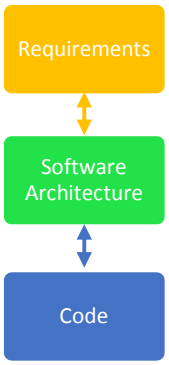
Software Architecture: A Travelogue

By David Garlan

This paper recounts the history of the architecture, its current state of practice and research and speculates on some of the important emerging trends, challenges and aspirations.

Software architecture is a gross structure which illuminates the top cover design decisions. It typically plays a key role as a bridge between requirements and implementation.



Software architecture can play an important role in at least six aspect of software development:

- Understanding: architecture design exposes high level constraints on system design as well as the rationale for making specific choice.
- Reuse: It supports reuse of both components and also frameworks into which component can be integrated.
- Construction: It provides a partial blurring for development by indicating the major components and dependencies between them.
- Evolution: It can expose the dimensions along which a system is expected to evolve.
- Analysis: It provide opportunities for analysis including system consistency checking, satisfaction of quality attributes and domain specific analysis for architecture built in specific styles.
- Management: Critical evaluation of it leads to a much clearer understanding of requirements, implementation strategies and potential rises.

Past

In the early decades of software engineering, architecture was largely ad-hoc affair. Descriptions typically relied on informal box-and-line diagram 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. It has been central to system design.

Within industry, two trends highlighted the importance of architecture:

- Recognition of a shared repertoire of methods, techniques, patterns and idioms for structuring complex software systems.
- Concern with exploiting commonalities in specific domains to provide reusable frameworks for product families.

Current Scenario

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.

The technological and methodological basis for architectural design has improved dramatically. Four important advances have been

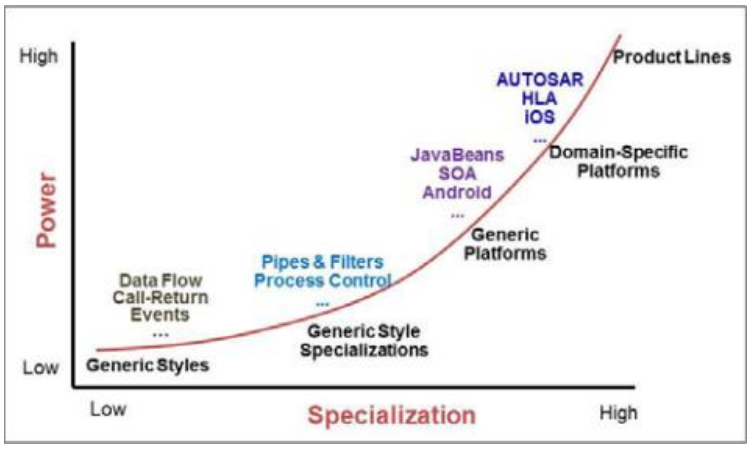
(1) the codification and dissemination of architectural design expertise
(2) the emergence of platforms and product lines, and their associated ecosystems
(3) the development of principles, languages and tools for architecture description
(4) the integration of architectural design into the broader processes of software development, and, in particular the relationship between architecture and agility

Codification and Dissemination

An architectural style typically specifies a design vocabulary, constraints on how that vocabulary is used, and semantic assumptions about the vocabulary. Other common styles include blackboard architectures, client-server architectures, repository-centered architectures, event-based architectures, N-tiered architectures, and service-oriented architectures. 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. Such techniques are sometimes referred to as architectural tactics. A number of tactics have been developed to perform mismatch problems and people are starting to see the introduction of automated mismatch repair tools.

Platforms and Product Lines

Two specific manifestations of trend (exploitation of commonality across multiple products in order to reduce development costs for new system through customization of a shared asset base) are improvements in our ability to create product lines within an organization and the emergence of cross-vendor platforms. Like architectural styles, the architectures underlying platforms and product lines take advantage of common architectural structures, but do so in a domain-specific way.



Three general approaches for architecture description

- Informal description
  - Use general-purpose graphical editing tool
  - easy to produce, and not requiring special expertise
  - cannot be formally analyzed for consistency, completeness, or correctness
- Semi-formal description
  - may lack detailed semantics, but provide a standardized graphical vocabulary supported by commercial tools
  - use 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
  - languages are limited by their lack of support for formal analysis and their lack of expressiveness for some architecturally relevant concepts
- Fully-formal description
  - referred to as "Architecture Description Languages" (ADLs)
  - notations provide both a conceptual framework and a concrete syntax for formal modeling of software architectures
  - typically provide tools for parsing, displaying, compiling, analyzing, or simulating architectural descriptions
  - challenges in this area include

- understanding when a system of partial views provides a complete representation of all aspects relevant to the design
- providing formal criteria for view consistency, including both structure and semantics
- using views to capture refinement relationships between abstract and concrete architectural designs

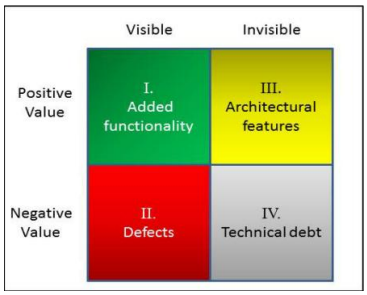
Architecture is, in fact, an enabler of agility, not an impediment to it

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

Architectural design, when it is really needed because of project novelty, has an uneasy relationship with traditional agile practices

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)

Four Types of Backlog Elements



- Category I (Green**)**: new features to be added to the system, as well as visible improvements in quality attributes.
- Category II (Red**)**: customer-visible defects, limiting usefulness or negatively impacting perceptions of the product.
- Category III (Yellow**)**: 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 (Grey**)**: 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. 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.

Challenges Ahead

Network-Centric Computing

Software systems have been created as closed systems, developed and operated under control of individual institutions

Architectures for such systems are largely static – allowing minimal run-time restructuring and variability

Within the world of pervasive services and applications available over networks, systems may not have such centralized control

There is no central authority for control or validation. Individual developers can provide, modify, and remove resources at will

New set of software architecture challenges:

- First, is the need for architectures that scale up to the size and variability of the Internet.
- Second, is the need to support computing with dynamically formed, task-specific, coalitions of distributed autonomous resources.
- Third, there is a need to architect systems that can take advantage of the rich computing base enabled by network-centric computing.
- Fourth, there is a need to ensure adequate security and privacy.

Pervasive Computing and Cyber-physical Systems

Referred to as the "Internet of Things"

New set of software architecture challenges:

- First, we will need architecture design tools that are suited to systems that combine both physical and software elements – cyber-physical systems.
- Second, architectures for these systems will have to be more flexible than they are today.
- Third, there is a need for architectures that effectively bridge the gap between technology and technologically-naïve users.

Fluid Architectures

New set of software architecture challenges:

- The first is architecting for ease of change and adaptation. The architect needs to be sure that such changes do not interfere with on-going services that must continue to function.
- Second is the problem of describing and reasoning about the architecture. The combinatorial explosion of possible future system configurations makes formal analysis difficult.

Socio-technical Ecosystems

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.