Only the Architecture You Need

Richard N. Taylor

1 Introduction

Software architecture has been around for a long time. Even prior to the identification of software engineering as a discipline in 1968, there was an explicit focus on techniques for software design. The 1970s saw many publications detailing various design techniques and strategies. In 1976 Peter Freeman stated, “Design is relevant to all software engineering activities and is the central integrating activity that ties the others together” [1]. More design techniques and strategies emerged in the 1980s, many of them addressing larger-scale systems. “Software architecture” as the label for this type of work took off in the 1990s, notably with the appearance of Perry and Wolf’s landmark paper [2]. Subsequent development of the field focused on various types of architectural models, description languages, analysis techniques, development environments, canonical solutions, and design processes. Example architectures abounded, conferences and workshops held, and many books emerged.

Yet despite all this progress, all too often architecture is ignored in application development. Consider the following dialog from an imagined movie, “The Treasure of the Silicon Valley,”[1] starring a venture capital investor performing due diligence for a potential acquisition, conversing with a start-up’s lead software developer:

> If you’re the chief software engineer on the project, show me your architecture.
>
> Architecture? Architecture?! We don’t need no stinkin’ architecture!

[1]With acknowledged inspiration from, and apologies to, “The Treasure of the Sierra Madre,” a 1948 film by John Huston.

R. N. Taylor (✉)
Institute for Software Research, University of California, Irvine, Irvine, CA, USA
e-mail: taylor@ics.uci.edu

© The Author(s) 2018

V. Gruhn, R. Striemer (eds.), *The Essence of Software Engineering*,
https://doi.org/10.1007/978-3-319-73897-0_5

This dialog is all too understandable. The VC wants to know what he's buying into and wants to perform his own analysis of the properties of the start-up's system. And most assuredly, he does not want to have to read a million lines of undocumented Python and JavaScript to get that insight. The developer, on the other hand, has been immersed in the details of the application since day one. He knows what he has, believes in what he has, and sees any call for an explicit architecture as a nonproductive demand on his nonexistent free time.

Very different dialogs appear in other "movie scripts." Developers in regulated industries, or those working under government contract, are well acquainted with the use of architectural models to facilitate communication and to demonstrate achievement of some mandated properties. What is appropriate and what is necessary can vary widely, just as projects and usage contexts vary widely.

The remainder of this chapter, then, considers several different development contexts, ranging from "personal software engineering" to large-scale organizational development of high-consequence software. For each we consider what kinds of architectural discipline are needed and what purposes such architectural information serve. Our perspective is one of cost-benefit analysis. Investment in architectural modeling and analysis should not exceed the benefits reaped by performing such tasks.

2 Software Architecture: Essence, Benefits, and Costs

Before considering the various development contexts and how they differ in terms of their need for architectural discipline, we provide a little background on software architecture and introduce some key terminology. This is not a full presentation of the key elements of software architecture, but rather a quick highlight of a few concepts that will appear throughout the remainder of this chapter. Software architecture is a well-developed field with numerous techniques and strategies developed to aid the architect. Many of these, along with careful definitions of the rich vocabulary, are fully presented in [3].

To begin, software architecture, as a term, derives from analogy to the architecture of buildings. The analogy, while imperfect, is strong and provides several key insights:

- Architecture exists independently from the building/source code.
- The properties of structures (whether buildings or code) are induced by their architectures, for example, how accommodating of change they can be.
- The necessary skills of an architect are different from the skills of a building contractor/programmer.
- The *process* of design and construction is not as important as the architecture (i.e., the *product* is ultimately what matters at the end of the day, rather than how you got there).

- Architecture is a body of knowledge that can be studied, taught, and improved.
- Every building/application has an architecture, whether implicit or explicit, whether good, bad, ugly, or elegant.

The best definition of software architecture is that articulated by Eric Dashofy and put forth in [3].

Definition A software system's *architecture* is the set of principal design decisions made about the system.

This definition places the notion of *design decision* upfront. Design decisions encompass every aspect of the system under development, including structure, functional behavior, nonfunctional properties, user interaction, and decisions related to the system's implementation and deployment. Every application embodies at least one design decision, and hence all systems have architectures.

Not all design decisions carry equal weight, however. *Principal* is a key modifier of "design decisions." It is a matter of degree and pertinence that grants a design decision "architectural status," that is, that makes it an *architectural design decision*. This also implies that not all design decisions are architectural. Indeed, many of the design decisions made in the process of system building (such as the programming details of the selected algorithms) will not impact a system's architecture.

Determining which decisions are principal is a function of context. It is the system's stakeholders (including, but not restricted to, the architect) who rightfully decide which design decisions are important enough to include in the architecture.

Given that stakeholders may come with very different priorities from a software architect, even nontechnical considerations may end up driving determination of the architecture. Moreover, different sets of stakeholders may designate different sets of design decisions as principal. Thus this definition of software architecture is neither simplistic nor simple. Architecture concerns the core decisions, and in a significant system those decisions do not come automatically or without dispute.

Architectural models are means of capturing architectures in a tangible form. Once again from [3], we have these definitions.

Definitions An architectural *model* is an artifact that captures some or all of the design decisions that comprise a system's architecture. Architectural *modeling* is the reification and documentation of those decisions. An *architecture description language* (ADL) is a notation for capturing architectural decisions as a model.

Lastly, we consider architectural styles, whose role will figure prominently in the subsequent discussion.

Definition An *architectural style* is a named collection of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system.

Many architectural styles are widely known, such as client-server, event-based, REST (REpresentational State Transfer) [4], and SCADA. Styles are essential tools in a software designer's toolbox. Styles capture the hard-won lessons of past experience, enabling a designer to reap known benefits in specified contexts in a new design.

2.1 Benefits

The benefits sought through a focus on software architecture include:

- Effective communication
- Conceptual integrity: intellectual control and management of complexity
- Adequate basis for supporting knowledge reuse
- Support for cost-effective product lines, including management of related variants

The most widely acknowledged benefit of a focus on software architecture is improved communication. That communication may be among developers or between developers and various stakeholders. Seemingly ubiquitous PowerPoint presentations of system designs, with circles, boxes, arrows, and colors, are attempts to communicate some of the key design decisions of a system. (Whether those attempts are effective or accurate is an entirely different subject.) Whatever the means of modeling, the objective is to communicate the essential decisions to others so that, for example, developers can proceed with their tasks knowing the context into which their work fits, or so that other designers can offer their opinions about the suitability of the design, based on their analysis of the represented decisions.

Maintaining conceptual integrity of a system as it evolves over time is perhaps the greatest challenge to a project manager. As systems evolve, responding to pressures for additional or changed features, new platforms, or simply to fix bugs, it is easy for their architectures to drift from their original key decisions. Knowing whether a new decision is consistent with key decisions made previously is of fundamental importance. Determining such consistency demands that there be a record of what those decisions are, and that is the function of explicit models of the architecture. With a model there is at least the hope of assessing the impact of a newly proposed change; without a model the project manager is left with only his memory.

Knowledge reuse is essential to the economic success of an enterprise; rediscovering insights and reinventing solutions is a recipe for failure. Knowledge reuse on a small scale became well known and popularized in the 1990s through use of design patterns: solutions to small-scale problems that are nonetheless common in programming [5] and which have been subsequently captured for reuse by others. Stepping up in scale to subsystems of modules and to whole applications, architectural styles enable developers to similarly reuse solutions captured through prior experience, thereby achieving the benefits yielded by adherence to those styles.

On a still larger scale, companies often flourish when their products dominate a market segment. Dominating a segment often results from acquiring deep knowledge of the domain and having experience developing multiple solutions to problems in the domain. New products in a domain are often incremental variants of prior products that leverage that knowledge. A domain-specific reference architecture can capture these insights in ways similar to design patterns and architectural styles and can provide guidance for the design and management of software product lines.

2.2 Techniques . . . and Costs

The literature of software architecture is full of techniques, strategies, languages, and tools intended to help the architect from initial conception of a system through its full product lifetime. Virtually all of the techniques center on, or require, some form of architectural model. Models form the basis for communication, analysis, and, if they are good models, implementation and evolution.

Modeling languages run the gamut from informal and shallow (most PowerPoint architectures) to technically rich and deep, upon which formal and automated analyses can be performed, and in some cases from which implementations can be automatically generated, or at least started.

Doing a good job of modeling—in which the key decisions are all identified and captured—is not an easy or quick task. Modeling languages are not all easy to use. Indeed, generally speaking, the easier a modeling language is to use, the less information it captures and the less useful it will be as a project proceeds; conversely, the most powerful languages have narrow ranges of applicability and can require costly and rare expertise to effectively employ.

The key issue in the application of any software architecture technology is the cost-benefit ratio.

Capturing knowledge, of the kind that enables new products in a domain to be built efficiently, is also costly. It requires, as new products are built and experience gained, that an investment be made in reflecting on that experience and refining domain models and architectures for potential future use. The potentiality is a risk; if the captured knowledge is not reused later in new products, then that effort was wasted.

Risks exist too, based on the current state of software architecture tools and techniques. Some modeling languages, for example, may provide significant benefit to architects during initial design stages, through facilitating communication and providing the basis for analysis. But when it comes time to push the design into implementation, the modeling language may provide little help. Indeed, with many languages, the task of showing conformance between the architecture of the code (the realized architecture) and the architectural model (the intended architecture) may be quite difficult. Moreover, when problems arise during implementation and a need for change to the intended architecture is identified, many design tools provide no help in "mapping back" to enable a disciplined approach to the redesign.

2.3 Summary and Roadmap

The benefits that have the potential to be realized through a disciplined application of software architecture are many and substantial. But the costs can be significant. The key, then, is to understand the demands of a development context, and for that context identify just the architecture techniques that are cost-effective. The following sections of the chapter will attempt to briefly do just that, examining three notional development settings: personal software engineering (working by yourself, for yourself), working in a team in a small corporate setting, and working in a large company on high-consequence software.

3 Personal Software Architecture

The imaginary screenplay between the venture capitalist and the start-up entrepreneur found in the Introduction section could very plausibly arise from a common scenario. An individual, the entrepreneur, learns programming in college and then decides to use his new skills by writing an app for his iPhone—writing it both for the pleasure and interest of doing so, and also because he has a particular way he likes to plan and record his vacations. His app, “MyTravel,” allows him to record an itinerary, include photos and commentary, and export to his personal blog. Naturally by doing good development work, his friends are impressed, want their own copy, and later ask him to add additional features. By word of mouth the popularity of the app increases until the group of friends decides to form a small company to further enhance and market the product. Sometime later as success grows, the need for venture capital appears and the “no stinkin’ architecture” dialog ensues.

But why would the inventor be so resistant to talking about architecture? Simply because of how his company evolved. At the outset of his efforts, he was just “messing around” and the project just accreted features in a haphazard fashion after that. He was just working for himself, with no intention of ever forming a company. He never took the time to focus on “architecture.”

There is a deep falsehood in this narrative, however. Unless the developer was truly ignorant, he will have used Apple’s app developer tools, such as Cocoa, the XCode software development kit, the Quartz framework, and user interface guidelines. And prominent in those materials is this statement: “MVC is central to a good design for a Cocoa application[2].” That is, Apple is directing developers’ attention to a particular architectural style, MVC (Model-View-Controller), and saying that it is of critical importance in the design of new iPhone applications. The Apple website goes on to say, “The benefits of adopting this pattern are

[2]https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html, Accessed July 2017.

numerous. Many objects in these applications tend to be more reusable, and their interfaces tend to be better defined. Applications having an MVC design are also more easily extensible than other applications. Moreover, many Cocoa technologies and architectures are based on MVC and require that your custom objects play one of the MVC roles." Thus the individual developer is compelled from the outset to be knowledgeable about and to utilize an important concept from software architecture.

The need for personal architectural knowledge goes beyond MVC, however. iPhone applications, and indeed virtually every user interface-intensive application, rely heavily on event-based architectural concepts. From handling interrupts through publish-subscribe architectural styles, to highly decoupled applications, event-based styles are powerful and common. Their power arises from supporting strong decoupling (which indeed may cross host and address space boundaries) and unpredictable sequences of events, leading to high extensibility. They are seductive and dangerous, however, because while an individual developer may initially only use events to coordinate two or three actors in an application, mentally keeping track of how the events are handled, the situation can quickly become confusing. Indeed, unwelcome surprises may appear as the developer slowly starts to recognize all the possible event interactions that may occur.

Cocoa and Quartz introduce even more architectural concepts into the solo entrepreneur's world. Cocoa's AppKit is a framework used to implement the user interface of an application. Quartz is a framework used to manipulate images. A framework is a programmatic bridge between concepts (such as "window" or "image") and lower-level implementation technologies. Frameworks can be very architectural in orientation, wherein they cleanly map architectural styles to code; unfortunately, they may also be rather random collections of nonetheless useful code.

So, indeed, our intrepid entrepreneur was using multiple explicitly architectural concepts from the outset, though he may have not known them under that label.

As time progresses, a key question for the entrepreneur is whether his memory is sufficient to remember all the design choices he has made, and to make future changes to his application in a manner that is consistent with the previously made decisions—or at least to be able to recognize when a prior decision is being changed, and then to understand all the downstream consequences of that change. To what extent has he bothered to record his design decisions in some accessible medium?

The cost-benefit analysis for a developer working on his own, for himself as a client, is pretty simple, though the analysis itself is "risky." He must know the concepts of software architecture, for they will surely figure into his development. But to what extent does he need to invest in capturing his decisions in a model of some sort? If his memory is good, or if the application has a very limited time horizon, then little or no such investment is warranted. But if there is a chance that the usage and development will progress to a context beyond that of personal development, then the investment might be very well worth it. It is that context that we examine next.

4 Team Software Architecture

The role of software architecture grows in importance dramatically as development shifts from an individual working for himself to a team developing a product for use within a company or for external sale. The consequences of poor decisions or poor engineering are much more severe, the need for communication within the team and across the company is critical, and the complexity of the product and project is much greater.

Though the sources of increased complexity are perhaps obvious, as an example consider some of the possible ways the "MyTravel" app could grow. The app could support ingestion of travel itinerary information from a wide range of external travel vendors by importing the information from confirmation emails. The app could enable export of selected information to Facebook. A dramatically larger user base could drive the back end of the app to the cloud, improving access and providing scalability. With wide usage and broad third-party integration, the need for security and authentication arises, very possibly supported by services available in the cloud.

4.1 Communication

Perhaps the single greatest need for architecture in this context arises in support of communication. Not only must all developers be on the same page, but also the development team must meet its accountability obligations, both to management within the company and potentially to external clients. If the architecture is not captured in any tangible form, then communication is limited very unsatisfactorily to mental recollections and verbal communication. If, rather, explicit models are used, both during design and during subsequent system evolution, then they may serve as the anchor for all types of communication. But what kinds of models? The options range from informal text and diagrams ("PowerPoint architectures") to semiformal UML, to precise architectural specifications in a formal architecture description language. The choice turns on what benefits are needed from the models, both immediately and also over the project's lifetime. The typical tragedy, unfortunately, is that this key decision is often made thinking only of immediate communication needs, such as to satisfy some corporate review board, and not, for example, with an eye to how a different downstream development group is going to attempt to decide through reviewing past documents whether a particular program reorganization will break security properties of the system. Making the choice of what to model and how to model, then, requires professional maturity and engineering discipline, qualities that are often in short supply.

Furthermore, the larger the team and the more diverse the set of project stakeholders, the greater the need for specialized visualizations, or projections, of a chosen architectural model. When talking to a customer, for instance, a less detailed view of the architecture is probably desirable. When internally reviewing the architecture for performance properties, a more detailed view is likely essential. Support for multiple views of a single model is unfortunately uncommon among modeling techniques.

4.2 Complexity

The complexity of a project is often the initial reason for creating a team to develop it. And often as a project evolves, its complexity only increases, as additional interconnections with external systems increase, additional features are added, and new usage modes considered. While the development team may initially select a satisfactory, coherent architecture, how is intellectual control over that architecture maintained over time? As features accrete, how does the team prevent the once-clean architecture from slowly devolving into a "big ball of mud?" Certainly explicit models and team discipline are part of the answer, but additional success can be found in judicious choice of architectural style; some styles are far more accommodating of change than others. Indeed, some styles, such as event/message based, are explicitly designed to foster and accommodate change.

Accommodating the need to *scale* to new dimensions of users, or data, or platforms is a particularly common and particularly difficult type of change to satisfy unless good architectural engineering is applied. Coping with issues of scale requires understanding the source of scaling pressures and then choosing or combining techniques suited to meet those challenges. The techniques available include, obviously, choice of architectural style and, especially, choice of appropriate connector technology. Further, with suitable modeling some types of scalability analysis can be undertaken prior to any system implementation.

A compelling illustration of this approach is the World Wide Web. The well-known REST architectural style mentioned earlier [4] arose from a careful consideration of the demands on an open, network-based, hypertext-oriented information integration system. REST was developed as a judicious combination of multiple simpler and well-known styles, such as replicated data, client-server, layered, and virtual machine, plus some additional constraints.

Through a team's shared understanding of a style—why it was chosen, what its constraints are, what benefits it elicits—and maintaining adherence to that style, a system's conceptual integrity can be preserved.

Essential to understanding and applying advanced architectural styles is understanding the rich range of connector technologies available. Perhaps the most common weakness in the educational background of new developers is lack of understanding of such connectors, and how different types of connectors can contribute to keeping system complexity under control while achieving scale and extensibility goals. Many new developers come armed only with method calls, the simplest and perhaps most limiting of all connector types. More capable connectors often come hidden inside various middleware packages. "Enterprise software buses," for example, can provide a much richer and dynamic connection style, and the many network protocols offer still other connection means. For a full treatment of connectors, see [6] or [3].

5 Summary

In a team setting the costs of development are higher than in the individual setting and the consequences of poor engineering are similarly higher. What was sufficient practice in the individual developer context is no longer adequate. There is a greater need for planning, and much greater need for support of communication between developers, customers, and management. Failure to adequately prepare for future years of product development and modification can lead to numerous downstream costs and problems, from scalability issues to costly errors and data breaches. How much architecture do you need? Again, the balance is between the costs of applying it and the benefits so realized. The essentials, for most projects and teams, are:

- Explicit modeling using some form of architectural description technology to support communication and analysis
- Broad knowledge of and ability to select and apply a range of architectural styles
- Similar depth in understanding and applying connector technologies

What happens when the organizational setting is even more consequential? For example, when the product competes in a governmentally regulated industry, or when the organization develops not just one product in a market space, but a large family of related products? We consider this context next.

6 High-Consequence Software

"High-consequence software" may mean software whose usage occurs in contexts where errors or failures can result in death or large-scale economic loss. The phrase may equally be used to describe software whose economic importance is so great that the fortunes of a company, or even an ecosystem of companies, may rise or fall depending on issues with the software. In either situation, the implication is that the context in which the software is designed, built, and used is larger still from what we have considered so far. Diverse (and distant) user communities may be involved, regulatory oversight may exist within multiple jurisdictions, and the development "organization" may no longer be a single company, but involve a cooperating set of agents. The "software" too may no longer represent a single application, but a family of related products, with variants for differing usage contexts. In many dimensions, then, the context is more complex and the stakes are higher.

The large consequences of key decisions and the large risks that may be entailed imply that professionalism is not an option, it is essential. But given the wide diversity of high-consequence software, what aspects of software architecture technology are essential?

First, in any high-consequence situation, is an even greater need for attention to techniques previously mentioned. Increased investment in modeling is warranted in support of communication, addressing the larger and more diverse community of stakeholders. Because of increased size and system complexity, specialized projections of the model likely will be needed. Independent regulatory oversight may demand specific, particular projections of a system in order to demonstrate compliance properties.

Similarly needed is a clear focus on and identification of key styles. As an illustration, consider the importance of precisely identifying the plug-in architecture for supporting third-party extensions to Adobe's Photoshop product, Apple's identification of the role of the MVC style, or the Web's identification of the REST style. In these cases, and many similar ones, explicit styles are key to enabling an ecosystem of independent development organizations to cooperate and mutually thrive.

Beyond such increased attention to the previously discussed techniques, two additional emphases deserve brief discussion. The first is domain-specific software architectures and its closely related cousin, architecture-based product families.

DSSAs and product families spring from slightly different origins but end up in a similar place. The key notion of a DSSA is capture and reuse of deep domain knowledge and experience with developing solutions within that domain. The key notion of a product family is management of related product variants. Seen together the notion is exploitation of deep experience (domain and solution) through management of a family of related products—in short, a technically based product line[3]. The technical bases for such product lines are configuration management, domain knowledge capture, and reference architectures. These concepts merge with architectural styles and explicit modeling to yield careful management, product generation, and highly efficient platform and market specialization.

Configuration management is a well-understood, universally practiced discipline, at least in its simplest form: version control. The focus in the high-consequence context is sensibly managing the relationships between features, deployment platforms, and architectural entities.

Domain knowledge capture is the discipline of effectively recording the fundamental characteristics of an application domain in such a way that new products in the domain can be described using terminology that enables unambiguous description of novel requirements as well as clear mapping of continuing requirements to concepts and entities in prior products. Domain knowledge may well be half of a development organization's competitive advantage; the other half is based in its experience with prior solutions in that domain.

[3]We explicitly distinguish this concept from "product lines" that are nothing more than applying a uniform marketing badge on products having no common technical foundation.

When an organization reflects on its product experiences and captures effective solution strategies (i.e., architectural decisions) in a form that supports reuse, those strategies are termed a reference architecture. A good reference architecture is a company's "secret sauce"; it is the knowledge that enables it to produce new solutions within a domain faster and cheaper than its competitors. It is an architectural style on a very large, grand scale. The fundamental question, though, is, how is that knowledge, that reference architecture, captured? Often it is merely in the heads of the company's lead engineers. What happens if those engineers resign? The cost-benefit analysis must consider how difficult and expensive it will be to invest in reifying that knowledge, versus the potential downsides should the key engineers depart to work for competitors.

The second additional emphasis is security engineering. Data breaches and security violations seem only exceeded in news articles by hyperventilation over AI and big data. The ubiquity of the problem, and the inability of repeated patches to do anything more than slightly delay the next problem, indicates that security is not an add-on feature. Security properties must be considered from the outset of a system's design. Indeed, it must be a key element in designing a system's architecture. Explicit architectural models are a starting point for security analysis and design. Consider the alternative—if the key design decisions are not recorded and made analyzable, then how can an engineer determine whether there are system vulnerabilities? Given the enormous range of system designs, there is little in general that can be said about designing for security properties, but an emerging view is that no perimeter defense will ever be sufficiently protective for decentralized systems [7, 8]. Rather, security must be considered at all levels of design, from the most abstract architecture to specific coding choices. In any event, explicit consideration is necessary.

7 Conclusion: Excuses Are Not Strategies

"We don't need no stinkin' architecture!" Really? Young companies always seem to have time to address today's emergency, but never the time to engineer at the right time to prevent future emergencies. That is an excuse, not a strategy.

Excuses are not strategies, but neither is untempered exhortation to use every type of software architecture technology. In the end, it is a cost-benefit analysis that must be applied, but it must be an analysis that looks beyond the next quarter's earnings report. Indeed, it must look to a significant product horizon. Only mature organizations can afford to do that, but only mature organizations survive.

Acknowledgments My understanding of software architecture has been enriched immensely through my long-standing collaborations with Professor Nenad Medvidovic of the University of Southern California and Dr. Eric Dashofy of The Aerospace Corporation.

References

1. Freeman, P.: The central role of design in software engineering. In: Freeman, P., Wasserman, A. (eds.) Software Engineering Education, pp. 116–119. Springer, New York (1976)
2. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes **17**(4), 40–52 (1992)
3. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice, 736 pgs. Wiley, Hoboken, NJ (2010)
4. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Trans. Internet Technol. **2**(2), 115–150 (2002)
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Addison-Wesley Professional, Reading, MA (1995)
6. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: Proceedings of the 2000 International Conference on Software Engineering, pp. 178–187. ACM Press. Limerick, Ireland, 4–11 June, 2000. http://sunset.usc.edu/classes/cs599_2000/Conn-ICSE2000.pdf
7. Gorlick, M.M., Strasser, K., Taylor, R.N.: COAST: an architectural style for decentralized on-demand tailored services. In: Proceedings of the 2012 Joint IEEE/IFIP Working Conference on Software Architecture (WICSA) & 6th European Conference on Software Architecture, pp. 71–80, IEEE. Helsinki, Finland, August 20–24, 2012. doi:10.1109/WICSA-ECSA.212.15
8. Gorlick, M.M.: Computational state transfer: an architectural style for decentralized systems. Ph.D. Dissertation. Department of Informatics, University of California, Irvine (2016). http://isr.uci.edu/sites/isr.uci.edu/files/techreports/UCI-ISR-16-3.pdf

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.