

Moreover, intent, expressed in terms of goals, is an appropriate abstraction for ensuring quality of test cases. Goal and intent modeling can help maintain the evolution of both code artifacts and requirement artifacts (*i.e.*, the test cases.) Binding intent with code elements can yield positive benefits in terms of productivity, quality, and maintainability of software systems.

We are following the concept of the "programmer's assistant" [31] that can provide interactive feedback on how to model intent and write code that meets the requirements specified in our intent model. This work is largely an extension to the previous work on the Inscape development environment, both with respect to how Inscape facilitated capturing the semantics of code, but also with respect to how Inscape emphasized coordination of software development teams.

The basic environment for Intent-First Design is an integrated development environment (IDE) with a language aware editor, plug-in tools to handle the management and evolution of the intent model, automation support for testing and validation, and integration with version management. Tool support for agile software development with intent is critical for entering and visualizing the intent-based requirements for the system, as well as providing a means of communicating those requirements to the rest of the team.

Changes to the requirements model, code, or test cases flag revision notices in the intent model to inform the developer of potential consistency problems between code and intent model. In this way, the intent model, and consequently the requirements, will be kept up to date with the current state of development.

In order to navigate the wealth of new information available to the developer and project manager, the IDE must provide several views into the code. The *code view* is view into the traditional program editor environment, annotated with intent. Intent can be displayed explicitly, or abstracted through graphical visual cues. The *intent view* displays a more comprehensive view of the state of the requirements as represented by use stories, features, non-functional goals, or whatever requirements abstraction is appropriate to the developer's process. Code elements are abstracted down to modules, objects, or whatever partitioning method the code and programming language support. Code and intent views are easily navigable through hypertext links.

The *status view* presents the intent model with respect to the current level of implementation and correctness. In the status view, user requirements can be prioritized and assigned. The *change view* is tied to version management and gives a view of the requirements and the code in terms of volatility. In the change view, code and requirements changes can be expressed in terms of intent, which helps identify uncertain requirements or unstable code modules. Intent provides a meaningful abstraction for talking about the status of the ongoing development effort, and for placing current efforts in context.

## 9. Conclusions

Architecture and design intent are a critical elements in both creating and evolving software systems. Without a shared understanding of intent it is all to easy to introduce faults in the software system and create failures in the development processes. This problem is exacerbated in the context of COTS and other components that must be treated as black boxes. In the context of custom components, we can spend time (re)discovering and (re)constructing architectural and design intent using internal details. However, in the context of COTS components we have less to go on and hence must have more explicit descriptions to enable us to use these components both correctly and effectively.

We have explored some of the issues involved in supporting the reification and use of architecture and design intent, discussed a selection of approaches, and presented some ideas we have about its use in both planned and agile contexts.

## 10. Acknowledgements

Some of the work presented here is exploratory work we have done with Matthew Hawthorne as part of his preparation for his PhD thesis proposal [14], in particular that of architecture rationale reification.

## 11. References

[1] The Agile Alliance. Manifesto for Agile Software Development. 2001. http://www.agilemanifesto.org

[2] Beck, K. *Test-Driven Development by Example*. Addison-Wesley, 2003.

[3] Beck, K. and Andres, C. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2004.

[4] Bosch, J., Software Architecture: The Next Step. In *Proceedings of the First European Workshop on Software Architecture (EWSA*

*2004)*, (St. Andrews, UK, May 21-22, 2004), 194-199.

[5] Brandozzi, M. From Goal Oriented Requirements to Architecture Prescriptions, Master's Thesis, Department of Electrical and Computer Engineering, The University of Texas at Austin, Supervisor: Perry, D.E., 2001.

[6] Brandozzi, M. and Perry, D.E., Transforming Goal Oriented Requirement Specifications into Architectural Prescriptions. In *Proceedings of the Software Requirements to Architectures Workshop (STRAW'01)*, (Toronto, ON, Canada, May 14, 2001), 54-60.

[7] Brandozzi, M. and Perry, D.E., Architectural Prescriptions for Dependable Systems. In *Proceedings of the Proceedings of the International Workshop on Architecting Dependable Systems (WADS 2002)*, (Orlando FL, May 25, 2002), 25-29.

[8] Brooks, F. *The Mythical Man Month: Anniversary Edition*. Addison-Wesley, 1995.

[9] Davison, J.W., Mancl, D.M. and Opdyke, W.F. Understanding and Addressing the Essential Costs of Evolving Systems. *Bell Labs Technical Journal*, *5* (2), August, 2000. 44-54.

[10] Dueñas, J.C. and Capilla, R., The Decision View of Software Architecture. In *Proceedings of the 2nd European Workshop on Software Architecture (EWSA 2005)*, (Pisa, Italy, June 13-14, 2005), 222-230.

[11] Garlan, D., Allen, R. and Ockerbloom, J. Architectural Mismatch: Why Reuse is So Hard. *IEEE Software*, *12* (6), November, 1995. 17-26.

[12] Grunbacher, P., Egyed, A. and Medvidovic, N., Reconciling Software Requirements and Architectures: The CBSP Approach". In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, (Toronto, ON, Canada, August 27-31, 2001). IEEE, 202-211.

[13] Grunbacher, P., Egyed, A. and Medvidovic, N. Reconciling Software Requirements and Architectures with Intent Modeling. *Software and Systems Modeling*, *3* (3), August, 2004. 235-253.

[14] Hawthorne, M.J. The Rationale Reification Approach to Architectural Design in Software-Based Systems, Dissertation Proposal, Department of Electrical and Computer Engineering, The University of Texas at Austin, Supervisor: Perry, D.E., In Progress.

[15] Hawthorne, M.J. and Perry, D.E., Exploiting Architectural Prescriptions for Self-Adaptive, Self-Managing Systems: A Position Paper. In *Proceedings of the Workshop on Self-Managed Systems (WOSS'04)*, (Newport Beach CA, October 31 - November 01, 2004).

[16] Jackson, M., The World and the Machine (Keynote Address). In *Proceedings of the 17th International Conference on Software Engineering (ICSE 17)*, (Seattle WA, April 23-30, 1995).

[17] Jani, D. Deriving Architecture Specifications from Goal Oriented Requirement Specifications, Department of Electrical and Computer Engineering, The University of Texas at Austin, Supervisor: Austin, T.U.o.T.a., 2004.

[18] Jani, D., Vanderveken, D. and Perry, D.E., Deriving Architectural Specifications from KAOS Specifications: A Research Case Study. In *Proceedings of the 2nd European Workshop on Software Architecture (EWSA 2005)*, (Pisa Italy, June13-14, 2005).

[19] Jani, D., Vanderverken, D. and Perry, D.E. Experience Report: Deriving Architecture Specifications from KAOS Specifications. The University of Texas at Austin, 2003. http://www.ece.utexas.edu/~perry/work/papers/R2A-ER.pdf

[20] van Lamsweerde, A. and Willemet, L. Inferring Declarative Requirements Specifications from Operational Scenarios. *IEEE Transactions on Software Enginering*, *24* (12), December 1998. 1089-1114.

[21] Lehman, M.M. and Belady, L.A. *Program Evolution: Processes of Software Change*. Academic Press, 1985.

[22] MacCormack, A., Verganti, R. and Iansiti, M. Developing Products on 'Internet Time': The Anatomy of a Flexible Development Process. *Journal of Management Science*, *47* (1), January 2001. 133-150.

[23] Perry, D.E., Software Interconnection Models. In *Proceedings of the 9th International Conference on Software Engineering (ICSE 9)*, (Monterey, CA, March 30 - April 2, 1987), 61-69.

[24] Perry, D.E., Version Control in the Inscape Environment. In *Proceedings of the 9th International Conference on Software Engineering (ICSE 9)*, (Monterey, CA, March 30 - April 2, 1987), 142-149.

[25] Perry, D.E., The Inscape Environment. In *Proceedings of the 11th International Conference on Software Engineering (ICSE 11)*, (Pittsburgh, PA, May 15-18, 1989), 2-11.

[26] Perry, D.E., The Logic of Propagation in The Inscape Environment. In *Proceedings of the SIGSOFT '89: Testing, Analysis and Verification Symposium*, (Key West FL, December 13 - 15, 1989), 114-121.

[27] Perry, D.E. and Popovich, S.S., Inquire: Predicate Based Use and Reuse. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference (KBSE)*, (Chicago IL, September 20-23, 1993), 144-151.

[28] Perry, D.E. and Stieg, C.S., Software Faults in Evolving a Large, Real-Time System: a Case Study. In *Proceedings of the 4th European Software Engineering Conference (ESEC'93)*, (Garmisch-Partenkirchen Germany, September 13-17, 1993), 48-67.

[29] Perry, D.E. and Wolf, A.L. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, *17* (4), October 1992. 40-52.

[30] Potts, C. and Bruns, G., Recording the Reasons for Design Decisions. In *Proceedings of the 10th International Conference on Software Engineering (ICSE 10)*, (Singapore, April 11-15, 1988), 418-427.

[31] Rich, C. and Waters, R.C. *The Programmer's Apprentice*. Addison-Wesley, 1990.

[32] Vanderveken, D. Deriving Architectural Descriptions from Goal-Oriented Requirements, Master's Thesis, Departement d'Ingenierie Informatique, Universite Catholique de Louvain, Supervisors: van Lamsweerde, A. and Perry, D.E., 2004.

[33] Vanderveken, D., van Lamsweerde, A., Perry, D.E. and Ponsard, C. Deriving Architectural Descriptions from Goal-Oriented Requirements Models. 2005. http://www.ece.utexas.edu/~perry/work/papers/R2A-05-damien.pdf