Software Architecture: A Travelogue

David Garlan

Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213 USA
garlan@cs.cmu.edu

ABSTRACT

Over the past two and a half decades software architecture has emerged as an important subfield of software engineering. During that time there has been considerable progress in developing the technological and methodological base for treating architectural design as an engineering discipline. However, much still remains to be done to achieve that. Moreover, the changing face of technology raises a number of challenges for software architecture. This travelogue recounts the history of the field, its current state of practice and research, and speculates on some of the important emerging trends, challenges, and aspirations.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: domain-specific architectures, languages, patterns.

General Terms

Documentation, Design, Standardization, Reliability

Keywords

Software architecture, software product lines, software frameworks, architecture description languages, architecture styles, architecture trends, architecture and agility.

1. INTRODUCTION

A critical issue in the design and construction of any complex software system is its architecture: that is, its organization as a collection of interacting elements – modules, components, services, etc. A good architecture can help ensure that a system will satisfy its key functional and quality requirements, including performance, reliability, portability, scalability, and interoperability. A bad architecture can be disastrous.

Over the past two and a half decades software architecture has received increasing attention as an important subfield of software engineering. Practitioners have come to realize that getting an architecture right is a critical success factor for system design and development. They have begun to recognize the value of making explicit architectural choices, and leveraging past architectural designs in the development of new products. Today there are numerous books on architectural design, regular conferences and workshops devoted specifically to software architecture, a growing number of commercial tools to aid in aspects of architectural design, courses in software architecture, major government and industrial research projects centered on software architecture, and an increasing number of formal architectural standards. Codification of architectural principles, vocabulary, methods, and practices has begun to lead to repeatable processes of architectural design, criteria for making principled tradeoffs among architectural decisions, and standards for documenting, reviewing, and implementing architectures.

However, despite this progress, as engineering disciplines go, the field of software architecture remains relatively immature. While the foundations of an engineering basis for software architecture are now clear, there remain numerous challenges and unknowns. We can therefore expect to see major new developments in the field over the next decade – both in research and practice. Some of these developments will be natural extensions of the current trajectory. But there are also a number of important new opportunities, brought about by the changing face of technology and its roles in society.

In 2000 I was invited to write an article, "Software architecture: a roadmap" [28] in which I assessed the current state and future prospects for architecture. This paper revises that article, now with the hindsight of almost a decade and a half, and attempts to provide a travelogue describing the conceptual terrain and its key features: its history, its current state, and how it may evolve in the future to address emerging challenges and opportunities. As we will see, many of the challenges are similar to those described in the 2000 paper, but with a somewhat different flavor today.

To provide a broader perspective than my own, and to hear from some of the prominent people in the field, I asked a few colleagues to contribute ideas and prose. I am grateful to Grady Booch, Paul Clements, Philippe Kruchten, and Mary Shaw, who agreed to help out, in part acting as "guides" for parts of landscape that they know well.

2. WHAT IS SOFTWARE ARCHITECTURE?

While there are numerous definitions of software architecture, at the core of most of them is the notion that the architecture of a system describes its gross structure. This structure illuminates the top-level design decisions, including things such as how the system is composed of interacting parts, what are the principal pathways of interaction, and what are the key properties of the parts and the system as a whole. Additionally, an architectural description includes sufficient information to allow high-level analysis and critical appraisal.

Software architecture typically plays a key role as a bridge between requirements and implementation (see Figure 1). By providing an abstract description of a system, the architecture exposes certain properties, while hiding others. Ideally this representation provides an intellectually tractable guide to the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

FOSE'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2865-4/14/05...$15.00
http://dx.doi.org/10.1145/2593882.2593886

overall system, permits designers to reason about the ability of a system to satisfy certain key requirements, explicitly captures the intent and principles that govern its design, and prescribes a blueprint for system construction and composition.

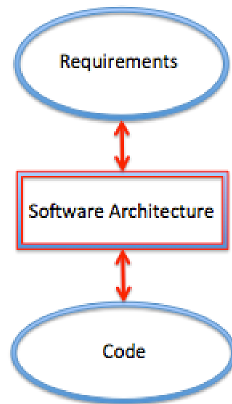

Figure 1: Software Architecture as a Bridge

For example, an architecture for a signal processing application might be constructed as a dataflow network in which the nodes read input streams of data, transform that data, and write to output streams. Designers might use this decomposition, together with estimated values for input data flows, computation costs, and buffering capacities, to reason about possible bottlenecks, resource requirements, and schedulability of the computations.

As another example, consider an application that provides application services over the Internet. A typical architecture for such a system will adopt an N-tiered organization, using a data tier to store persistent information, one or more tiers to provide application functionality, and a user interface tier. Given this overall structure, an architect will need to decide which capabilities to assign to each tier, how to provide adequate privacy and security for communicated and stored data, how to guarantee reasonable response times, how to ensure that the system will scale gracefully as the number of clients increases over time, and what technologies will be used to realize the design.

To elaborate, software architecture can play an important role in at least six aspects of software development:

1. *Understanding:* Software architecture simplifies our ability to comprehend large systems by presenting them at a level of abstraction at which a system's high-level design can be easily understood [57]. Moreover, at its best, architectural description exposes the high-level constraints on system design, as well as the rationale for making specific architectural choices [37][15].
2. *Reuse:* Architectural design supports reuse of both components and also frameworks into which components can be integrated. Domain-specific software architectures, frameworks, platforms and architectural patterns are various enablers for reuse, together with libraries of plug-ins, add-ins and apps [12][14].
3. *Construction:* An architectural description provides a partial blueprint for development by indicating the major components and dependencies between them. For example, a layered view of an architecture typically documents abstraction boundaries between parts of a system's implementation, identifies the internal system interfaces, and constrains what parts of a system may rely on services provided by other parts [15].
4. *Evolution:* Architectural design can expose the dimensions along which a system is expected to evolve. By making explicit a system's "load-bearing walls," maintainers can better understand the ramifications of changes, and thereby more accurately estimate costs of modifications [32]. In many cases such evolution and variability constraints are manifested in product lines, frameworks and platforms, which dictate how the system can be instantiated or adapted through the addition of application-specific features and components [10][14].
5. *Analysis:* Architectural descriptions provide opportunities for analysis, including system consistency checking [3][7], conformance to constraints imposed by an architectural style [1], satisfaction of quality attributes [16], and domain-specific analyses for architectures built in specific styles [23][31][44][47].
6. *Management:* For many companies the design of a viable software architecture is a key milestone in an industrial software development process. Critical evaluation of an architecture typically leads to a much clearer understanding of requirements, implementation strategies, and potential risks, reducing the amount of rework required to address problems later in a system's lifecycle [8][16].

3. THE PAST

In the early decades of software engineering, architecture was largely an ad hoc affair.[1] Descriptions typically relied on informal box-and-line diagrams, which were rarely maintained once a system was constructed. Architectural choices were made in an idiosyncratic fashion – often by adapting some previous design, whether or not it was appropriate. Good architects – even if they were classified as such within their organizations – learned their craft by hard experience in particular domains, and were unable to teach others what they knew. It was usually impossible to analyze an architectural description for consistency or to infer non-trivial properties about it. Nor was there any way to check that a system's implementation faithfully represented its architectural design.

However, despite their informality, even from the earliest days of software development, architectural descriptions have been central to system design. As people began to understand the critical role that architectural design plays in determining system success, they also began to recognize the need for a more disciplined approach. Early authors began to observe certain unifying principles in architectural design [52], to call out architecture as a field in need of attention [51][58], and to establish a more-formal working vocabulary for software architects [57]. Tool vendors began thinking about explicit support for architectural design. Language designers began to consider notations for architectural representation [46]. Standards organizations began to promote standardized languages and tools.

[1] To be sure, there were some notable exceptions. Parnas recognized the importance of system families [49], and architectural decomposition principles based on information hiding [50]. Others, such as Dijkstra, exposed certain system structuring principles [22].

Within industry, two trends highlighted the importance of architecture. The first was the recognition of a shared repertoire of methods, techniques, patterns and idioms for structuring complex software systems. For example, the box-and-line-diagrams and explanatory prose that typically accompany a high-level system description often refer to such organizations as a "pipeline," a "blackboard-oriented design," or a "client-server system." Although these terms were rarely assigned precise definitions, they permitted designers to describe complex systems using abstractions that made the overall system intelligible. Moreover, they provided significant semantic content about the kinds of properties of concern, the expected paths of evolution, the overall computational paradigm, and the relationship between this system and other similar systems.

The second trend was the concern with exploiting commonalities in specific domains to provide reusable frameworks for product families. Such exploitation was based on the idea that common aspects of a collection of related systems can be extracted so that each new system can be built at relatively low cost by "instantiating" the shared design and reusing shared artifacts. Familiar examples include the standard decomposition of a compiler (which permits undergraduates to construct a new compiler in a semester), standardized communication protocols (which allow vendors to interoperate by providing interoperable services), and customizable frameworks, platforms and product lines.

During the 1990s and 2000s these trends, and others, led to an explosion of interest in software architecture. Sometimes referred to as the "Golden Age of Software Architecture" [56], during this period the field matured rapidly, producing many books on software architecture, improved theories and formalisms for reasoning about architecture, tools to automate their construction and implementation, and methods for integrating architecture into mainstream software development. The next section surveys some of the more important of these.

4. SOFTWARE ARCHITECTURE TODAY

Although there is considerable variation in the state of the practice, today software architecture is widely visible as an important and explicit design activity in software development. Job titles now routinely reflect the role of software architect; companies rely on architectural design reviews as critical staging points; and architects recognize the importance of making explicit tradeoffs within the architectural design space [42].

In addition, the technological and methodological basis for architectural design has improved dramatically. Four important advances have been (1) the codification and dissemination of architectural design expertise; (2) the emergence of platforms and product lines, and their associated ecosystems; (3) the development of principles, languages and tools for architecture description; and (4) the integration of architectural design into the broader processes of software development, and, in particular the relationship between architecture and agility.

4.1 Codification and Dissemination

One early impediment to the emergence of architectural design as an engineering discipline was the lack of a shared body of knowledge about architectures and techniques for developing good ones. Today the situation has improved dramatically, due in part to the publication of books on architectural design [7][12][23][43][57][60] and courses [37].

An important common theme in these is the use of standard architectural *styles.*[2] An architectural style typically specifies a design vocabulary, constraints on how that vocabulary is used, and semantic assumptions about that vocabulary. For example, a pipe-and-filter style might specify vocabulary in which the processing components are data transformers (filters), and the interactions are via order-preserving streams (pipes). Constraints might include the prohibition of cycles. Semantic assumptions might include the fact that pipes are lossless and preserve the order of data written to them.

Other common styles include blackboard architectures, client-server architectures, repository-centered architectures, event-based architectures, N-tiered architectures, and service-oriented architectures. Each style is appropriate for certain purposes, but not for others [55]. For example, a pipe-and-filter style would likely be appropriate for a signal processing application, but not for an application in which there is a significant requirement for concurrent access to shared data. Moreover, each style is typically comes with a set of analyses that can be performed on systems in that style. For example, it makes sense to analyze a pipe-and-filter system for end-to-end latency, whereas transaction rates would be a more appropriate analysis for a repository-oriented style.

The identification and documentation of such styles (as well as their more domain-specific variants) enables others to adopt previous architectural structures as a starting point. In that respect, the architectural community has paralleled other communities in recognizing the value of established, well-documented design patterns, such as those found in [25].

Although styles are often a good starting point for architectural design, in practice they need to be complemented by techniques for improving specific quality attributes of a system. Examples include the use of redundancy to improve availability, caching to improve performance, and authentication to improve security. Such techniques are sometimes referred to as *architectural tactics*. Books on software architecture now survey many of these [7], and entire books have been written on tactics for specific quality attributes such as performance and security [53].

Additionally, the realities of software construction often force one to compose systems from components and frameworks that were not architected in a uniform fashion. For example, one might combine a database from one vendor, with middleware from another, and a user interface framework from a third. In such cases the parts do not always work well together – in large measure because they make conflicting assumptions about the environments in which they were designed to work [29][30]. This has led to the need to identify architectural strategies for bridging mismatches. Although, we are far from having well understood ways of systematically detecting and repairing such mismatch, a number of tactics have been developed to deal with this problem, and we are starting to see the introduction of automated mismatch repair tools.

[2] Some treatments of software architecture use the term "pattern" in place of "style". For a discussion of this terminology, see [15] pages 32-36.

4.2 Platforms and Product Lines

As noted earlier, one of the important trends in software engineering has been exploitation of commonality across multiple products in order to reduce development costs for new systems through customization of a shared asset base [10][14]. Two specific manifestations of that trend are improvements in our ability to create product lines within an organization and the emergence of cross-vendor platforms. At the heart of such efforts is an architectural design that determines what parts of a system are shared, and how those parts can be extended to provide the capabilities needed for a specific system.

Like architectural styles, the architectures underlying platforms and product lines take advantage of common architectural structures, but do so in a domain-specific way. By narrowing the domain over which they apply, they trade off generality for power – for example, in the form of opportunities for code reuse and the ability to perform specialized analyses. Figure 2 illustrates this point.

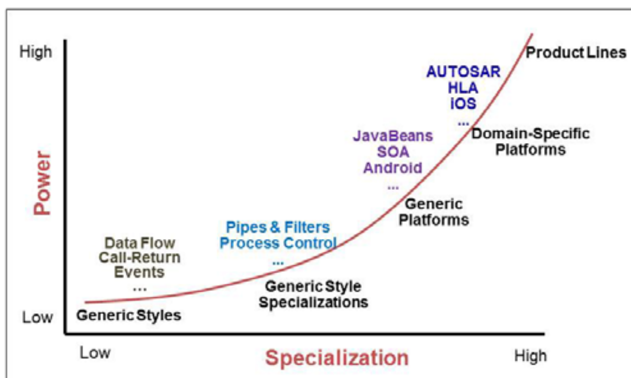

Figure 2: Power vs. Specialization in Architecture Reuse

At the far left are generic architectural styles, which are largely domain-independent. To the right are more-specific versions of these styles. They narrow the class of system to which they apply, but remain largely domain independent. For example, a pipe-filter style will likely apply to any system that is primarily transformational, but can be equally useful in domains such as signal processing, music synthesis, scientific workflow computation, or analysis of large volumes of data (using, map-reduce, for example).

Further to the right are generic component integration platforms, which vary in their domain-specificity, but typically provide a way for multiple vendors to extend a base system with new components. Platforms usually provide a set of common services and features, and prescribe requirements for application-specific components (e.g., functions, services, and applications) that are hosted on the platform. Requirements specify the services they must provide to the platform, as well as constraints on how they can access the shared services of the platform. Platforms are typically manifested as one or more reusable code libraries that must be incorporated into a system based on the platform.

Example generic platforms include those that support Internet-based services (such as any number of service-oriented architecture platforms), platforms for mobile devices (such as Android), and domain-specific platforms (such as the High Level Architecture (HLA) for Distributed Simulation [5] or the AUTOSAR architecture for automotive systems[3]).

At the far right we find product lines, which are highly domain-specific, applying specifically to a set of products within a particular company. Product lines provide huge economies of scale when used appropriately, and there are numerous success stories of their use.[4] However, they also require up-front investment in creating a reusable asset base, as well as on-going organizational commitment to maintain, promote, and apply them to new product development [10][14][48].

Although good architectural design is at the heart of any successful platform or product line, perhaps ironically, their use can substantially reduce the architectural responsibilities of software developers, since many, if not most, of the architectural decisions have already been made by the platform or product line designers. For instance, many platforms provide a security model, that when used correctly obviates the need for additional security tactics.

While the use of platforms and product lines reduces the architectural burden on developers, it may not entirely eliminate it. Indeed, critical quality attributes such as performance and reliability may depend heavily on *how* the platform services are used or the product line is instantiated.

One open issue in product lines is whether or not it's necessary to create a design for a product line architecture at all. Paul Clements has argued that his experience at BigLever Software, Inc. (a company that helps organizations adopt automation-based product line engineering) has taught him that it is best to concentrate on the architectures for the individual products.[5] If an overarching product line architecture is deemed useful, it can be seen as an emergent conceptual design that is the aggregation of the individual products' (possibly quite different) architectures. That is, rather than investing in a large up-front effort to specify the common and variable parts of a single instantiable design, a company should use automation to turn out product instances by creating (and then exercising) variation points in the shared assets -- including those shared assets that represent the architecture. In this way, architecture can be treated consistently with the range of other software development artifacts (requirements, design specifications, code, tests, user manuals, etc.). In all cases, commonality and variability will then emerge (and be captured) over time as needed by the products. (Cf., the discussion about agile development in Section 4.4.).

On the other hand, other experience suggests that there is considerable value in formalizing the product line or platform itself – or at least the parts of it that describe how it can be extended or specialized. For instance, consider Microsoft's experience with tools that check device driver conformance to the API protocols required by their operating system [6]. Such tools have greatly reduced the errors that device driver providers made in providing extensions to the Windows platform.

---

[3] http://www.autosar.org/

[4] For example, see the Software Product Line Hall of Fame, http://splc.net/fame.html.

[5] Paul's blog can be found at the Big Lever Newsletters website: www.biglever.com/newsletters/Pauls_three_surprises_3.html

4.3 Architecture Description

A critical question for software architects is how to describe their architectural designs. Ideally those descriptions should convey their design intent clearly to others, allow critical evaluation, and require low overhead to create and maintain.

Broadly speaking, today there are three general approaches. The first is *informal description.* Such descriptions typically use general-purpose graphical editing tools (PowerPoint, Visio, etc.) coupled with prose to explain the meaning of the drawings.

Informal descriptions have the advantage of being easy to produce, and not requiring special expertise. But they have a host of disadvantages. The meaning of the design may not be clear since the graphical conventions will likely not have a well-defined semantics. Informal descriptions cannot be formally analyzed for consistency, completeness, or correctness. Architectural constraints assumed in the initial design are not enforced as a system evolves. There are few tools to help architectural designers with their tasks.

The second approach is *semi-formal description.* This approach uses generic modeling notations that may lack detailed semantics, but provide a standardized graphical vocabulary supported by commercial tools. The primary example of a semi-formal description language is UML.[6] In 2005 the Object Management Group (which manages the UML standard) adopted UML 2.0, incorporating explicit constructs for architectural modeling, such as components, connectors, ports and hierarchical descriptions [9].

A general-purpose modeling language, such as UML, has the advantages of using notations that practitioners are likely to be familiar with, that are supported by commercial tools, and that provide a link to object-oriented modeling and development. But such languages are limited by their lack of support for formal analysis and their lack of expressiveness for some architecturally relevant concepts. (For example, connectors are not first class entities in UML in the same way as components are [15].)

The third approach is *fully-formal description*. In response to the problems with the other two approaches a number of researchers have proposed formal notations for representing and analyzing architectural designs. Generically referred to as "Architecture Description Languages" (ADLs), these notations provide both a conceptual framework and a concrete syntax for formal modeling of software architectures [46]. They also typically provide tools for parsing, displaying, compiling, analyzing, or simulating architectural descriptions.

In the 1990's many ADLs were proposed, each with certain distinctive capabilities [46]. For instance Wright supports the formal specification and analysis of interactions between architectural components [3], xADL provides an extensible specification language for architectures based on XML [19], and Acme supports the formal definition of architectural styles [34]. To date those languages have largely failed to become established in industrial practice. However, they *have* had an impact the revision to UML in 2005, and in several cases have formed the basis for standardized, formal ADLs used in specific domains. For instance, both AADL [24] and SysML[7] are standardized languages for modeling and analyzing real-time, safety critical systems, such as avionics and automotive systems.

It remains to be seen whether more-formal approaches to architectural description will gain in popularity over time. Today there is a strong interest across the field of software engineering in *model-based engineering*, reflecting the maturation of the field from ad hoc practices to formal engineering practices. Architectural models are an important class of engineering model that can potentially play a prominent role in that evolution [27].

Independent of the level of formality associated with architectural representations, there are a number of important cross-cutting principles that are now well understood. The most important of these is the idea that in order to describe system's architecture, one must use multiple views or viewpoints, reflecting different perspectives on the architectural design problem. For example, it is helpful to distinguish between coding structures and run-time structures, since they represent different kinds of design commitments, and impact the emergent properties of a system in very different ways. The former is particularly useful for reasoning about maintainability and the cost of change, while the latter is useful for reasoning about properties such as performance, scalability, and reliability. Several systems of views have been proposed, including Kruchten's 4+1 approach [40], the IEEE 1471-2000 Standard [38], and the "Views and Beyond" approach [15].

However, while the utility of multiple views is well understood, the consequent problem of reconciling multiple views is not. Research is needed to provide ways to make sure that views are consistent in areas where their concerns overlap, and that changes to one view are appropriately propagated to other views. Some important challenges in this area include (a) understanding when a system of partial views provides a complete representation of all aspects relevant to the design; (b) providing formal criteria for view consistency, including both structure and semantics; and (c) using views to capture refinement relationships between abstract and concrete architectural designs.

4.4 Agility and Architecture

One important issue for software architecture is how best to integrate architectural practices into the broader software development processes. At the core of this issue are questions such as "how much architecture is enough?" and "when should architectural design be performed?"

Considerable progress has been made in terms of establishing processes for architectural evaluation and documentation [15][16]. In particular, architectural reviews are now commonplace in many organizations, where the benefits have been well documented [45]. Additionally, studies have been done to quantify the amount of architectural design that is appropriate for various sizes and classes of systems [8].

One of the interesting debates over the past few years has been the role of architecture in agile processes. It may seem at first glance that architecture would have no place in a rapidly evolving system, where an organization's development practices deemphasize the creation of artifacts that do not manifest

[6] Some of the diagram types in the UML family of modeling notations *do* have formal semantics, but the principle ones used for modeling architectural structure do not.

[7] http://www.sysml.org/

themselves directly as user-visible functionality.[8] But further reflection suggests that architecture is, in fact, an *enabler* of agility, not an impediment to it. We can see this in the emergence of extensible platforms (mentioned earlier), which can be rapidly customized with new plug-ins and applications.

Many in the agile community would argue that architecture is important, but that it will emerge naturally as a system evolves through extension and refactoring.[9] This idea that architecture will emerge spontaneously is reinforced by the fact that most software development efforts today do not require a significant amount of bold new architectural design: the most important design decisions have been made earlier, are fixed by pre-existing conditions, or are a *de facto* architectural standard in the respective industry (cf., Section 4.2). Choices of operating system, servers, programming language, database, middleware, and so on, are predetermined in the vast majority of software development projects, or have a very narrow range of possible variations.

Architectural design, when it is really needed because of project novelty, has an uneasy relationship with traditional agile practices. Unlike the functionality of the system, it cannot easily be decomposed into small, incremental chunks of work, user stories or "technical stories". The difficult aspects of architectural design are driven by systemic quality attributes (security, high availability, fault tolerance, interoperability, scalability, etc.), or are development-related (testability, certification, and maintainability). In general, these cannot be easily decomposed, and tests to determine their satisfaction are difficult to produce up-front. Moreover, key architectural choices usually cannot be easily retrofitted on an existing system by means of simple refactorings. Late decisions may require the replacement of large bodies of the code, and therefore many of the important architectural decisions have to be made early (although not necessarily all at once, up front).

A number of people have grappled with this tension, including Cockburn's "walking skeleton",[10] the Scaled Agile Framework of Leffingwell et al.,[11] and Fairbank's "risk-driven" approach [23]. A common theme is that architectural design and the incremental building of a system (i.e., its user-visible functionality) must go hand-in-hand. The important questions to answer are: How do we pace ourselves? How do we address architectural issues and make decisions, over time, in a way that will lead to a flexible architecture, and enable developers to proceed? In which order do we pick the quality attribute aspects and address them?

One of the ways to understand these questions is in terms of technical debt [18]. At any time, a software development team is faced with a choice of what to focus on in the next release cycle, iteration, or sprint. In the "backlog" of things not yet done, there are four kinds of elements, illustrated in Figure 3 (adapted from [41]).

Items that have *visible value*:

- *Category I* (the **green stuff)**: new features (services, functionalities, capabilities) to be added to the system, as well as visible improvements in quality attributes (capacity, response time, interoperability).
- *Category II* (the **red stuff)**: customer-visible defects, limiting usefulness or negatively impacting perceptions of the product.

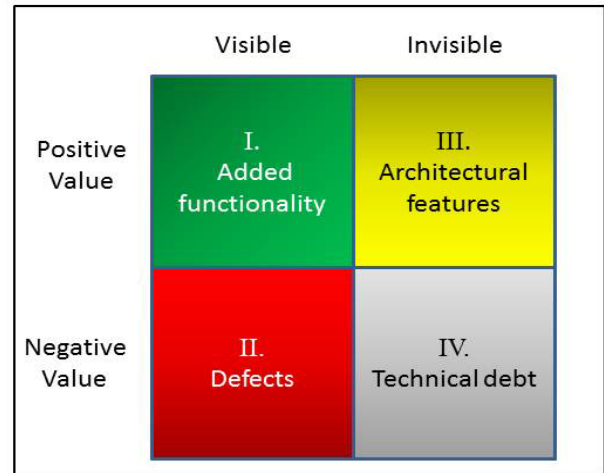

Figure 3: Four Types of Backlog Elements

Both categories impose a development cost to address and tradeoffs are involved in determining how to prioritize them: how much defect fixing relative to new features can we afford to do?

But there are also items that are *invisible* to the outside world:

- *Category III* (the **yellow stuff)**: architectural elements, infrastructure, frameworks, deployment tools, etc. Known to the internal development team, and architects, they are often deferred in favor of more addressing the visible elements. Their cost is often "lumpy": they are hard to break down into small increments. We know that they add value, in the long term, by increasing future productivity, and often improving key quality attributes. But this value is hard to quantify.
- *Category IV* (the **grey stuff)**: elements that have both a negative value, and are invisible – technical debt. These are the result of earlier architectural and implementation decisions that may have seemed wise at the time, but which in the current context are suboptimal and hurt the project – usually through reduced productivity or impact on the evolution of the system. Category IV elements are known by the development team, but rarely expressed at the level of key decision makers, who determine the future release roadmap. Shortcuts, or failure to develop the yellow stuff of Category III, increases the amount of grey stuff in Category IV, further inhibiting progress.

A compounding factor is that the categories have *dependencies* between them – especially, dependencies of the visible (I and II) and the invisible (III and IV) categories. Making tradeoffs between the various categories now becomes more complicated, and requires diverse expertise, not just consideration of market value. Time plays a crucial role, too: the value of delivering a new feature is immediate, while the value of developing a good architecture may be reaped only over a long period of time.

---

[8] Literature in the agile community is full of mantras such as, YAGNI (You Ain't Gonna Need It), No BUFD (No Big Up-Front Design), and "Defer decision to the last responsible moment"

[9] For example, Principle #11 in the "agile manifesto" says "The best architectures, requirements, and designs emerge from self-organizing teams" [2].

[10] http://alistair.cockburn.us/Walking+skeleton

[11] http://scaledagileframework.com/

While the challenge of determining the value and timing of architectural investments remains an open one, economic concepts such as Net Present Value, the Incremental Funding Method, and Real Options, combined with dependency analysis, may help decision making choices for short- or long-term development planning [21][59].

5. CHALLENGES AHEAD

What about the future? Although software architecture is on a much more solid footing than two decades ago, it is not yet fully established as a discipline that is taught and practiced across the software industry. One reason for this is simply that it takes time for new approaches and perceptions to propagate. Another reason is that the technological basis for architecture design (as outlined earlier) is still maturing. In both of these areas we can expect that a natural evolution of the field will lead to steady advances.

However, the world of software development, and the context in which software is being used, is changing in significant ways. These changes promise to have a major impact on how architecture is practiced. In this section I consider some the important trends and their implications for the field of software architecture.

5.1 Network-Centric Computing

Over the past decade the primary computational model for systems has shifted from localized computation to a network-centric model. Increasingly personal computers and a wide variety of other devices (handheld devices, laptops, phones) are used primarily as a user interface that provides access to remote data and computation, rather than as the primary locus of computation. This trend is not surprising since a network-centric model offers the potential for significant advantages. It provides a much broader base of services than is available on local devices. It permits access to a rich set of computing and information retrieval services that can be used almost anywhere (in the office, home, car, and factory), providing ubiquitous access to information and services.

This trend has a number of consequences for software engineering, in general, and software architecture, in particular. Historically, software systems have been created as closed systems, developed and operated under control of individual institutions. The constituent components may have been acquired from external sources, but when incorporated in a system they came under control of the system designer. Architectures for such systems are largely static – allowing minimal run-time restructuring and variability.

However, within the world of pervasive services and applications available over networks, systems may not have such centralized control. The Internet is an example of such an open system: it is minimally standardized, chiefly at the level of the protocols, addresses, and representations that allow individual sites to interact. It admits of considerable variation both in the hardware that lies below these standards and the applications that lie above. There is no central authority for control or validation. Individual sites are independently administered. Individual developers can provide, modify, and remove resources at will.

For such systems a new set of software architecture challenges emerges [54]. First, is the need for architectures that scale up to the size and variability of the Internet. While many of the traditional architectural paradigms will likely apply, the details of their implementation and specification will need to change.

For example, one attractive form of composition is implicit invocation – sometimes termed "publish-subscribe." Within this architectural style components are largely autonomous, interacting with other components by broadcasting messages that may be "listened to" by other components. Most systems that use this style, however, make many assumptions about properties of its use. For example, one typically assumes that event delivery is reliable, that centralized routing of messages will be sufficient, and that it makes sense to define a common vocabulary of events that are understood by all of the components. In an Internet-based setting all of these assumptions are questionable.

Second, is the need to support computing with dynamically-formed, task-specific, coalitions of distributed autonomous resources. The Internet hosts a wide variety of resources: primary information, communication mechanisms, applications and services, control that coordinates the use of resources, and services such as secondary (processed) information, simulation, editorial selection, or evaluation. These resources are independently developed and independently supported; they may even be transient. They can be composed to carry out specific tasks set by a user; in many cases the resources need not be specifically aware of the way they are being used, or even whether they are being used. Such coalitions lack direct control over the incorporated resources. Selection and composition of resources is likely to be done afresh for each task, as resources appear, change, and disappear. Unfortunately, it is hard to automate the selection and composition activity because of poor information about the character of services and hence with establishing correctness.

Composition of components in this setting is difficult because it is hard to determine what assumptions each component makes about its operating context, let alone whether a set of components will interoperate well (or at all) and whether their combined functionality is what you need. Moreover, many useful resources exist but cannot be smoothly integrated because they make incompatible assumptions about component interaction. For example, it is hard to integrate a component packaged to interact via remote procedure calls with a component packaged to interact via shared data in a proprietary representation.

One emerging solution to this problem is the creation of ecosystems based on a common architectural style or platform, augmented with a rich library of components that can be composed in well-understood ways within a given domain of computation. (Cf. also Section 4.2.) As an example, consider Yahoo! Pipes, a component composition platform that allows one to compose and execute data steam processing applications through a web browser using a large number of transformations available from an on-line library.[12] (More on this topic in Section 5.4.)

Third, there is a need to architect systems that can take advantage of the rich computing base enabled by network-centric computing. Today cloud computing platforms provide almost unlimited access to storage and computation, but exploiting these requires architectures that can scale to large volumes of data and a huge array of available services.

[12] http://pipes.yahoo.com/pipes/

Fourth, there is a need to ensure adequate security and privacy. As systems migrate away from the direct control of individual organizations, architects must find ways to ensure confidentiality, availability, and integrity of their data. In many cases, this will involve a judicious combination of both private and public infrastructure, together with architectures that can bridge that divide.

5.2 Pervasive Computing and Cyber-physical Systems

A second related trend is toward pervasive computing, in which the computing universe is populated by a rich variety of software-enhanced devices: toasters, home heating systems, entertainment systems, smart cars, etc. This trend is leading to an explosion in the number of devices in our local environments – from dozens of devices to hundreds or thousands of devices. In some circles this is referred to as the "Internet of Things".

There are a number of consequent challenges for software architecture. First, we will need architecture design tools that are suited to systems that combine both physical and software elements – cyber-physical systems. Such systems cannot be understood without considering both the properties of their physical elements (power requirements, mechanics, control interfaces, etc.) and their software elements (data stores, communication protocols, security features, etc.). Today design notations and tools tend to favor one or the other; what is needed are unifying approaches that permit designers to focus on different facets of a system, but provide guarantees of consistency and completeness for the overall system.

Second, architectures for these systems will have to be more flexible than they are today. In particular, devices, components, services and other computational elements are likely to come and go in an unpredictable fashion. Handling reconfiguration dynamically, while guaranteeing uninterrupted service, is a hard problem. (See also Section 5.3.)

Third, there is a need for architectures that effectively bridge the gap between technology and technologically-naïve users. Currently, our computing environments are configured manually and managed explicitly by the user. While this may be appropriate for environments in which we have only a few, relatively static, computing devices (such as a couple of PCs and laptops), it does not scale to environments with hundreds of devices. We must therefore find architectures that provide much more automated control over the management of computational services.

Such architectures will need to raise the level of abstraction for configuring pervasive systems, allowing users to focus on their high-level tasks (e.g., being entertained, ensuring the security of their home, managing the energy usage of their appliances) [36]. For example, how many of us have been frustrated trying to figure out how to configure an entertainment system, with multiple remote devices and their myriad buttons? Note that in general it is not merely a matter of making individual devices "smarter". It will require new architectures that span multiple devices, and are able not only to simply their use, but proactively adapt themselves to user's needs.

5.3 Fluid Architectures

In the past, software systems were largely static: changes to system functionality happened rarely, and usually required significant manual oversight and installation. Today systems are much more dynamic: new applications are downloaded onto mobile platforms; new devices are installed in homes; mobile devices enter and leave our computing environment; new web services become available through web applications; system upgrades are required to address discovered security vulnerabilities, etc.

Such fluidity raises a number of architectural challenges. The first is architecting for ease of change and adaptation. For many systems, this requires the ability to dynamically discover and compose available services and resources. For other systems it requires well-defined interfaces that support rapid customization, extension and upgrade. In addition, the architect needs to be sure that such changes do not interfere with on-going services that must continue to function. It would be unfortunate, indeed, if when downloading a new app to your phone, no one could contact you, or if an on-line e-commerce site were to become unavailable to customers when system upgrades were propagated.

Second is the problem of describing and reasoning about the architecture. Since the actual run-time configuration is largely unknown at design time, how can one specify the architecture? Clearly, what is needed is a way to characterize the set of all possible systems. But architectural modeling notations (in any of the categories mentioned in Section 4.3) often have difficulty at characterizing such families. Moreover, the combinatorial explosion of possible future system configurations makes formal analysis difficult. Partial solutions to this problem may emerge from the application of product line variability representation and analysis [48].

Despite these challenges the existence of fluid architectures opens up a set of new opportunities. Chief among these are the ability to create systems that can take a stronger role in ensuring their own health and quality. Such systems are sometimes referred to as self-adaptive systems or autonomic systems [39], and are the subject of considerable research in their own right [13][20].

While there are many approaches to self-adaptation, one prominent technique is to adopt a control systems view of the problem. In this approach each system is coupled with a control layer that is responsible for monitoring the state of that system, detecting opportunities for improvement, and effecting change in as it runs. Central to such an approach (as with any control systems approach) is the use of system models in the control layer. These models reflect the current state of the system, and are used to detect problems and decide on courses of action. It turns out that architectural models are particularly useful in this regard [35]. As such, architectural models now become useful not only as design-time artifacts, but as run-time artifacts as well.

5.4 Socio-technical Ecosystems

The emergence of platforms as a central basis for modern system development has brought with it a new set of challenges for software architecture. Beyond the technical issues of platform design itself (noted earlier), architects must now also consider the socio-technical ecosystems that arise around the platform and are necessary to ensure its sustainability. Such ecosystems include not only the *platform* developers, but also the much larger community of developers who provide platform *extensions* (apps, services, etc.), users who must purchase and install those extensions, governance rules and processes to qualify potential extensions,

incentive systems to motivate developers of extensions, legal and economic systems, and so on [11].

Table 1: Some End-User Architecting Domains

Domain	Type of compositions	Tools
Astronomy	electromagnetic image processing tasks	Astrogrid
Bioinformatics	biological data-analysis services	Taverna
Digital music production	audio sequencing & editing	Steinberg VST
Environmental Science	spatio-temporal experi-ments	Kepler
Film production	scripting animation-effects and components	Maya/Adobe after-effects
Gaming	scriptable and composa-ble 3D engines	Mavenlink
Geospatial Analysis	interactive visualization of geographical data	Ozone widget framework
Home Automa-tion	home devices & services	Control4 Composer
Neuroscience	brain-image processing	Loni Pipeline
Scientific com-puting	transformational work-flows	WINGS
Socio-technical Analysis	network creation, analy-sis & simulation	SORASCS
Virtual Tools	scientific experiment pipelines	Labview

Today we have only a very weak understanding of the interrelationship between all of these moving parts, and the way in which architectural design facilitates a sustainable ecosystem. Consider, for example, the architectural issue of the kinds of interaction the platform will support between components provided by third parties. Limited interaction can provide stronger guarantees, but may reduce flexibility, and will affect the extent to which a component developer can depend on components built by others. Different choices will have a strong impact on the nature of the ecosystem and potentially its long-term viability, as illustrated by the difference between the iOS and Android ecosystems.

One important emerging class of ecosystems allows end users to do their own system composition to create innovative computational systems. Such users typically have minimal technical expertise, and yet still want strong guarantees that the parts will work together in the ways they expect, and to understand the properties of their compositions (How fast will they run? Will they guarantee data privacy?). In essence, this requires support for "end-user architecting" [33]. Some examples of domains and their respective end-user architecting tools are listed in Table 1. Unfortunately, such tools are often difficult to use, and may require considerable low-level knowledge of invocation conventions, parameter settings, and data formats. An open research problem is finding ways to provide improved architecting environments and task-oriented interfaces for such users.

6. CONCLUSION

The field of software architecture is one that has experienced considerable growth over the past two and a half decades, and it promises to continue that growth for the foreseeable future. Although architectural design has matured into an engineering discipline that is broadly recognized and practiced, there are a number of significant challenges that will need to be addressed. Many of the solutions to these challenges are likely to arise as a natural consequence of dissemination and maturation of the architectural practices and technologies that we know about today. Other challenges arise because of the shifting landscape of computing and the roles that software systems play in our world: these will require significant new innovations. In this paper we have attempted to provide a high-level overview of this terrain—illustrating where we have come over the past few years, and speculating about where we need to go to meet the demands of the future.

Reflecting on the differences between now and the state of the world characterized in the 2000 article [28], we can see that many of the challenges noted there are still with us, but with some changes. The challenge of exploiting reuse has matured substantially since 2000 with broad-based use of platforms, frameworks and product lines. The architectural challenges of pervasive computing, noted in 2000, remain, but have broadened to include more generally cyber-physical systems, which affect most of the things we depend on (phones, cars, energy, houses, etc.). Beyond the simple pervasiveness of software, its emerging role as an enhancement to our physical environment suggests the need for a significant broadening of the architectural discipline. The area of network centric computing has blossomed via the Internet, and now requires that we understand how the architectures of the past can be adapted to the connected, distributed, data-rich world of the future. Finally, in this article we noted the importance of ecosystems and their symbiotic relationship with architecture.

7. ACKNOWLEDGEMENTS

This article has benefitted from many long-standing interactions and collaborations with numerous colleagues and students over the past two decades. I would like to thank Grady Booch, Paul Clements, Philippe Kruchten, and Mary Shaw for their contributions to the content and prose of this paper. Paul provided some of the ideas and text for product lines in Section 4.2; Philippe on agile computing and technical debt in Section 4.4; and Mary on dynamic coalitions in Section 5.2. Section 2 describing the roles of software architecture was adapted from an introductory article on software architecture [26]. I would also like to credit past and present members of the ABLE Research Group, and colleagues at the Software Engineering Institute. Finally special thanks to Andreas Metzger and Klaus Pohl who made many helpful suggestions for improvements.

8. REFERENCES

[1] Abowd, G., Allen, R., and Garlan, D. Using style to understand descriptions of software architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering*. ACM Press, December 1993.

[2] Agile Alliance, *Manifesto for Agile Software Development*, June 2001, http://agilemanifesto.org/.

[3] Allen, R. and Garlan, D. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, Vol. 6(3):213-249 July 1997.

[4] Allen, R. and Garlan, D. Formalizing architectural connection. In *Proceeding of the 16th International Conference on Software Engineering*, pps 71-80, May 1994.

[5] Bachinsky, S., Mellon, L., Tarbox, G., and Fujimoto, R. RTI 2.0 architecture. In *Proceedings of the 1998 Spring Simulation Interoperability Workshop*, 1998.

[6] Ball, T., Levin, V., Rajamani, S. K. A Decade of Software Model Checking with SLAM, *Communications of the ACM*, Vol. 54. No. 7, 2011, Pages 68-76.

[7] Bass, L., Clements, P. and Kazman, R. *Software Architecture in Practice, Third Edition.* Addison Wesley, 2012.

[8] Boehm, B. and Turner, R. *Balancing Agility and Discipline: A Guide for the Perplexed.* Addison-Wesley/Pearson Education, 2003.

[9] Booch, G., Rumbaugh, J, and Jacobson, I. *The Unified Modeling Language User Guide, 2nd Edition*. Addison Wesley, 2005.

[10] Bosch, J. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. ACM Press/Addison-Wesley Publ. Co. 2000.

[11] Bosch, J. From software product lines to software ecosystems. In *Proceedings of the 13th International Software Product Line Conference* (SPLC '09, Pittsburgh, PA, USA, 111-119. 2009.

[12] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. *Pattern Oriented Software Architecture: A System of Patterns, Volume 1*. Wiley, 1996.

[13] Cheng, B., de Lemos, R., Giese, H., et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: *Software Engineering for Self-Adaptive Systems*. Lecture Notes in Computer Science, 5525. Springer,, pp. 1-26. 2009.

[14] Clements, P. and Northrop, L. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2002.

[15] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R. and Stafford, J. A. *Documenting Software Architectures: Views and Beyond, Second Edition*. Addison-Wesley, 2011.

[16] Clements, P., Kazman, R., Klein, M. *Evaluating Software Architectures: Methods and Case Studies.* Addison Wesley, 2002.

[17] Coglianese, L. and Szymanski, R., DSSA-ADAGE: An Environment for Architecture-based Avionics Development. In *Proceedings of AGARD'93*, May 1993.

[18] Cunningham, W. The WyCash Portfolio Management System. Proc. OOPSLA, ACM Press, 1992.

[19] Dashofy, E. M., Van der Hoek, A., and Taylor, R. N.. A Highly-Extensible, XML-Based Architecture Description Language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001)*, Amsterdam, Netherlands.

[20] De Lemos, R., et al. Software engineering for self-adaptive systems: A second research roadmap. *Software Engineering for Self-Adaptive Systems II*. Lecture Notes in Computer Science, Vol. 7475:1-32, Springer, 2013.

[21] Denne, M. and Cleland-Huang, J. The Incremental Funding Method: Data-Driven Software Development *IEEE Software*, 21(3), pp. 39-47, 2004.

[22] Dijkstra, E. W. The structure of the "THE" – multiprogramming system. *Communications of the ACM*, 11(5):341-346, 1968.

[23] Fairbanks, G. *Just Enough Software Architecture: A Risk-Driven Approach.* Marshal and Brainerd, 2010.

[24] Feiler, P., Gluch, D. Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley, 2012.

[25] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1995.

[26] Garlan, D. and Perry, D. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[27] Garlan, D. and Schmerl, B. Architecture-driven Modelling and Analysis. In T. Cant editor*, Proc. of the 11th Australian Workshop on Safety Related Programmable Systems (SCS'06)*, Vol. 69 of Conferences in Research and Practice in Information Technology, Melbourne, Australia, 2006.

[28] Garlan, D. Software Architecture: a Roadmap. In A. Finkelstein editor, *Proceedings of the Conference on The Future of Software Engineering*, Pages 91--101, ACM Press, 2000.

[29] Garlan, D., Allen, R. and Ockerbloom, J. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17-28, November 1995.

[30] Garlan, D., Allen, R. and Ockerbloom, J. Architectural Mismatch: Why Reuse is Still So Hard. *IEEE Software*, pp. 66-69, July 2009.

[31] Garlan, D., Allen. R. and Ockerbloom, J. Exploiting style in architectural design environments. In *Proc of SIGSOFT'94: The second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 170-185. ACM Press, December 1994.

[32] Garlan, D., Barnes, J. M., Schmerl, B. and Celiku, O. Evolution Styles: Foundations and Tool Support for Software Architecture Evolution. In *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture* 2009 & *European Conference on Software Architecture* 2009, September 2009.

[33] Garlan, D., Dwivedi, V., Ruchkin, I. and Schmerl, B. Foundations and Tools for End-User Architecting. In D. Garlan and R Calinescu editors, *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, vol. 7539:157-182 of Lecture Notes in Computer Science, Springer, 2012.

[34] Garlan, D., Monroe, R. and Wile, D. Acme: Architectural Description of Component-Based Systems. In *Foundations of Component-Based Systems*, Pages 47-68, Cambridge University Press, 2000.

[35] Garlan, D., Schmerl, B. and Cheng, S. Software Architecture-Based Self-Adaptation. In M. Denko, L. Yang

and Y. Zhang editors, *Autonomic Computing and Networking*. Springer, 2009.

[36] Garlan, D., Siewiorek, D., Smalagic, A. and Steenkiste, P. Project Aura: Towards Distraction-Free Pervasive Computing. *IEEE Pervasive Computing*,. 1(2):22-31, April 2002.

[37] Garlan. D., Shaw, M., Okasaki, C., Scott, C., and Swonger, R. Experience with a course on architectures for software systems. In *Proceedings of the Sixth SEI Conference on Software Engineering Education.* Springer Verlag, LNCS 376, October 1992.

[38] IEEE Standard 1471-2000. IEEE Recommended Practice for Architectural Description for Software-Intensive Systems. http://standards.ieee.org/findstds/standard/1471-2000.html

[39] Kephart, J.O., Chess, D.M. The vision of autonomic computing. *IEEE Computer*, vol.36, no.1, pp.41,50, Jan 2003.

[40] Kruchten, P. B. The 4+1 view model of architecture. *IEEE Software*, pages 42-50, November 1995.

[41] Kruchten, P., Nord, R. and Ozkaya, I. Technical debt: from metaphor to theory and practice, *IEEE Software*, 29(6), pp. 18-21, 2012.

[42] Kruchten, P., Obbink, H., Stafford, J. The Past, Present, and Future for Software Architecture. *IEEE Software.* pp. 22-30 March/April (vol. 23 no. 2), 2006.

[43] Lattanze, A. *Architecting Software Intensive Systems: A Practitioners Guide.* CRC Press. 2009.

[44] Magee, J., Dulay, N., Eisenbach, S. and Kramer, J. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference*, ESEC'95, September 1995.

[45] Maranzano, J. F., Rozsypal, S.A., Zimmerman, G.H., Warnken, G.W., P.E. and Wirth, Weiss, D.M.. Architecture reviews: practice and experience. *IEEE Software,* 22(2), pp.34-43, March-April 2005.

[46] Medvidovic, N. and Taylor, R. N. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pages 70-93,January 2000.

[47] Medvidovic, N., Oreizy, P., Robbins, J. E. and Taylor, R. N. Using object-oriented typing to support architectural design in the C2 style. In *SIGSOFT'96: Proceedings of the 4th ACM Symposium on the Foundations of Software Engineering*. ACM Press. Oct 1996.

[48] Metzger, A, Pohl, K. Software Product Line Engineering and Variability Management: Research Achievements and Challenges. FOSE'14, May 31 – June 7, 2014, Hyderabad, India.

[49] Parnas, D. L. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5:128-138, March 1979.

[50] Parnas, D. L., Clements, P. C. and Weiss, D. M. The modular structure of complex systems. *IEEE Transactions on Software Engineering*. SE-11(3):259-266, March 1985.

[51] Perry, D. E. and Wolf, A. L. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40-52, October 1992.

[52] Rechtin, E. *Systems architecting: Creating and Building Complex Systems*. Prentice Hall, 1991.

[53] Schmidt, D., Stal, M., Rohnert, H. and Buschmann, F. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects.* Wiley, 2000.

[54] Shaw, M. Architectural Requirements for Computing with Coalitions of Resources. *In Proceedings of the 1st Working IFIP Conf. on Software Architecture*, February 1999.

[55] Shaw, M. and Clements, P. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of COMPSAC 1997*, August 1997.

[56] Shaw, M. and Clements, P. The Golden Age of Software Architecture. *IEEE Software,* 23(2), pp. 31-39. March/April 2006.

[57] Shaw, M. and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[58] Shaw, M. Toward Higher-Level Abstractions for Software Systems. Proceedings of the Tercer Simposio Internacional del Conocimiento y su Ingerieria, October 1988 (printed by Rank Xerox), pp.55-61. Reprinted in Data and Knowledge Engineering, 5 (1990) pp.19-128.

[59] Sullivan, K. J., Chalasani, P., Jha, S. and Sazawal, V. Software Design as an Investment Activity: A Real Options Perspective, in *Real Options and Business Strategy: Applications to Decision Making*, L. Trigeorgis (ed.), Risk Books, 1999.

[60] Taylor, R. N., Medvidovic, N. and Dashofy, E. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.