Proceedings of the ASME 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference
IDETC/CIE 2007
September 4-7, 2007, Las Vegas, Nevada, USA

DETC2007-34761

STUDYING THE DYNAMICS OF THE ARCHITECTURE OF SOFTWARE PRODUCTS

Manuel E. Sosa[1]
Assistant Professor of Technology and Operations Management
INSEAD
Fontainebleau, FRANCE
manuel.sosa@insead.edu

Tyson Browning
Assistant Professor of Enterprise Operations
Neeley School of Business
Texas Christian University
Fort Worth, Texas
t.browning@tcu.edu

Jürgen Mihm
Assistant Professor of Technology and Operations Management
INSEAD
Fontainebleau, FRANCE
jurgen.mihm@insead.edu

ABSTRACT

This paper reports on an exploratory study of how the architecture of a software product evolves over time. Because software is embedded in many of today's complex products, and it is prone to relatively rapid change, it is instructive to study software architecture evolution for general insights into product design. We use metrics to capture the intrinsic complexity of software architectures as they evolve through successive generations (version releases). We introduce a set of product representations and metrics that take into account two important features used to manage the complexity in software products: layers and modules. We also capture organizational data associated with the product under development. We propose a three-step approach for the analysis and illustrate it using successive versions of an open source product, Ant. One of our findings is that software architectures seem to evolve in a non-linear manner similar to the S-shaped curve that characterizes technology evolution at the industry level. We also find several parallel patterns among architectural and organizational dynamics. Implications for research and practice are discussed.

Keywords: product architecture, modularity, complexity, organizational structure, software development.

INTRODUCTION

Previous work has studied the implications of product architecture decisions to various aspects of the firm [e.g., 1, 2, 3]. Yet, little attention has been paid to understanding how product architectures evolve over time. How does the architecture of a product evolve over several generations? That is the central question we address in this paper. We propose a theoretical framework and research approach to study the dynamics of complex product architectures. We illustrate our approach by examining the architecture of software products because they are complex, exhibit fast change rates (like fruit flies in studies of biological evolution), and offer (through their source code) an efficient, reliable, and standardized medium to capture their architecture.

Studying how system architectures are established and how they evolve over time is important to help organizations cope with changes throughout the life cycle of a product. We still do not fully understand what underlying forces drive changes in the architecture of products from one generation to the next. As a result, organizations typically struggle to manage the architectural changes associated with their products [4].

Complex systems require both decomposition and integration, which in turn determines their architecture. The formal literature on product decomposition and product architecture begins with Alexander [5], who describes the design process as the decomposition of designs into minimally coupled groups. Simon [6] elaborates by suggesting that complex systems should be designed as hierarchical structures consisting of "nearly decomposable systems," such that strong interfaces occur within systems and weak interfaces occur across systems. Hence, complex software and hardware products are typically decomposed into sub-systems and components in order to be designed. Yet, such sub-systems and components then need to be integrated to ensure that the overall product functions as a whole.

In the product domain, Ulrich [2, p. 419] defines product architecture as the "scheme by which the function of a product is allocated to physical components" and by which components interact. The IEEE similarly defines product architecture as "the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and

[1] Corresponding author

evolution" [7]. Establishing the architecture of a system includes breaking down the product into functional and physical elements, mapping the functional elements onto the physical elements, and specifying the interfaces among the interacting elements [8]. In the software domain, architecting involves organizing or structuring the code into modules and layers with the appropriate set of dependencies between them [9-11].

The product architecture is typically established during the concept development phase of product development [8]. In a completely new system, the architecture is established from scratch and evolves in subsequent generations of the product. However, in most product development efforts, system architects start with the existing architecture of the product (or a similar one) and modify it to map the added functionality into the product [4]. This is particularly true in software development, where products can be developed in a more flexible and additive manner. Software products evolve rapidly by adding features and fixing bugs from one version to the next, which poses important coordination challenges for organizations that need to manage their resources carefully to cope with such changes [12].

The fast dynamic evolution typically associated with software development makes software products especially attractive for a study of how complex product architectures and organizations evolve over time. Understanding the structure (or architecture) of systems is important because "structure always affects function" [13, p. 268], yet the architecture of evolving systems is complex in different ways, which makes studying their dynamics a challenging endeavor.

This paper makes two important contributions. First, it provides a theoretical framework, a basic set of metrics, and a research approach for exploring the dynamics of complex software architectures. Second, based on empirical evidence from a case study of an open source project, we uncover several patterns and insights regarding the dynamics of software architectures and their relationships to organizational dynamics. These findings indicate several promising avenues for future research.

THEORETICAL FRAMEWORK

We argue that complexity and how designers deal with it is the key product property we need to study to gain understanding of the evolution of product architectures. What is complexity? Although previous research in varied fields has studied complex systems, neither precise definition nor specific measures of system complexity have been agreed upon [6, 13-16]. At a minimum, the common understanding seems to be that complexity is a property of a system that depends on the number of elements that form the system and the way they interact [17]. Because the architecture of products not only affects and reflects their complexity but also the mechanisms to deal with it, we focus on examining the dynamics of product complexity as a way to explore the evolution of product architectures.

Intrinsic Software Complexity

In the context of designing hardware products, Suh [16, 18] defines absolute complexity as the vector sum of two orthogonal components, real and imaginary complexity. We are interested in real complexity and will term our complexity measure *intrinsic complexity*, because it is the complexity the product intrinsically shows upon examination of the code itself.

Obviously one element of intrinsic complexity is the number of elements comprising the system. The other main driver of intrinsic complexity is the coupling among the elements. While each element in a system, to be a proper member of that system, typically requires at least one direct relationship with another element, many elements often have much more than this nominal connection to the rest of the system. The greater an element's connectedness to other elements, the greater its potential impact on the overall system (all else being equal)—e.g., the greater the cost of changing the element, as additional tests must be performed, and the greater the likelihood of constraints. Increased connectivity increases the possibility of change propagation [19-22]. Moreover, if components are connected in multiple ways, this can create loops where change feeds back, and the resulting cycles may not dissipate for some time. We call these coupled elements. Coupling dramatically increases complexity, because it implies rapid growth in the number of design conditions that must be considered and accommodated.

Dealing with Complexity: Modules and Layers

Good software architects know the value of grouping and layering as a way to manage complexity [9]. As a result, they organize and group elements into modules and organize modules into layers to form a hierarchy. While having 100 elements together might be too complicated to manage, having ten groups of ten elements—where each group makes sense from a functional, structural, or some other accepted perspective—makes things simpler, even though the intrinsic complexity of the system is the same in terms of the number of elements and interactions [23]. Two types of groups (or modules) are commonly used in software: (1) *functional groups*, where elements perform similar types of functions but have few or no direct data exchange relationships (e.g., a function library), and (2) *iterative groups*, where elements are highly coupled, to the point that it makes sense for them to be developed with close, mutual consideration. The literature on modularity has measured product modularity based on the notion that modules enclose highly interdependent components which are more loosely connected to other components in the product [24, 25]. Others suggest using the eigenstructure of the product design structure matrix (DSM) to determine the modules that are inherently more interdependent based on their patterns of interactions [26]. In this paper, we do not measure modularity explicitly, but instead measure the complexity of various aspects of the product architecture which are determined by the presence of modules and layers.

Layering is the process of arranging modules in a hierarchical manner. That is, layering imposes an authoritative hierarchy among certain groups, such that elements in a higher layer can "call" (unilaterally send outputs to or request inputs from) an element in a lower layer, while the reverse situation would violate the intent (or the rules) of the design [1, 27, 28].

A RESEARCH APPROACH TO STUDY SOFTWARE ARCHITECTURE DYNAMICS

To explore the dynamics of complex software architectures, we structure our research approach in three steps:

- Capture the evolution of software architecture properties. To do this it is important to use appropriate product architecture representations and metrics that allow us to capture the relevant aspects of both the intrinsic complexity of software products and how software architects deal with it using modules and layers. We develop two sets of metrics that capture both structural complexity and architectural evolution. We track these metrics over time over several product generations.
- Capture the evolution of organizational attributes. In addition to software architecture properties, it is also important to capture process and organizational variables associated with the establishment of the software architecture. We focus on three types of variables: 1) *Organizational workload* determined by the new functional requirements of the product; 2) *Organizational resources* dedicated to carrying the workload; and 3) *Organizational coordination* associated with the informal communication patterns among development actors during the development effort.
- Compare the dynamics of product architectures and organizational attributes. To understand what drives the dynamics of software architectures, we compare product architecture metrics and organizational attributes over time.

REPRESENTING SOFTWARE ARCHITECTURES

To measure the complexity associated with software architectures, we first need to represent how the components of the product interact, how they are grouped into modules, and how modules are organized into a hierarchy. To capture the basic features that characterize complex system architectures, we use two complementary representations: a hierarchy tree and a partitioned product DSM. A tree representation indicates module membership and layering, whereas a product DSM captures the interactions between components both within and across modules.

Figure 1 shows the tree representation of one of the versions of the software product we study in this paper, Ant 1.30. The tree representation shows how the 126 components comprising this version of the product are organized into eight modules and three layers. To define modules and layers, we first define the tree's top and bottom levels. The top level is the root node, which represents the entire product as an integrated whole. The bottom level is the "leaf" level in which one finds individual components that comprise the product. Note that where one chooses to end the decomposition and declare the lowest level is the modeler's choice. In our analysis, we stop at the "source file" level,[2] although we could go down further to the level of lines of code or even machine language. However, two main arguments led to our choice: First, source files tend to provide a set of common functionality (e.g., a set of low-level mathematical functions) and are often maintained by a single author, who designs them as one integrated piece of software. Second, the main attributes of the architecture become apparent by the "class file" level, so further decomposition would only obscure these insights. This is consistent with previous work focused on representing software architectures [e.g., 22, 27].

We define a *module* as a group of components (or other modules). Because a module can group other modules (in lower levels), a hierarchy is inevitably formed between the top and bottom levels. We distinguish two types of modules: *component modules* that cluster product components ("class files") and are defined immediately above the leaf level, and *sub-system modules* which cluster other modules (and perhaps also some individual components). For example, Figure 1 shows eight component modules and three sub-system modules ("ant", "taskdefs", and "util").

Next, we define *layers* to indicate the distance from the root level at which modules are formed. For example, the first layer in Figure 1 is formed by the three modules below the root node (layer 0). Such a tree representation has three layers because one of the modules in layer 1, "ant," is formed by four modules: two component modules, "types" and "*," which cluster 12 and 23 components, respectively, and two sub-system modules, "taskdefs" and "util," which use another layer to cluster their components into two modules each. It is important to note that lower layers have higher layer numbers.

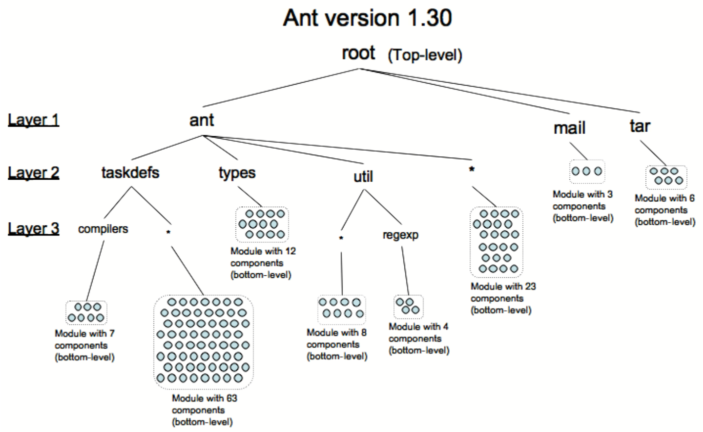

Ant version 1.30
root (Top-level)
Layer 1
ant
mail
tar
Layer 2
taskdefs
types
util
*
Layer 3
compilers
*
*
regexp
Module with 3 components (bottom-level)
Module with 6 components (bottom-level)
Module with 12 components (bottom-level)
Module with 23 components (bottom-level)
Module with 7 components (bottom-level)
Module with 63 components (bottom-level)
Module with 8 components (bottom-level)
Module with 4 components (bottom-level)

Figure 1: Tree diagram of Ant version 1.30

To represent the interactions between the components of the product, both within and across modules (and across layers), we use a product DSM representation. A DSM is a matrix

[2] In Java, files and classes are typically the same, except for "inner classes" (classes within classes), which we do not consider explicitly.

representation introduced by Steward [29], originally used to analyze task interdependencies in complex development projects [30, 31]. Such a matrix representation has also been used to capture the architecture of complex products to analyze the patterns of interactions between components in complex (hardware) products [21, 32-34]. A product DSM is a square matrix whose rows and columns are identically labeled with the product components, and whose off-diagonal cells indicate component interfaces. We use the convention where the components labeling the columns depend on the components labeling the rows.

In the software domain, a DSM representation has been used to capture the interactions between "class functions" that comprise software applications [22, 27, 28]. Typically, the rows and columns in a product DSM are ordered so as to maximize the density of clusters of components along the diagonal, so that clusters (modules) encapsulate the majority of interfaces. This approach, called *clustering* [31], is generally recommended for hardware products because of the highly symmetric nature of many spatial and structural design dependencies between physical components [32, 33]. However, when analyzing the architecture of software products, we instead use the clusters defined by the system architects and partition (triangularize) the DSM, also called *sequencing* [31], to uncover the dependencies that define the truly coupled components. To this end, the usage of a sequencing algorithm is appropriate because software products exhibit a significantly large proportion of uni-directional dependencies (e.g., "calls") between components [22, 27]. In that sense, software components are more similar to activity- and time-based processes than hardware products. Software components process inputs to produce outputs that will be used by other components in the application.

Because we want to be consistent with the convention used in software development in which software components (e.g. Java classes) are built on top of other ones that serve as platforms, we place the outputs of a component in the columns of the DSM while the inputs are placed in the rows of the DSM. As a result, a sequenced DSM is one where the rows and columns are ordered so as to minimize the number of marks above the diagonal, so that the elements sequenced first in the matrix depend on the elements sequenced last. This follows the convention of block diagrams used in software development, in which the library and utility modules are at the bottom of the diagram and serve as platform upon which other software components are built. Super-diagonal marks in the sequenced DSM represent feedback marks, where components that are supposed to provide outputs to other components also receive inputs from them. Note that our partitioning-based approach to organize the software architecture data starts with the hierarchical "clusters" (or modules) defined by the systems architects and uses the partitioning algorithm to identify the interdependent set of components within each module (within each layer). Hence, in the presence of several layers we partition each layer of the system recursively starting from layer 1. This distinguishes our approach to document software architectures from previous work, both in the product and software domains, which has typically relied on clustering algorithms or heuristics to group the elements of the system without regard to layering [22, 34]. Moreover, it is important to emphasize that our intention is simply to document the software architecture to analyze its evolution over time rather than to find the optimal architecture for a given version.

On a side note, those familiar with DSM techniques will notice two innovations here. First, we are applying a sequencing algorithm to a component-based DSM, a combination which did not exist [31] prior to the work by Sangal *et al.* [27]. Second, we reverse the typical order of dependency in the DSM. Traditionally, a DSM using the convention where the components labeling the columns depend on the components labeling the rows would show feedback *below* the diagonal. This is done because, as is conventional in software, the "higher level" components are said to depend on the "lower level" ones for functionality, and, unlike other time-based DSM applications to date, all of the components indeed exist simultaneously.

In a complex software product with several layers, like in Figure 1, we partition the DSM layer by layer so that modules within the same layer are arranged so as to minimize super-diagonal marks. (To sequence within each layer, we use the algorithm originally proposed by Steward [29].) Figure 2 shows a DSM representation of Ant 1.30. The DSM shown is a 126x126 matrix with 476 off-diagonal marks representing the "calls" between the 126 "classes" that comprise Ant 1.30. The DSM is sequenced by layer so that feedback marks above the diagonal are minimized both within and across modules. This DSM has 12 marks above the diagonal, six of them in layer 2 within module ("ant"—"*") and six of them across modules (four within layer 2 and two within layer 3. Note that the branches of the tree in Figure 1 are arranged to correspond to the sequenced DSM. The branches on the left of the tree depend on the branches on the right.

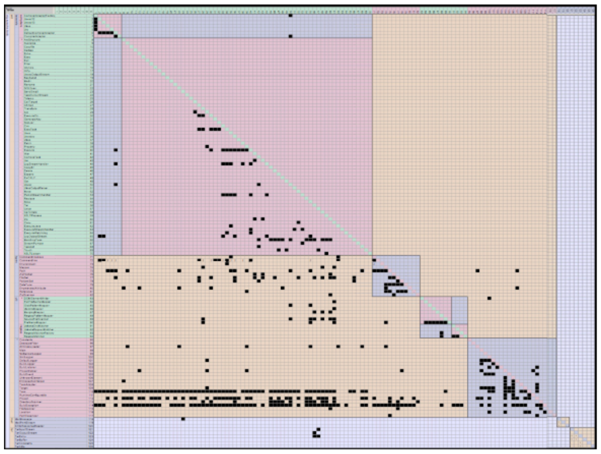


Figure 2: Complete DSM for Ant 130

METRICS TO STUDY SOFTWARE ARCHITECTURES

We develop two types of metrics to study the dynamics of software architectures: (1) *Structural architectural metrics*, which measure various aspects of software complexity for each version of the product, and which we track over several product versions; and (2) *Dynamic architectural metrics*, which capture architectural changes from one version to the next.

Structural architectural metrics

We measure intrinsic complexity as a function of the number of product components and the number of interactions among them. Here, we disregard the hierarchical organization of modules, treating interactions within a layer or module and across layers and modules the same. Then, we take into account how system architects manage intrinsic complexity with layers and modules.

Intrinsic Complexity

At a basic level, intrinsic product complexity is a function of the number of elements (n) and the interactions between the elements (k) [6]. The simplest metric of complexity is just $C_1 = n*k$. From there, the metrics become more sophisticated. For instance, metrics should take into account that feed-forward and feedback interactions contribute differently to the complexity of the product. One way to consider that begins by simply separating the contribution of these two types of interactions as: $C_1 = n*k = n*(k_{feed\text{-}fwd} + k_{feedback})$.

We developed an alternative metric of system complexity, C_2, which accounts for the amplifying effects of feedback interactions and coupled components. The details of this metric are included in a separate appendix available from the authors upon request. However, for initial simplicity, we base our analyses on C_1. (In the case of the Ant product, which has a relatively small number of feedback interactions, the general results are fairly consistent with the ones obtained using C_2.)

Note that C_1 assumes that complexity grows proportionally to both the number of elements and number of interactions in the system; it is not normalized against system size. This is intentional, because at this point we simply want to apply a general expression of complexity that allows us to measure complexity in various regions inside the product. Normalizing against size would take away this dimension of complexity.

Obviously, capturing the notion of complexity into one single measure is an abstraction from reality. Thus, one might question how good our measure for capturing the complexity of a system is, beyond its obvious face value. Since there is no agreed upon measure of complexity, ultimately this question can only be answered by relating alternative complexity measures of a system with meaningful process or organizational attributes to test which measure has better predictive value. In that sense it is an empirical question upon which we shed light in the next section.

The Effects of Modules and Layers

Because interactions among components are not randomly distributed within the product, but instead are clustered within modules, which in turn are organized into layers, we measure complexity considering the interaction of components within the modules themselves and then across modules per each layer. As before, we distinguish between the complexity associated with feed-forward and feedback interactions.

Internal complexity of component modules:

To measure the average internal complexity of the modules that cluster product components, we consider each module above the "bottom level" (or leaf level) as an isolated system and determine its internal complexity by considering feed-forward and feedback interactions separately. Then, we take a simple average of these values across the number of component modules present in the product. (Weighting the average by the number of components in each module provides similar results.) Note that the strength of the interaction between two components is "1" when there is at least one function call from the first component to the second, and zero otherwise. Hence,

$$\textit{Avg module internal feed forward complexity} = \frac{\sum_{module} n_{module} \bullet k_{module\ feedforward}}{number\_of\_modules}$$

$$\textit{Avg module internal feedback complexity} = \frac{\sum_{module} n_{module} \bullet k_{module\ feedback}}{number\_of\_modules}$$

Cross-module complexity

To determine the complexity associated with the interactions between components across modules, we calculate feed-forward and feedback average complexities for each layer. To do this, we estimate the strength of the interaction between two modules by the density of the cross-boundary interaction. That is, if one module (formed by two components) interacts with another module (formed by three components), then there are six possible ways in which one module can interact with the other module. The strength of the interaction is therefore the number of actual interactions divided by the potential number of interactions. In this case, if there were two actual interactions, then the strength of the cross-module interaction would be 0.33. Finally, to measure the complexity of the layer, we consider it as a system with m modules and interactions equal to the summation of the cross-module interactions (ρ). Hence,

$$\textit{feed forward cross-module complexity} = m \bullet \sum_{interactions} \rho_{feedforward}$$

$$\textit{feedback cross-module complexity} = m \bullet \sum_{interactions} \rho_{feedback}$$

Our measures differ from previous metrics used to characterize product architecture. Similar to Guo and Gershenson [24], we recognize that it is important to distinguish between interactions within and across modules. Yet, their aggregated metric is based on the notion of density, while ours are built on the notion of complexity described above. In the software domain, MacCormack *et al.* [22] suggest using a measure of propagation cost associated with a product architecture, based on the reachability matrix [14, 21]. Although such a metric is informative for assessing how changes in one component can potentially propagate to other components in the product (and we show it for comparison in the table of data in the Appendix), it does not appropriately capture the complexity dynamics of a system because it is normalized against the size of the system. As mentioned, normalizing against system size suppresses an important aspect of complexity that may itself pose important organizational coordination challenges. For example, we expect to observe different coordination effort between an organization that develops a fully connected product with a dozen components and an organization that develops a fully connected product with hundreds of components.

Dynamic architectural metrics

Dynamic metrics capture the architectural change relative to the previous version of the product. This set of metrics is designed to explore how software designers structure their work and more importantly how that structure changes over time. Especially, we want to be able to detect reorganizations of code (often called refactoring [35]).

The hierarchical structure of a software product is embedded in the tree as represented in Figure 1. Hence, our metrics need to characterize the essence of the tree. Our most prevalent static measure, N_x, is the total number of subsystem and component modules (nodes of the tree above the leaf level) in the tree the designers used to structure the code for version x of the product. Note that this measure can be used recursively for different subsections of the entire tree.

We define $D_{e,x}$, $D_{a,x}$ and $D_{t,x}$ as the evolution metrics for N_x. To measure $D_{e,x,}$ we consider that a node has been *eliminated* if in the following version the node is not part of the same layer. Note that eliminations include shifting a node to a lower or higher layer. E_x is the number of eliminations from version x-1 to version x. To measure $D_{a,x}$ we consider that a node has been *added* if the node does not exist on a given layer in version x-1 but does exist for the version x. Again note that additions include shifts from a lower or higher layer. A_x is the number of additions from version x-1 to version x. Finally, $D_{t,x}$ measures the total changes due to both additions and eliminations from version x-1 to version x. Hence, we can define evolution metrics as follows:

$$D_{e,x} = \frac{E_x}{N_x}\ ; \qquad D_{a,x} = \frac{A_x}{N_x}\ ; \qquad D_{t,x} = \frac{A_x + E_x}{N_x}$$

Treating elimination and addition separately allows for differentiating between simple growth and active reorganizations of code. Making the measures relative takes into account that what defines a major reorganization depends on the existing architecture of the product. All measures can be defined for each individual layer.

Other metrics could characterize the tree. Depth and average breadth would be among the obvious candidates. The distribution of depth per "leaf" and the distribution of breadth per node would be more complicated measures. Alternative evolution measures include "minimal number of moves" with moves either being a shift in layer, an addition, or an elimination. We prefer the measures above for their simplicity and ease of interpretation, on one hand, and their explanatory power on the other.

How informative are these static and dynamic architectural metrics? Again, this is an empirical question. We will consider them to be informative if they capture the variation in software architecture across product versions so that they help us interpret product changes. They will be even more insightful if we can relate such variation to product, process, or organizational performance indicators. That is what we investigate next for the case of one software product, *Ant.*

ANT: AN EXAMPLE FROM OPEN SOURCE SOFTWARE DEVELOPMENT

About *Ant*

We studied a readily-accessible, open-source software application called Apache Ant. Ant is a Java-based tool for automating software build processes. Further information is available at www.apache.org.

To get information about Ant's product architecture, we used the openly available source code of the product. The first version, 1.0, was released in July 2000, and six major releases have followed, with additional minor releases in between. To capture the architecture of the software product, we used a commercial software application which builds a partitioned DSM representation of the software architecture [27]. The "bottom-level" of components is defined by the source files or "classes," and the interactions (or design dependencies) are function calls. Hence, if the source code for class A references the source code for class B, then the designer of A might need to be aware of what class B does. The membership of the components in layers and modules is captured in the source code via the naming convention. That is, the naming of the classes reflects not only the unique identifier of the class but also the module and branch in the product hierarchy to which it belongs. Hence, we are able to objectively and automatically capture the product components, their interactions, and their organization into modules and layers.

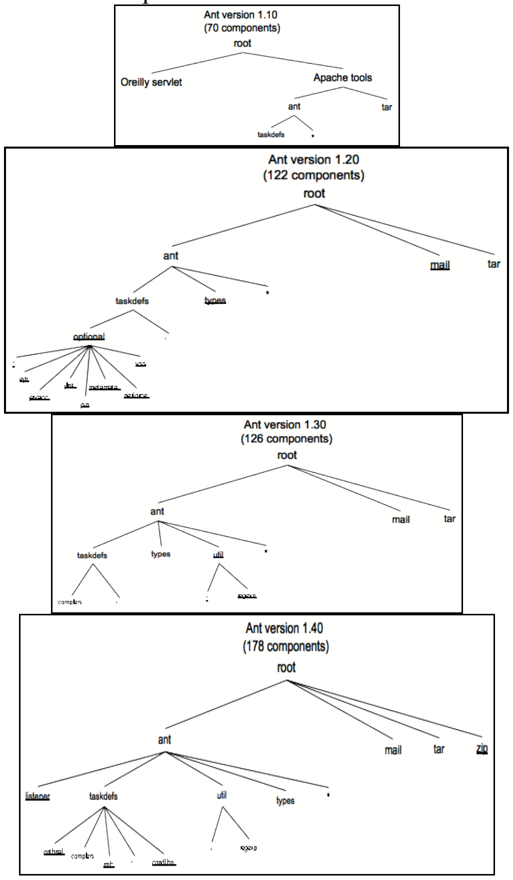
From an organizational viewpoint, Ant is developed by volunteers [36], who fall into three categories: users, developers, and committers. Users provide feedback to developers in the form of bug reports and feature suggestions. Developers contribute time, code, documentation, or resources. A developer that makes sustained, welcome contributions may be invited to become a committer. Committers are responsible for Ant's technical management. All committers have write access to Ant's source repositories. Committers may cast binding votes on any technical discussion regarding the project (www.apache.org).

The committers involved with each version of Ant are listed in its documentation. The other developers are not. To find an approximate list of developers, we scanned most of the e-mails in the developer e-mail archive over the time interval from version *x*-1 to version *x*. This method provided us with a larger list of contributors than could be found in the documentation, and it also allowed us to count the approximate number of e-mails associated with each developer.

Software Product Data

In this subsection we report the product data associated with each version of Ant. We first illustrate the hierarchy of the product data via a tree representation. Then we use DSMs to show the interaction data (at layer 2).

The plots included in Figure 3 illustrate the evolution in the product hierarchy of Ant. The first three versions of the product exhibit the most significant changes. In version 1.10, Ant is part of a bigger system that includes "Apache tools" and "Oreilly servlet." In version 1.20, Ant becomes a standalone application complemented with other utilities such as "mail" and "tar." Version 1.20 does not show a dominant architecture for the salient module, "ant." The dominant architecture appears to emerge in version 1.30 and continues to grow in the subsequent versions.



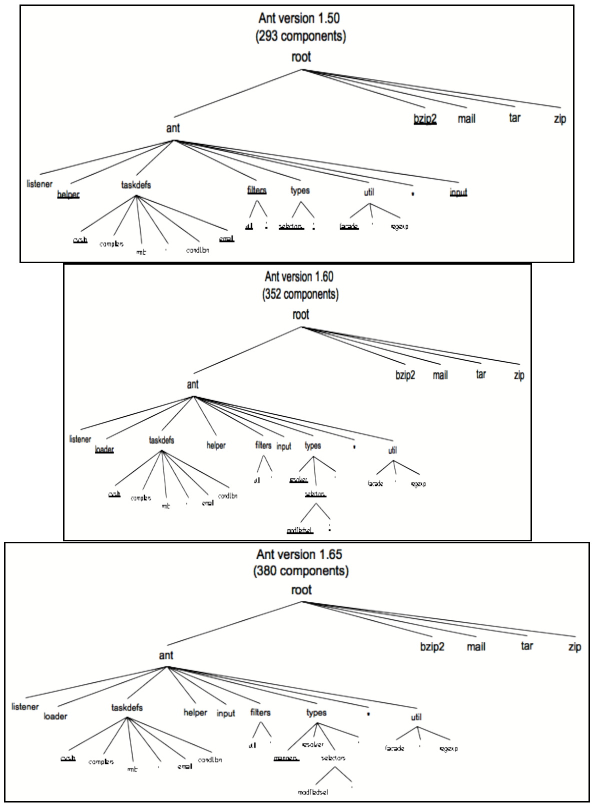
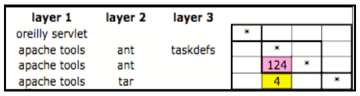


Figure 3: Tree representations of versions of Ant

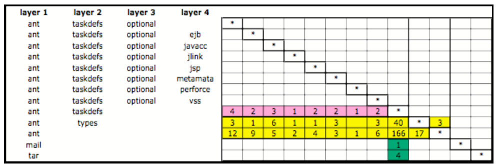
The DSMs in Figure 4 show the interactions across component modules for each version. Note that these DSMs show only the interactions across modules for each layer. The cross-module interactions in layer 1 are shown by the green cells, in layer 2 by the yellow cells, in layer 3 by the rose cells, and by the blue cells in layer 4 (where present). The DSMs are sequenced, so we can distinguish between feed-forward and feedback interactions by examining the marks below and above the diagonal, respectively. By inspection, the majority of the cross-module interactions occur in layer 2, which is formed by the modules comprising the "ant" module. The interactions at layer 1 represent the interactions between "ant" and the other utility modules such as "mail" and "tar." The interactions at layer 3 provide a finer level of granularity within modules of "ant," such as "taskdefs" and "util."

Ant 1.10



layer 1	layer 2	layer 3				
oreilly servlet			*			
apache tools	ant	taskdefs		*		
apache tools	ant			124	*	
apache tools	tar			4		*

Ant 1.20



layer 1	layer 2	layer 3	layer 4													
ant	taskdefs	optional		*												
ant	taskdefs	optional	ejb		*											
ant	taskdefs	optional	javacc			*										
ant	taskdefs	optional	jlink				*									
ant	taskdefs	optional	jsp					*								
ant	taskdefs	optional	metamata						*							
ant	taskdefs	optional	perforce							*						
ant	taskdefs	optional	vss								*					
ant	taskdefs			4	2	3	1	2	2	1	2	*				
ant	types			3	1	6	1	1	3		3	40	*	3		
ant				12	9	5	2	4	3	1	6	166	17	*		
mail												1			*	
tar												4				*

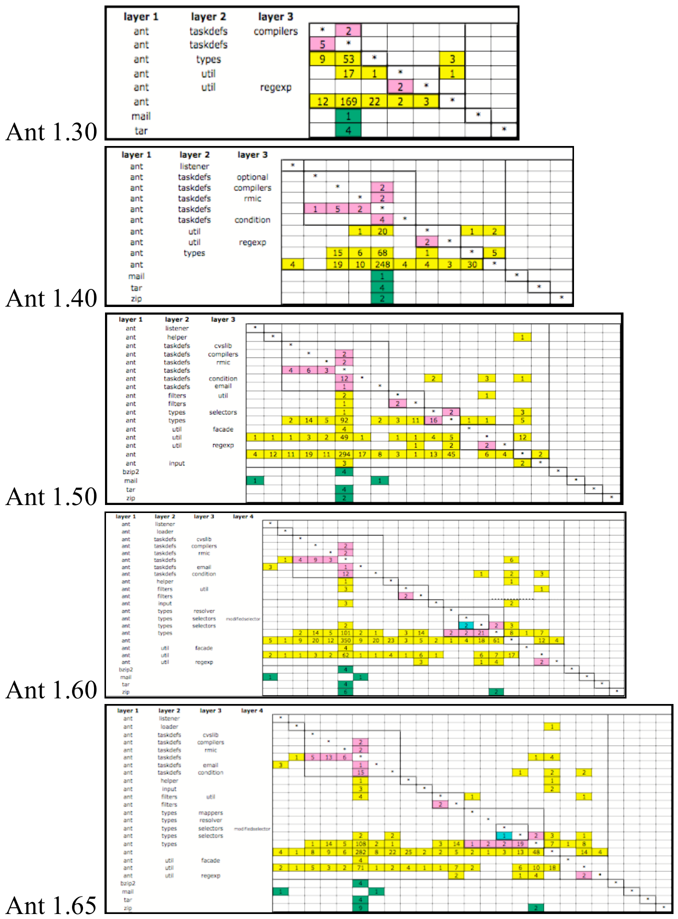


Figure 4: DSM representations of versions of Ant

Organizational Data

As described above, Ant is developed by volunteers forming a kind of virtual team. We gathered organizational data along three dimensions: workload, resources, and coordination effort.

The workload associated with *Ant*:

For each version, workers made up to three types of modifications from the previous version: "changes," "tasks," and "bug fixes." *Changes*, as the term is used by Apache Ant, signify a new and better way of implementing an existing feature or capability of the software. That is, the feature was not necessarily performing incorrectly (which would be a bug), but someone found a way to provide the feature more efficiently or effectively—or, the way the feature was implemented had to be adjusted to accommodate some other change, bug fix, or task. *Tasks* are new features or capabilities added to the current version. *Bug fixes* are corrections of existing features that were not performing correctly. Bug fixes range from major to almost unnoticeable. Only the ones deemed significant enough by the developers were actually documented and counted.

Resources used to carry out the workload: Number of developers

As open source software, Ant's number of developers is determined by their volunteerism, which is of course determined by many personal factors such as available time, enthusiasm, interest, etc. Conventional wisdom would also suggest that the way the software architecture handles the complexity of the product through modules and layers has some influence here, since it affects the learning curve for a new developer. Most new developers start by contributing to a small, localized portion of Ant, so the product architecture is probably not immediately apparent to many of them. However, as their proposed changes and fixes require checks against more and more other files and modules (because of a large number of dependencies), they could become discouraged by the high coordination costs.

Coordination effort: E-mail communication

The primary coordination mechanism for Ant's virtual development team is e-mail. There are two main e-mail forums, called the user list and the developer list. The user list is primarily to collect user inputs and feedback, while the developer list is where developers and committers exchange ideas, plans, and votes on changes, bug fixes, and tasks. We therefore focused e-mail counts on the developer list.

Table 1 (included in the appendix) summarizes the product and organizational data relevant for our analysis

ANALYSIS

We performed two types of analysis. First, we analyze the evolution of software architectures by examining how product complexity and the product architecture change over time. Then, we compare these software architecture dynamics with the evolution of the organizational attributes. Because of the exploratory nature of our work at this point, our analysis is focused on uncovering patterns of dynamics rather than statistical inference.

Software Architecture Dynamics

In this subsection we plot various measures of product complexity over time. The horizontal axis represents the number of days from the first product release. Figure 5a shows the evolution of the overall complexity of Ant measured by C_1. To distinguish between feed-forward and feedback product complexity, Figure 5b plots the ratio of feedback to feed-forward interactions for each version.

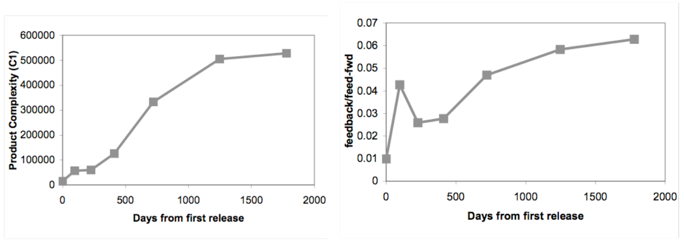


Figure 5: Product complexity: (a) C_1 (b) Ratio feedback/feed-forward

To evaluate how developers manage complexity using modules and layers, we plot the average number of components per module (Figure 6a) and the average number of modules per layer (Figure 6b). These plots illustrate the "volatile" nature of the architectural changes in the first few versions before a dominant architecture emerges. After version 1.30 the use of modules and layers stabilizes.

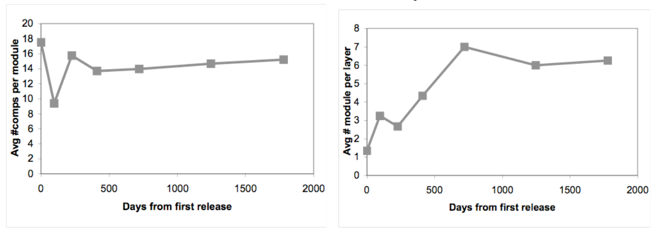


Figure 6: (a) Average number of components per module and (b) Average number of modules per layer

Because interactions between product components are not randomly distributed, but instead are organized into modules and layers, we measure complexity in various areas of the product to take such an organization into account. Figure 7 shows the average internal complexity of component-modules (both feed-forward and feedback). Although feed-forward complexity does not show a clear pattern, feedback complexity shows a significant increase in later versions, which suggests that software architectures increase their use of "shortcuts" and "design rule" violations, as indicated by the presence of feedback interactions within component modules. (Fowler *et al.* refer to this phenomenon as "software decay" [35, p. xvi].)

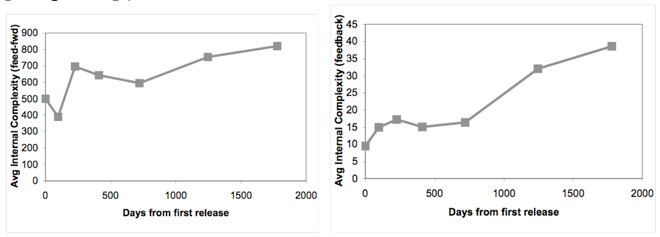


Figure 7: Average internal component-module complexity: (a) Feed-forward (b) Feedback

To take into account the effects of layers, we measure C_1 separately at layers 2 and 3. Such measures capture the complexity associated with interactions across sub-system modules. Figure 8 shows the value of complexity (both feed-forward and feedback) for layers 2 and 3, respectively. The complexity at layer 2 increases significantly until version 1.50 and then decreases.

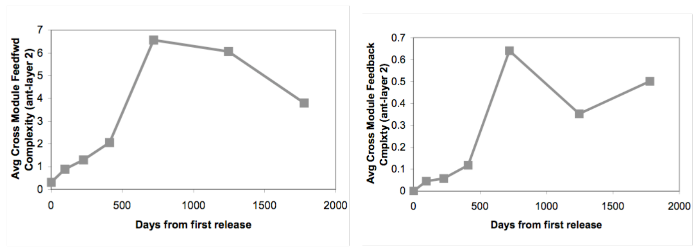


Figure 8: Cross-module complexity in layer 2: (a) Feed-forward (b) Feedback

Looking at layer 3, Figure 9 plots C_1 for the largest sub-system module of "ant," called "taskdefs." Since "taskdefs" was a component-module in version 1.10, the plots start after version 1.20. The complexity of "taskdefs" increases significantly after version 1.30.

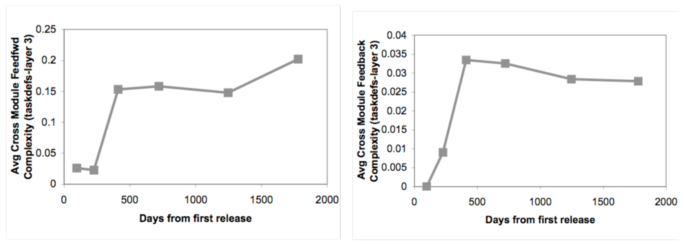


Figure 9: Cross-module complexity in layer 3: (a) Feed-forward (b) Feedback

Figure 7 examines the evolution of the internal complexity of component modules (both feed-forward and feedback complexity, but note the different *y*-axis scales). Figure 7b shows how component modules become more interdependent over time as the number of feedback dependencies increased the coupling of their components. Figures 8 and 9 show the evolution of complexity of the most relevant sub-system modules of Ant at layers 2 and 3, respectively. In both cases, a significant increase in complexity is observed once the dominant architecture is established in version 1.30. Moreover, when examining the complexity of sub-system "ant" (at layer 2), a peak in complexity is achieved in version 1.50, after which the subsequent versions show a decline in complexity (Figure 8). That is, it seems developers focused on reducing the dependencies across the modules that form "ant." Moreover, such a reduction in complexity is not accompanied by reducing the complexity of "taskdefs" (at layer 3), which remains relatively stable after version 1.50 (see Figure 9), but instead is achieved by removing dependencies across modules (at layer 2) and by adding new components to existing modules rather than creating new ones.

Finally, we explicitly analyze how designers deal with complexity by monitoring the changes in the module and layer structure from one version to the next. First, Figure 10 plots the number of modules (N_x) and thus shows the evolution of the total number of modules in the code. As with the intrinsic complexity in Figure 5a, the overall tendency seems to represent continuous growth, which conforms with the general belief that an architecture starts out simple and then grows to become more and more complex. (The local peak at version 1.20 is due to the fact that the former version used code from a different branch of the apache tree that later became incorporated into the "ant" tree. This artificially increased the number of modules in version 1.20.)

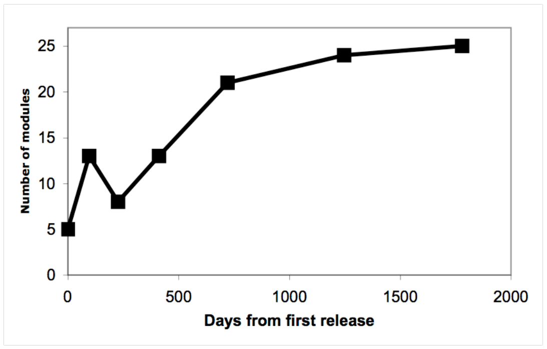


Figure 10: Number of component modules

Figures 11 and 12 reveal that the dynamics of this evolution are more complicated than the overall growth in modules might suggest. Figure 11 plots metrics $D_{a,x}$, $D_{e,x}$ and $D_{t,x}$ for the modules of Ant for versions under analysis. Clearly, the earlier versions are characterized not only by additions in the architecture but also by eliminations of some modules. Figure 12 plots the same metrics for the components (or leaves of the product tree), and interestingly the pattern is mirrored. Taking both results into account, it becomes apparent that the shift from version 1.20 to version 1.30 of the code is characterized by a major reorganization in which not only virtually the entire tree is eliminated, but a major portion of the leaf code is redone. From then on the code base mainly grows with the shift from version 1.40 to 1.50 marking a minor reorganization. In that sense the pure growth model needs to be modified. For Ant, code evolved with growth and reorganizations interacting.

These plots provide the basis for some interesting (albeit tentative) insights about the development of software products like Ant. First, Figure 5 shows that the intrinsic complexity of Ant and the efforts of designers to deal with it through code structuring increase over time in a non-linear manner following an S-shaped curve. This suggests that the architecture of software products follows a cycle similar to the one followed by new technologies at the industry level [37, 38]. Figure 6a, 11 and 12 suggest that the dominant version for Ant is established after version 1.30. Two aspects of the analysis lead us to draw this conclusion. Figure 6a shows that after the dominant architecture is established, the average size of modules remains relatively stable. Hence, as new components are added to the product, new modules are created to maintain the average size of modules. Figures 11 and 12 suggest that the architects, while coding, found that the original architecture had its limitations and needed to be redone. They used the first versions as learning vehicles that provided them with insights to establish the final architecture in version 1.30 so that they could build the functionality thereafter – with version 1.50 introducing a minor revision. Hence, these dynamic metrics allow for a differentiated evaluation of code evolution.

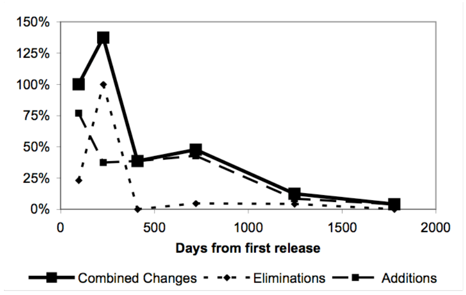


Figure 11 Changes in modules between versions

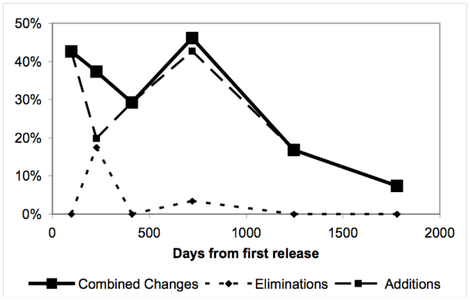


Figure 12: Changes in leaf nodes between versions

Comparing Product and Organizational Dynamics

How do software architecture dynamics compare with organizational dynamics? We address this question in this subsection. The following plots summarize the organizational attributes associated with each major release. Figure 13 shows the workload associated with both "product changes and tasks" (i.e. number of product improvements and new features) and the number of "documented bug fixes."

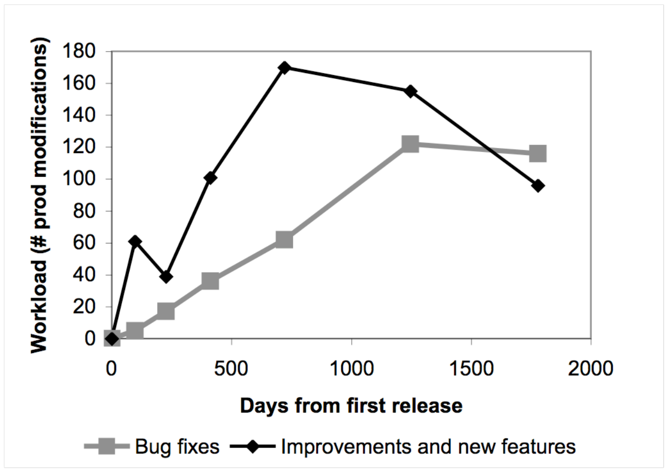
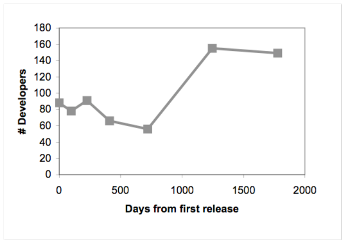
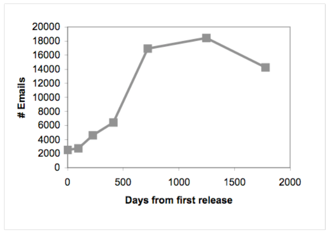


Figure 13: Workload associated with Ant

Figure 14 shows the amount of resources and the coordination effort dedicated to Ant development. We capture this by plotting the number of developers involved, the number of e-mails exchanged on each release, and the number of e-mails per developer. Note that the number of "product improvements and new features" (Figure 13) and the "number of e-mails" exhibit correlated patterns, suggesting that the coordination effort is significantly associated with the added features rather than the bug fixing. Yet, it would be interesting to explore further what proportion of e-mails were associated with bug fixing versus intended product changes.





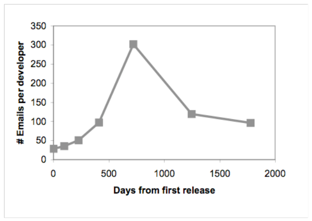


Figure 14: (a) # of developers; (b) # of e-mails; (c) # e-mails per developer

To better compare architectural and organizational metrics, we overlap product complexity metrics and organizational metrics in the same plot. Figure 15 shows how product complexity (C_1) and the overall workload exhibit similar patterns. Interestingly, there is a strong association between the complexity across modules at layer 2 and "product changes" (Figure 16).

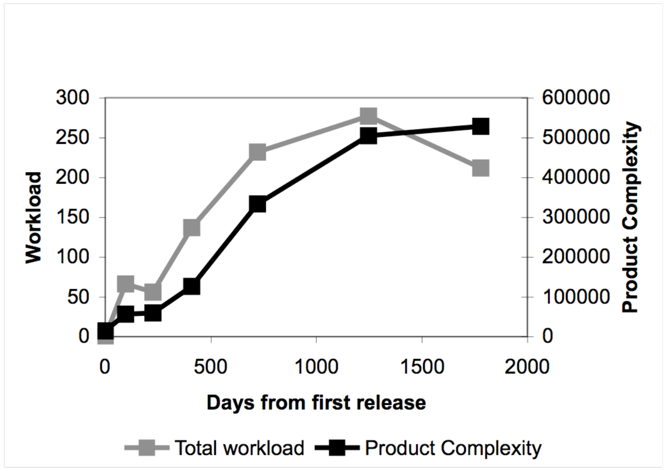


Figure 15: Comparing overall workload and product complexity

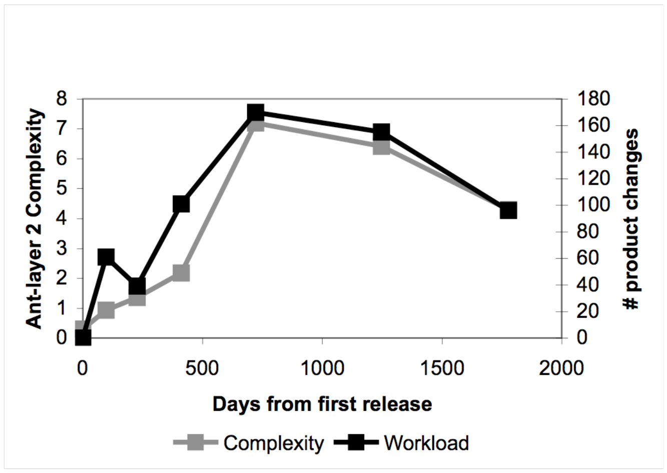


Figure 16: Comparing "ant" module complexity and workload

We also compare product complexity and coordination effort by comparing the complexity of the most relevant module ("ant") and the number of "e-mails per developer." Figure 17 shows the similar evolution of these two metrics.

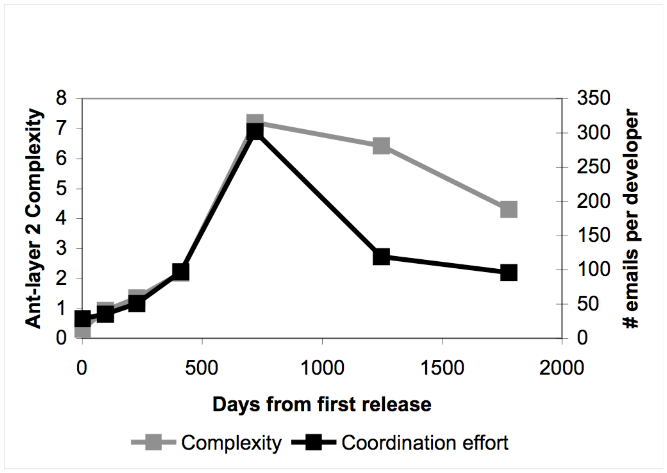


Figure 17: Comparing product complexity and coordination effort

Comparing the evolution of product complexity metrics and organizational features such as workload and coordination effort leads to important insights. First, there is a significant relationship between the dynamics of the product architecture and the organization. Product changes and bug fixes increase in a way similar to the complexity of the overall product (Figure 15). Moreover, when examining product complexity at a finer level of granularity, we found that the most informative level of analysis (for the case of Ant) is layer 2, which is the one that contains the most important sub-system module of the product ("ant"). Figure 16 shows the strong association between the evolution of "ant" complexity and "product changes," and Figure 17 shows the strong association between "ant" complexity and the coordination effort, measured by the number of e-mails per developer exchanged prior to each release. Hence, it seems product changes are strongly associated with cross-module interactions of the sub-system "ant," which in turn are strongly associated with the coordination effort needed to implement such changes. The causal loop diagram shown in Figure 18 summarizes the links we observed between the workload, product complexity, and coordination effort during the development of Ant. It remains to be investigated if there is a feedback link from the coordination effort to the workload.

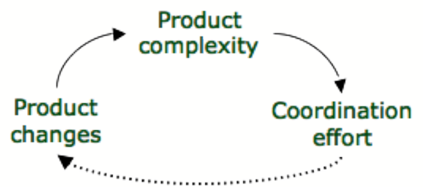


Figure 18: Linking workload, product complexity, and coordination effort

DISCUSSION

Implications for Software Developers

It is important to monitor the architecture of the product and identify the key sub-system modules that drive most of the product changes. Interactions across modules require particular managerial attention.

It is also important to define the most appropriate level of granularity at which to analyze the architecture. Based on our case study, it seems like one or two sub-system modules are the ones that concentrate most of the development effort. Hence, particular effort should be put into understanding the architecture of these salient modules. In the case of Ant, the most informative module is the sub-system module "ant" in layer 2, which contained over 90% of the product components. Understanding the drivers of architectural complexity in "ant" by focusing on the interactions across the modules that form it would be a major predictor of the coordination effort associated with Ant development.

As for the architectural evolution of Ant, it is important to emphasize that the architecture of a new product does not magically emerge in the first version of the product. Establishing the architecture of the product is a dynamic process that goes through distinct phases which require different managerial competences. The initial phase is a phase characterized by experimentation in which a search for the optimal architecture takes place. After a few versions, a dominant configuration emerges and the growth phase takes off. This phase focuses on taking advantage of the established architecture to improve and increase the functionality of the product. Eventually, limits to growth start appearing and the complexity of the architecture saturates, which might call for a refactoring of parts of (or the entire) product architecture.

What does the dynamic evolution of the product data tell us? It confirms that massive reorganizations of code are part of software development. This is somewhat in contrast with large-scale development efforts in other mature industries. In large assembly projects such as cars and airplanes, major reorganizations are avoided once the project has been started. In that sense we can see the prototyping approach of software development at work. But while reorganizations happen, they nonetheless tend to subside towards the later stages of development. As a result, the Ant project followed a prototyping approach during the beginning and became a more linear project towards the end of building the functionality.

Theoretical Implications

This paper presents a structured research approach to investigate the dynamics of software architectures. We have developed architectural metrics of product complexity that capture not only intrinsic complexity but also the mechanisms by which developers manage complexity (i.e. modules and layers). We propose a three-step approach to study the dynamics of software architectures over a large number of products so that theoretical propositions about architectural dynamics can be tested statistically: (1) capture the architectural dynamics, (2) capture the organizational dynamics, and (3) compare product and organizational dynamics. Our current efforts are focused on applying this approach across several software products.

One of our basic findings is that the dynamics of software architectures follow an S-shaped path. That is, the complexity of the architecture slowly increases in the first few versions of the product as the effort is focused on organizing the architecture (mapping functional and "physical" elements of the product). Once the dominant architecture emerges, then significant growth in complexity will be visible, until limits to growth saturate the current architecture, leading to some additional reconfiguration of the dominant architecture. This is consistent with the model of technology evolution based on industry studies that have examined the emergence and adoption of new technologies [37, 38].

Another interesting outcome from this study pertains to the ability of our approach to identify the layer(s) and module(s) wherein most of the complexity and architectural dynamism reside. This is extremely important for modelers to know, since it has been noted that product architecture models can differ greatly depending on the chosen level of analysis [39].

Studying the product and organizational dynamics of Ant has allowed us to address some methodological challenges associated with studying software architectural dynamics. Yet, many open questions remain to be addressed in future research. Does software complexity always increase over time? How do open source and "closed source" software architectures differ? How does the architecture of the product impact the participation of new developers? Our current research efforts aim to find insightful answers to these questions.

ACKNOWLEDGEMENTS

We appreciate the support from Lattix Inc., which provided us with the software application needed to document the architecture of software products. We thank Neeraj Sangal for insightful feedback throughout this research. We appreciate the assistance of Marc Ortiz and Guilin Tang during the data collection of organizational attributes of Ant. We also appreciate

the comments from three anonymous reviewers from the *ASME Design Theory and Methodology* conference.

REFERENCES

[1] C. Y. Baldwin and K. B. Clark, *Design Rules: The Power of Modularity*, vol. 1. Cambridge, MA: MIT Press, 2000.
[2] K. T. Ulrich, "The Role of Product Architecture in the Manufacturing Firm," *Research Policy*, vol. 24, pp. 419-440, 1995.
[3] A. A. Yassine and L. L. Wissmann, "The Implications of Product Architecture on the Firm," *Systems Engineering*, vol. 10, pp. 118-137, 2007.
[4] R. M. Henderson and K. B. Clark, "Architectural Innovation: The Reconfiguration of Existing Product Technologies and the Failure of Established Firms," *Administrative Science Quarterly*, vol. 35, pp. 9-30, 1990.
[5] C. Alexander, *Notes on the Synthesis of Form*. Cambridge, MA: Harvard University Press, 1964.
[6] H. A. Simon, *The Sciences of the Artificial*, 2nd ed. Cambridge, MA: MIT Press, 1981.
[7] IEEE, "IEEE Recommended Practice for Architectural Description of Software-Intensive Systems," Institute of Electrical and Electronics Engineers Standards Association, IEEE Std 1471-2000, Sep. 21 2000.
[8] K. T. Ulrich and S. D. Eppinger, *Product Design and Development*, 3rd ed. New York: McGraw-Hill, Inc., 2004.
[9] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice Hall, 1996.
[10] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, pp. 1053-1058, 1972.
[11] D. L. Parnas, "Designing Software for Ease of Extension and Contraction," *Transactions on Software Engineering*, vol. 5, 1979.
[12] A. D. MacCormack, R. Verganti, and M. Iansiti, "Developing Products on "Internet Time": The Anatomy of a Flexible Development Process," *Management Science*, vol. 47, pp. 133-150, 2001.
[13] S. H. Strogatz, "Exploring Complex Networks," *Nature*, vol. 410, pp. 268-276, 2001.
[14] J. N. Warfield, *A Structure-Based Science of Complexity: Transforming Complexity into Understanding*. Amsterdam: Kluver Publishing, 2000.
[15] S. A. Kauffman, *The Origins of Order: Self-Organization and Selection in Evolution*. New York: Oxford University Press, 1993.
[16] N. P. Suh, *Axiomatic Design: Advances and Applications*. New York: Oxford University Press, 2001.
[17] S. K. Ethiraj and D. Levinthal, "Modularity and Innovation in Complex Systems," *Management Science*, vol. 50, pp. 159-173, 2004.
[18] N. P. Suh, "Theory of Complexity, Periodicity, and the Design Axioms," *Research in Engineering Design*, vol. 11, pp. 116-131, 1999.
[19] T. Jarratt, C. Eckert, and P. J. Clarkson, "Development of a Product Model to Support Engineering Change Management," presented at Proceedings of the TCME 2004, Lausanne, Switzerland, 2004.
[20] P. J. Clarkson, C. Simons, and C. Eckert, "Predicting Change Propagation in Complex Design," *Journal of Mechanical Design*, vol. 126, pp. 788-797, 2004.
[21] D. M. Sharman and A. A. Yassine, "Characterizing Complex Product Architectures," *Systems Engineering*, vol. 7, pp. 35-60, 2004.
[22] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code," *Management Science*, vol. 52, pp. 1015-1030, 2006.
[23] V. Tang and V. Salminen, "Towards a Theory of Complicatedness: Framework for Complex Systems Analysis and Design," presented at 13th International Conference on Engineering Design (ICED), Glasgow, Scotland, 2001.
[24] F. Guo and J. K. Gershenson, "A Comparison of Modular Product Design Methods on Improvement and Iteration," presented at ASME International Design Engineering Technical Conferences (Design Theory & Methodology Conference), Salt Lake City, UT, 2004.
[25] J. K. Gershenson, G. J. Prasad, and Y. Zhang, "Product Modularity: Measures and Design Methods," *Journal of Engineering Design*, vol. 15, pp. 33-51, 2004.
[26] K. Hölttä, E. S. Suh, and O. d. Weck, "Tradeoff Between Modularity and Performance for Engineered Systems and Products," presented at International Conference on Engineering Design (ICED), Melbourne, Australia, 2005.
[27] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using Dependency Models to Manage Complex Software Architecture," presented at 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages And Applications (OOPSLA), San Diego, CA, 2005.
[28] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The Structure and Value of Modularity in Software Design," *ACM SIGSOFT Software Engineering Notes*, vol. 26, pp. 99-108, 2001.
[29] D. V. Steward, "The Design Structure System: A Method for Managing the Design of Complex Systems," *IEEE Transactions on Engineering Management*, vol. 28, pp. 71-74, 1981.
[30] S. D. Eppinger, D. E. Whitney, R. P. Smith, and D. A. Gebala, "A Model-Based Method for Organizing Tasks in Product Development," *Research in Engineering Design*, vol. 6, pp. 1-13, 1994.
[31] T. R. Browning, "Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions," *IEEE Transactions on Engineering Management*, vol. 48, pp. 292-306, 2001.
[32] M. E. Sosa, S. D. Eppinger, and C. M. Rowles, "A Network Approach to Define Modularity of Components in Product Design," *Journal of Mechanical Design*, 2007.
[33] M. E. Sosa, S. D. Eppinger, and C. M. Rowles, "Identifying Modular and Integrative Systems and Their Impact on Design Team Interactions," *Journal of Mechanical Design*, vol. 125, pp. 240-252, 2003.
[34] T. U. Pimmler and S. D. Eppinger, "Integration Analysis of Product Decompositions," presented at ASME International Design Engineering Technical Conferences (Design Theory & Methodology Conference), Minneapolis, 1994.
[35] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley, 1999.
[36] J. A. Roberts, I.-H. Hann, and S. A. Slaughter, "Understanding the Motivations, Participation, and Performance of Open Source Software Developers: A Longitudinal Study of the Apache Projects," *Management Science*, vol. 52, pp. 984-999, 2006.
[37] W. J. Abernathy and J. Utterback, "Patterns of Industrial Innovation," *Technology Review*, vol. 80, pp. 40-47, 1978.

[38] J. M. Utterback, *Mastering the Dynamics of Innovation: How Companies Can Seize Opportunities in the Face of Technological Change*. Boston, MA: Harvard Business School Press, 1996.

[39] S. K. Fixson, "Product Architecture Assessment: A Tool to Link Product, Process, and Supply Chain Design Decisions," *Journal of Operations Management*, vol. 23, pp. 345-369, 2005.

APPENDIX: SUMMARY OF DATA

Table 1: Summary of product and organizational data for Ant's major releases

	Version						
Variable	**1.10**	**1.20**	**1.30**	**1.40**	**1.50***	**1.60***	**1.65***
Release Date	7/19/2000	10/24/2000	3/3/2001	9/3/2001	7/10/2002	12/18/2003	6/2/2005
Days Since Last Release	N/A	97	130	184	310	526	532
Product Data							
# Components (*n*)	70	122	126	178	293	352	380
# Component Modules (*m*)	4	13	8	13	21	24	25
# Layers	3	4	3	3	3	4	4
# Dependencies (*k*)	206	465	476	706	1137	1434	1389
Dependencies below diagonal (feed forward)	204	446	464	687	1086	1355	1307
Dependencies above diagonal (feedback)	2	19	12	19	51	79	82
Product complexity, $C_1=n*k$	14420	56730	59976	125668	333141	504768	527820
Product complexity, C_2	2124	9974	8183	15617	32958	46719	51735
Propagation cost (% reachable dyads)	11.5%	15.5%	14.6%	12.4%	13.0%	15.5%	16.7%
Organizational data							
# Developers & Committers	88	78	91	66	56	155	149
# New Developers	N/A	48	40	0	9	130	117
# "Changes"	N/A	42	26	83	147	131	94
# Bugs Fixed (documented)	N/A	5	17	36	62	122	116
# Tasks Added	N/A	19	13	18	23	24	2
# E-mails (developer list)	2494	2728	4581	6404	16912	18438	14256

*The measures for these versions include interim minor versions; e.g., 1.5 was released 310 days after 1.4, but during that time there was a 1.4.1. The bugs fixed as part of 1.4.1 are counted as part of those fixed for 1.5 (i.e., everything since 1.4), etc.