Software Architecture: Past, Present, Future

Wilhelm Hasselbring

1 Introduction

For large, complex software systems, the design of the overall system structure (the software architecture) is an essential challenge. The *architecture* of a software system defines that system in terms of components and connections among those components [55, 58]. It is not the *design* of that system which is more detailed. The architecture shows the correspondence between the requirements and the constructed system, thereby providing some rationale for the design decisions. This level of design has been addressed in a number of ways including informal diagrams and descriptive terms, module interconnection languages, and frameworks for systems that serve the needs of specific application domains. An architecture embodies decisions about quality properties. It represents the earliest opportunity for evaluating those decisions. Furthermore, reusability of components and services depends on how strongly coupled they are with other components in the system architecture. Performance, for instance, depends largely upon the complexity of the required coordination, in particular when the components are distributed via some network. The architecture is usually the first artifact to be examined when a programmer (particularly a maintenance programmer) unfamiliar with the system begins to work on it. Software architecture is often the first design artifact that represents decisions on how requirements of all types are to be achieved. As the manifestation of early design decisions, it represents design decisions that are hardest to change and hence most deserving of careful consideration.

W. Hasselbring (✉)
Kiel University, Kiel, Germany
e-mail: hasselbring@email.uni-kiel.de

V. Gruhn, R. Striemer (eds.), *The Essence of Software Engineering*,
https://doi.org/10.1007/978-3-319-73897-0_10

In the following, I take a look backwards to the past development of software architecture as a discipline (Sect. 2) and at the present state (Sect. 3) and provide my view on the envisioned future (Sect. 4), before I summarize in Sect. 5.

2 Past: Focus on Architecture Description and Reuse

Long before software architecture emerged as a discipline [55], pioneers such as Parnas [49] and others [19] observed that it is not enough for a software system to produce the correct functions. Other software qualities such as dependability and maintainability are also important and can only be achieved by careful structuring of software systems.

The concept of software architecture as a distinct discipline started to emerge in the 1990s with architecture description languages, formalization, and classification of architectural styles. The study of software architecture has evolved from the seminal work of Perry and Wolf [50], Shaw and Garlan [55], and others. Architecture description languages (ADLs) [42] have emerged to provide formal rigor to architecture representation. The ANSI/IEEE standard, IEEE-Std-1471-2000, aims to codify the best practices and insights of both the systems and software engineering communities in the area of architecture documentation [36].

The structures proven in practice were cataloged and explained as patterns [13]. On the programming level, reuse is usually accomplished by means of high-level programming language constructs, function libraries, or object-oriented class frameworks. On the design level, design patterns and established software architectures are essential. Design patterns [26] are "micro-architectures" while software architectures are more coarse-grained designs. A design pattern describes solutions to a recurring problem. Patterns form larger wholes like pattern languages to provide guidance for solving complex problems. Patterns express the understanding gained from practice in software design and construction. The patterns community catalogs useful design fragments and the context that guides their use.

An architecture description language is a set of notations, languages, standards, and conventions for architectural models. In Sect. 2.1, the formalization of such architectural models is discussed. An architectural model captures part of the knowledge about an architecture for a single system or a family of systems in a domain (i.e., a reference architecture). In Sect. 2.2, the reuse of reference architectures in the context of software product line engineering is discussed.

2.1 Formalization of Architectural Models

In industrial practice, software architectures are usually described informally or semi-formally with diagrams using boxes, circles, and lines together with accompanying prose. The prose explains the diagrams and provides some rationale for

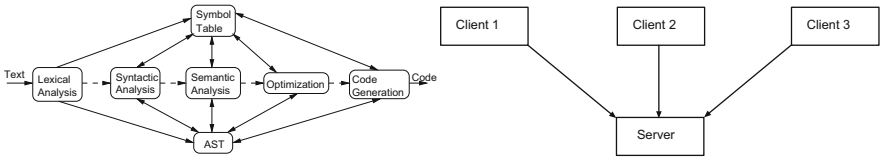

Fig. 1 Typical *pipeline* architecture for the various phases of a compiler (left) and a *client-server* architecture for information systems (right)

the chosen architecture. Typical examples are *pipeline* architectures for the various phases of a compiler or a *client-server* architecture for information systems, as illustrated in Fig. 1. Shaw and Clements called this method "boxology" [54].

Such figures often give an intuitive picture of the system's construction, but the semantics of the components and their connections/interactions may be interpreted by different people in different ways (due to the informality). Thus, such descriptions have been criticized because they lack (formal) semantics. However, they are useful for communication with stakeholders and for project planning. The degree of formality depends on the intended use of architectural models.

The UML is often employed for architecture documentation [40]. However, the UML—as a general object-oriented modeling language—provides only limited support for architecture documentation. For instance, it still lacks basic architectural concepts such as layers.

Formalizations of architecture descriptions developed in parallel with language development [2, 3]. Some specific advantages of formality in software architecture description may be summarized as follows:

- Software architectures become amenable to analysis and evaluation [16]. This helps to evaluate architectures and to guide in the selection of architectural variations as solutions to specific problems.
- Software architectures can be a basis for *design reuse* [24, 53], provided that the individual elements of the architectural descriptions are defined independently and in a precise way. Reusable architectures give designers a *blueprint* in development by helping them avoid typical design errors.
- Software architectures support improved program understanding as a basis for system evolution if its specification is well understood: Retaining the designer's intention about a system organization should help maintainers preserve the system's design integrity [8, 45].
- Formality can allow prototyping for early design evaluation [14].
- Testing may be supported by deriving test plans from formal architectural descriptions [10, 44].
- Proper tool support for designing and analyzing software architectures becomes possible [56].

The recognition that architectural analysis must reconcile multiple views helped to frame the requirements for formalism. An ADL defines a set of notations (e.g., diagrams, formal languages, natural language text fields) for each view that the ADL includes. Architecture models provide one or more views of an architecture. Views highlight certain types of information and hide other types. Examples of well-known architectural views include data-flow control-flow diagrams, state-transition diagrams, data models and entity-relation diagrams, structure charts, and object-oriented hierarchy diagrams.

Formalization of architectural styles aims to allow formal checks of conformance between architecture and implementation, to predict the impact of changes, and to formally reason about a system's architectural description [2]. Various approaches to formalizing architectural styles have been proposed [9, 31, 48]. The formal comparison allows for a detailed analysis of similarities and differences among the architectural variations.

2.2 Software Product Lines for Reusing Software Components

Software architectures usually address the quality attributes for individual systems. Systems in the same domain often have similar architectures that reflect domain concepts. Reference architectures capture architectural commonality of multiple, related systems, that is, systems within the same domain. Reference architectures are central to domain-specific reuse, in that they provide a framework for creating assets and constructing systems within a domain. Domain engineering thus allows for product line development, which seeks to achieve reuse across a family of systems.

A software product line consists of software systems that have some common functionality and some variable functionality [11]. The interest in software product lines emerged from the field of software reuse when developers and managers realized that they could obtain much greater reuse benefits by reusing software architectures instead of only reusing individual software components. The basic philosophy of software product lines is reuse through the explicitly planned exploitation of commonalities of related products and proper management of *variability* in software systems [25].

Product development in software product lines is organized into two stages: domain engineering and application engineering with reuse of software components [29]. The idea behind this approach to application engineering is that the investments required to develop the reusable artifacts during domain engineering are outweighed by the benefits of deriving the individual products during application engineering.

Reference architectures play an important role in domain engineering. *Domain engineering* is an activity for building reusable components, whereby the systematic creation of domain models and architectures is addressed. Domain engineering aims

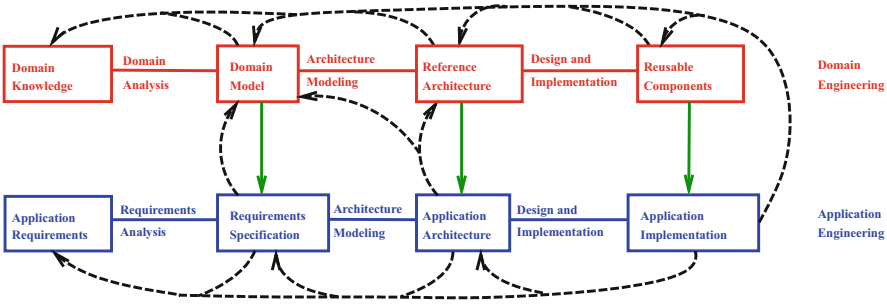

Fig. 2 Domain and application engineering for component-based software product lines [29]. In application engineering, software systems are developed from reusable components created by a domain engineering process. As indicated by the dashed arrows, multifaceted feedback is possible

at supporting *application engineering* which uses the domain models and reference architectures to build concrete systems, as illustrated in Fig. 2 [29]. The domain model characterizes the *problem space*, while the *reference architecture* addresses the *solution space* in domain engineering. The emphasis is on reuse and product lines.

3 Present: Establishment of Domain-Specific Architectures and Focus on Quality Attributes

As discussed in the previous section, the past emphasis was mainly on generic architectural styles such as pipe-and-filter architectures. However, the Domain-Specific Software Architecture (DSSA) engineering process was introduced early in the 1990s to promote a clear distinction between domain and application requirements [57]. A Domain-Specific Software Architecture consists of a domain model and a reference architecture. DSSA was promoted for domains such as avionics.

Meanwhile, various architectures are established for many domains and applications. Where total architectural solutions do not yet exist, partial ones certainly do in the form of catalogs of architectural patterns that help solve many problems and achieve various quality attributes. Various domain-specific architectures emerged, particularly from industrial practice. Examples are data warehouse architectures [60] for business analytics and, more recently, microservice architectures [46] for Internet services. Exemplary, we will take a closer look at recent microservice architectures in Sect. 3.1. Many present architecture approaches focus on quality requirements, which are discussed in Sect. 3.2.

3.1 Example: Microservice Architectures

Microservices [41, 46] are an architectural style for software which currently receives a lot of attention in both industry and academia. Especially so-called Internet-scale systems use them to satisfy their enormous scalability requirements and to rapidly deliver new features to their users. As the name implies, services are the building blocks and main means of modularization in microservice architectures. Services run in separate process contexts and can be individually deployed, replaced, and retired. The services are built around business capabilities by cross-functional teams, which are responsible for every aspect of the service from development to productive operation.

Traditionally, information system integration aims at achieving high data coherence among heterogeneous information sources [28, 30]. However, a great challenge with integrated databases is the inherently limited horizontal scalability of transactional database management [1]. One of the intentions of microservice architectures is to overcome the limited scalability of such monolithic architectures [32]. A system has a microservice architecture when that system is composed of many collaborating microservices, typically without centralized control [46]. Microservices are built around business capabilities and take a full-stack implementation of software for that business area, including individual data stores.

Microservice architectures provide small services that may be deployed and scaled independently of each other and may employ different middleware stacks for their implementation. Microservice architectures intend to overcome the shortcomings of monolithic architectures where all of the application's logic and data are managed in one deployable unit.

A vertical decomposition into self-contained systems (scs-architecture.org) along business services is recommended. Besides scalability, an appropriate modular structure supports program comprehension, resilience (inhibiting error propagation), and autonomous teams with good knowledge of their vertical domain. Microservice architectures facilitate scalability [32], as well as agility and reliability [33].

An example vertical decomposition of an e-commerce system into self-contained services is illustrated in Fig. 3. A vertical microservice is a part of the platform that is responsible for a single bounded context in a business domain [20]. Verticals could be as small as a microservice, but most of the time, they are more coarse grained. The trade-off between many small components and a few large components must be considered in service and system design [29].

Microservices should follow the "Shared Nothing" principle: They do not share state, no infrastructure component, no database, or other shared resources. The big advantages of shared-nothing architectures are horizontal scalability and improved fault tolerance. The reason for this is apparent: If two components are not sharing anything, they are obviously unable to have a negative impact on each other. Small services are easier to deploy and, since they are autonomous, are less likely to cause system failures when they go wrong.

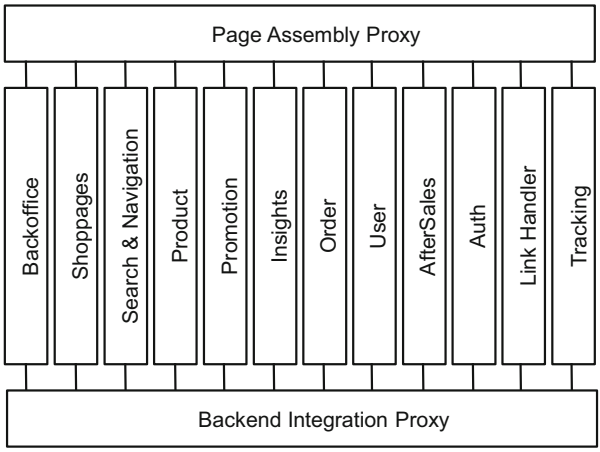

Fig. 3 Example vertical decomposition of an e-commerce system into self-contained microservices [33]

The well-known Conway's law states that organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations [17]. If the organizational structure is decomposed vertically and according to the microservices structure into cross-functional feature teams, scaling development capacities according to changing business requirements is enabled. The feature teams should be highly independent, having members of all roles and skills that are required to build and maintain their microservices. Microservices reinforce modular structure, which is particularly important for larger teams. Decoupling teams is as relevant as decoupling software modules.

Being a highly distributed architecture, microservices are particularly susceptible to partial failures. Therefore, microservices must be designed to cope gracefully with the unavailability of required services to prevent cascading failures. Several patterns have emerged for this purpose [47], such as the circuit-breaker pattern. In this pattern, dependencies such as other services are wrapped in a so-called circuit-breaker object, which serves as a proxy for the dependency and monitors its availability. If the dependency becomes unavailable or unresponsive, the circuit breaker trips and takes appropriate measures, such as immediately returning cached data or default values. After some time, the circuit breaker may check whether the dependency is available again, and if so, return to proxy mode.

Decentralizing responsibility for data across microservices has implications for managing updates. The traditional approach to dealing with updates is to use transactions to guarantee consistency when updating multiple resources. This approach is often used within monoliths. Using transactions this way helps with consistency, but imposes significant coupling, which is problematic across distributed services. Distributed transactions are notoriously difficult to implement, and as a consequence, microservice architectures emphasize transaction-less coordination

between services, with explicit recognition that consistency may only be eventual consistency and problems are dealt with by compensating operations.

However, be aware that microservice architectures also come with costs. Maintaining consistency, monitoring, alarming, and fault tolerance are difficult for a distributed system, which means that you have to operate a much more complex system than in monolithic architectures. You need a mature operations team to manage lots of services, which are being redeployed frequently.

3.2 Focus on Quality Requirements

Quality requirements are the most important requirements for architectural work. The study of software architecture recognizes the dependency between an architecture and a software system's quality attributes such as performance, modifiability, and security. Architectural analysis and evaluation emerged connecting quality attributes and architectural design decisions [16]. For the analysis of quality attributes, software architecture models are used to analyze whether the system can meet its nonfunctional requirements [18]. The goals of architectural evaluations are estimations about the effects of architectural decisions, concerning quality attributes of software. Scenario-based software architecture evaluation methods, for instance, usually assess maintainability [4]. Architectural evaluation for software design helps to understand the consequences of design decisions, enables substantiated design decisions, helps to identify trade-offs, helps to check compliance, and supports managing risks.

Quality can be addressed by a model-based or a measurement-based approach. Model-based quality analysis provides information at an early stage, that is, before the system is implemented [51]. This approach may identify problems early and may reduce rework costs. However, developing the appropriate models requires additional effort and time. Measurement-based approaches to quality perform real measurements [62]. However, this is only applicable when the system is implemented, but can provide real-life data.

Nonfunctional attributes, such as scalability and fault tolerance for high availability, are addressed by microservice architectures (Sect. 3.1). A consequence of using microservices is that applications need to be designed such that they can tolerate the failure of individual services. Since services can fail at any time, it is important to be able to detect the failures quickly and, if possible, automatically restore services. Thus, microservice applications put a lot of emphasis on real-time monitoring of the application, checking both technical metrics (e.g., how many requests per second is the database getting) and business-relevant metrics (such as how many orders per minute are received). Monitoring [62] can provide an early warning system of something going wrong that triggers development teams to follow up.

Quality does not just happen. It needs to be thought about and carefully considered. If you do not pay attention to system qualities, they can be hard to achieve at the last moment.

4 Future: Proper Integration of Architecture Work into Agile Software Development

The tension between the agile and architecture communities still is fairly high. Ford is often cited for his statements "Architecture is the stuff that's hard to change later" and "By deferring important architectural and design decisions until the last responsible moment, you can prevent unnecessary complexity from undermining your software projects" [23]. However, software architecture should be recognized as a key foundation to agile software development, despite the fact that it is often ignored by the agile community who has nicknamed it BUFD (big up-front design).

Meanwhile, the agile community observes a renaissance with innovation in software architecture [59]: Organizations have accepted that "cloud" is the de facto platform of the future, and the benefits and flexibility it brings have ushered in a "renaissance" in software architecture. The disposable infrastructure of cloud has enabled the first "cloud native" architecture, namely, microservices. Continuous delivery, a technique that is radically changing how tech-based businesses evolve, amplifies the impact of cloud as an architecture. ThoughtWorks [59] expect architectural innovation to continue, with trends such as containerization and software-defined networking providing even more technical options and capabilities.

As observed by Keeling [37], promoting business agility requires sound architecture design. The question is how best to achieve agility through architecture. Highly modular architectures that allow for rapid experimentation are critical to a successful integration of architecture work into agile software development. Lightweight, agile methods are promising because they enable teams to learn fast, fail fast, change fast, and communicate effectively. These factors are essential for self-organizing teams, which in turn enable better designs to emerge [37].

Section 4.1 introduces the envisioned role of architecture owner in agile teams, before the relationship between software development and operations is discussed in Sect. 4.2. Achieving reliability with agile development, runtime adaptivity, and keeping architecture knowledge up to date for long-living software systems are discussed in Sects. 4.3–4.5.

4.1 Integrating Architecture Owners into Agile Teams

Treating architecture as a phase ignores its foundational role in software development. Architecture work should be integrated with all software development activities. With agile development, you may have an "Architecture Owner" role, who should involve the entire team to make informed and accepted architectural decisions. The big question is, how many architectural and design decisions upfront (before Sprint 1)? A promising approach is to create an architecture *vision* in Sprint "zero." You need to accept constraints, identify and promote desired quality attributes, assign functional responsibilities to elements, and guide the design with patterns. Try to establish a clear architecture vision.

With agile software development, you can design the architecture through stories and use cases, so driven by requirements. Use case scenarios describe specific interactions between the user of an application system and the system itself. We should test the architecture against the scenarios that are associated with the quality attribute requirements for the system, such as performance. You can add backlog items for technical debt and quality-related architecture work. Quality-related acceptance criteria can be attached to user stories. You should monitor system qualities, for instance, via an operational dashboard. Do not worry about getting your architecture right on the first day. Model and implement incrementally. Prioritize the architecture features and mitigate the key risks.

Architecture validations encourage communication among project stakeholders. Continuously validating the architecture tells us whether we still have the right team structure. Fixing defects early is the best approach. Communication and collaboration aspects of architecture are just as important as developing it. Architecture evaluations help significantly with collaboration and communication. The architecture owner should provide architecture leadership in a collaborative manner and help the team members enhance their capabilities in understanding architectural principles and trade-offs involved. The architecture owner should be integrated with day-to-day development and pay attention to details.

Some decisions are too important to leave until the *last* responsible moment, so choose the *most* responsible moment. Choose an architecture, before it chooses you, as may otherwise happen with *emergent* architectures.

4.2 Integrating Software Development and Operations: DevOps

The DevOps movement continues what agile started. The DevOps movement intends to improve communication, collaboration, and integration between software developers (Dev) and IT operations professionals (Ops). Automation is key to DevOps success: automated building of systems out of version management repositories; automated execution of unit tests, integration and system tests; automated deployment in test and production environments; including performance benchmarks [64]. DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality [6, 12].

The deployment pipeline is the place where the architectural aspects and the process aspects of DevOps intersect [6]. Architectural choices need to facilitate continuous delivery. Microservice architecture is designed for minimizing coordination needs and allowing independent deployment. Multiple simultaneous versions may be managed with feature toggles and backward/forward compatibility. Feature toggles support rollback, canary testing, and A/B testing. Microservices leverage continuous integration [43] and continuous deployment [52] to promote DevOps. Microservice architectures and DevOps benefit from each other [5].

4.3 Achieving Reliability with Agile Software Development

For large software systems, a major difficulty for automated regression testing is caused by the high computational costs of tests. To ensure high code coverage, a potentially exponential set of test configurations must be executed. A solution to this challenge could be a proper modularization of the software such that the software components become testable in isolation. Modularization approaches such as microservices may also facilitate automated regression testing of large software systems. Tooling and automation is needed to satisfy quality assurance needs. The combination of modular microservice architectures with automated quality assurance allows to retain reliability with agile software development [33].

Many agile teams only focus on functional testing, but there is a lot more to test, for instance, performance. Performance tests may be automated in continuous integration setting via regression benchmarking [64]. A strong model of architecture-based testing, backed by formal reasoning and appropriate tooling, could have a major economic impact on software system development.

4.4 Using Architecture Models for Runtime Adaptability

Scalable systems should allow to react to changing workloads automatically via elastic capacity management [63], as offered by cloud infrastructures [35]. With microservice architectures, you can dynamically replicate those microservices to cloud infrastructures that are under heavy load. It is not necessary to scale the complete system, as it would be required with a monolithic system.

Fault tolerance is intended to ensure the delivery of the correct services in the presence of active faults. It is implemented by error detection and subsequent system recovery. Error detection finds an erroneous system state. The following system recovery transforms the system state that contains one or more errors into a state without detected errors. A possible solution is given by dynamic adaptation. In the case of errors, dynamic adaptation can ensure that the best possible system functionality is achieved and that critical functions are kept alive (survivability). Realizing survivability instead of fault tolerance provides an immense potential for saving costs, for ensuring the safety of the system, and for achieving acceptable availability. The abovementioned circuit-breaker pattern provides such properties.

To enable such runtime adaptability, we need architecture information in the running system: architecture description as an executable deliverable, also called models@runtime [7].

4.5 Keeping Architecture Knowledge up to Date for Long-Living Software Systems

The highest costs in software development are generally in system maintenance and the addition of new features. If done early on, architectural evaluation can reduce that cost by revealing a design's implications [16]. This, in turn, can lead to an early detection of errors and to the most predictable and cost-effective modifications to the system over its life cycle. Software architecture captures and preserves designers' intentions about system structure and behavior, thereby providing a defense against design decay as a system ages [38]. The quality and longevity of a software-reliant system is largely determined by its architecture. Technical debt [39] should be avoided. Bad architectural decisions are a major contributor to technical debt.

Architecture knowledge is often lost as we move to code. The best architecture is worthless if the code does not follow it [15]. For such long-living software, it is important to keep the documentation and knowledge about the software up to date [27]. Without conformance between architecture documentation and code, the architecture documentation becomes irrelevant. We might establish conformance by construction—via model-driven software engineering—or by reverse engineering (analyzing an artifact statically or dynamically to determine its architecture). Software system comprehension with reverse engineered architecture models is helpful in this context [21, 22].

Successful, large systems have a long lifetime. They must evolve to meet changing requirements. Existing systems which must be maintained are sometimes called legacy systems. Modernization of legacy software systems is required [34, 61]. When a software system evolves, ideally its prescriptive architecture is modified first. In practice, however, the system—and thus its descriptive architecture—is often directly modified. This happens because of perception of short deadlines, need for code optimizations, inadequate tool support, etc.

Understanding the relationship between architectural decisions and a system's quality attributes reveals software architecture evaluation as a useful risk-reduction strategy. Decisions are the main deliverable of (agile) architecture work, while keeping a backlog of architectural concerns. Managing cost and risks is the primary business goal and prioritizing factors for architecture owners. For longevity, decide at the most responsible moment, not the last possible moment.

5 Summary

Software architectures are essential to develop and maintain large-scale, long-living software systems. The understanding of these systems is improved by a high-level abstract view on a system. Architecture supports the reuse of components and frameworks. Architecture makes expected evolution explicit and separates functionality and connection mechanisms of components and services, so that

they can evolve individually. Full automation of quality assurance and software deployment allows for early fault and error detection, thus reducing repair times both during development and during operations.

Finding the right balance for architecture work is the art of agile architecture ownership. We can expect a coalescence of architecture work and agile software development practices. Architecture owners should decide at the *most responsible* moment, not the *last possible* moment.

References

1. Abbott, M., Fisher, M.: The Art of Scalability, 2nd edn. Addison-Wesley, Reading (2015)
2. Abowd, G., Allen, R., Garlan, D.: Formalizing style to understand descriptions of software architecture. ACM Trans. Softw. Eng. Methodol. **4**(4), 319–364 (1995)
3. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. Softw. Eng. Methodol. **6**(3), 213–249 (1997)
4. Babar, M., Gorton, I.: Comparison of scenario-based software architecture evaluation methods. In: Proceedings of the 11th Asia-Pacific Software Engineering Conference, pp. 600–607 (2004)
5. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables DevOps: migration to a cloud-native architecture. IEEE Softw. **33**(3), 42–52 (2016)
6. Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. Addison-Wesley, Reading (2015)
7. Bencomo, N., France, R., Cheng, B.H.C., Aßmann, U. (eds.): Models@run.time. Lecture Notes in Computer Science, vol. 8378. Springer, Cham (2014)
8. Bennett, K.H., Rajlich, V.T.: Software maintenance and evolution: a roadmap. In: Proceedings of the Conference on The Future of Software Engineering, ICSE '00, pp. 73–87. ACM, New York (2000)
9. Bernardo, M., Inverardi, P. (eds.): Formal Methods for Software Architectures. Lecture Notes in Computer Science, vol. 2804. Springer, Berlin (2003)
10. Bertolino, A., Corradini, F., Inverardi, P., Muccini, H.: Deriving test plans from architectural descriptions. In: Proceedings of the 22th International Conference on Software Engineering, pp. 220–229. IEEE Computer Society Press, Limerick (2000)
11. Bosch, J.: Design & Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley, Harlow (2000)
12. Brunnert, A., van Hoorn, A., Willnecker, F., Danciu, A., Hasselbring, W., Heger, C., Herbst, N., Jamshidi, P., Jung, R., von Kistowski, J., Koziolek, A., Kroß, J., Spinner, S., Vögele, C., Walter, J., Wert, A.: Performance-oriented devOps: a research agenda. Technical report, Standard Performance Evaluation Corporation (SPEC) (2015)
13. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-Oriented Software Architecture: On Patterns and Pattern Languages, vol. 5. Wiley, Chichester (2007)
14. Christensen, H.B., Hansen, K.M.: An empirical investigation of architectural prototyping. J. Softw. Syst. **83**(1), 133–142 (2010)
15. Clements, P., Shaw, M.: The golden age of software architecture revisited. IEEE Softw. **26**(4), 70–72 (2009). https://doi.org/10.1109/MS.2009.83
16. Clements, P., Kazman, R., Klein, M.: Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley, Reading (2001)
17. Conway, M.E.: How do committees invent? Datamation **14**(4), 28–31 (1968)
18. Crnkovic, I., Larsson, M., Preiss, O.: Concerning predictability in dependable component-based systems: classification of quality attributes. In: Architecting Dependable Systems II. Lecture Notes in Computer Science, vol. 3549. Springer, Berlin (2005)

19. Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R.: Structured Programming. Academic, London (1972)
20. Evans, E.: Domain-Driven Design. Addison-Wesley, Reading (2004)
21. Fittkau, F., Roth, S., Hasselbring, W.: ExplorViz: visual runtime behavior analysis of enterprise application landscapes. In: 23rd European Conference on Information Systems (ECIS 2015 Completed Research Papers), pp. 1–13. AIS Electronic Library (2015)
22. Fittkau, F., Krause, A., Hasselbring, W.: Software landscape and application visualization for system comprehension with ExplorViz. Inf. Softw. Technol. **87**, 259–277 (2017)
23. Ford, N.: Evolutionary architecture and emergent design: environmental considerations for design, part 2 (2010). https://www.ibm.com/developerworks/java/library/j-eaed18/index.html
24. Frakes, W.B., Kang, K.: Software reuse research: status and future. IEEE Trans. Softw. Eng. **31**(7), 529–536 (2005)
25. Galster, M., Weyns, D., Tofan, D., Michalik, B., Avgeriou, P.: Variability in software systems – a systematic literature review. IEEE Trans. Softw. Eng. **40**(3), 282–306 (2014)
26. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison Wesley, Reading (1995)
27. Goltz, U., Reussner, R., Goedicke, M., Hasselbring, W., Märtin, L., Vogel-Heuser, B.: Design for future: managed software evolution. Comput. Sci. Res. Dev. **30**(3), 321–331 (2015)
28. Hasselbring, W.: Information system integration. Commun. ACM **43**(6), 32–36 (2000)
29. Hasselbring, W.: Component-based software engineering. In: Handbook of Software Engineering and Knowledge Engineering, pp. 289–305. World Scientific Publishing, Singapore (2002)
30. Hasselbring, W.: Web data integration for E-commerce applications. IEEE Multimedia **9**(1), 16–25 (2002)
31. Hasselbring, W.: Formalization of federated schema architectural style variability. J. Softw. Eng. Appl. **8**(2), 72–92 (2015)
32. Hasselbring, W.: Microservices for scalability: keynote talk abstract. In: Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE 2016), pp. 133–134. ACM, New York (2016)
33. Hasselbring, W., Steinacker, G.: Microservice architectures for scalability, agility and reliability in e-commerce. In: Proceedings 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pp. 243–246. IEEE, Gothenburg (2017)
34. Hasselbring, W., Reussner, R., Jaekel, H., Schlegelmilch, J., Teschke, T., Krieghoff, S.: The Dublo architecture pattern for smooth migration of business information systems. In: Proceedings of the 26th International Conference on Software Engineering (ICSE 2004), pp. 117–126. IEEE Computer Society Press, Edinburgh (2004)
35. Heinrich, R., Schmieders, E., Jung, R., Rostami, K., Metzger, A., Hasselbring, W., Reussner, R., Pohl, K.: Integrating run-time observations and design component models for cloud system analysis. In: Proceedings of the 9th Workshop on Models@run.time, Workshop Proceedings, vol. 1270, pp. 41–46. CEUR, Aachen (2014)
36. IEEE recommended practice for architectural description of software-intensive systems (2000). (also ISO/IEC DIS 25961 (2006))
37. Keeling, M.: Lightweight and flexible: emerging trends in software architecture from the SATURN conferences. IEEE Softw. **32**(3), 7–11 (2015)
38. Kruchten, P., Obbink, H., Stafford, J.: The past, present, and future of software architecture. IEEE Softw. **23**(2), 22–30 (2006)
39. Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: from metaphor to theory and practice. IEEE Softw. **29**(6), 18–21 (2012)
40. Lange, C.F.J., Chaudron, M.R.V., Muskens, J.: In practice: UML software architecture and design description. IEEE Softw. **23**(2), 40–46 (2006)
41. Lewis, J., Fowler, M.: Microservices (2014). http://martinfowler.com/articles/microservices.html
42. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**(1), 70–93 (2000)
43. Meyer, M.: Continuous integration and its tools. IEEE Softw. **31**(3), 14–16 (2014)

44. Muccini, H., Bertolino, A., Inverardi, P.: Using software architecture for code testing. IEEE Trans. Softw. Eng. **30**(3), 160–171 (2004)
45. Müller, H., Wong, K., Tilley, S.: Dimensions of software architecture for program understanding. In: Proceedings of the International Workshop on Software Architecture (IWSA '95), Dagstuhl (1995)
46. Newman, S.: Building Microservices. O'Reilly, Sebastopol (2015)
47. Nygard, M.T.: Release It! – Design and Deploy Production-Ready Software. The Pragmatic Bookshelf, Raleigh (2007)
48. Pahl, C., Giesecke, S., Hasselbring, W.: Ontology-based modelling of architectural styles. Inf. Softw. Technol. **51**(12), 1739–1749 (2009)
49. Parnas, D.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12), 1053–1058 (1972)
50. Perry, D., Wolf, A.: Foundations for the study of software architecture. ACM SIGSOFT Softw. Eng. Notes **17**(4), 40–52 (1992)
51. Reussner, R.H., Becker, S., Happe, J., Heinrich, R., Koziolek, A., Koziolek, H., Kramer, M., Krogmann, K.: Modeling and Simulating Software Architectures: The Palladio Approach. MIT Press, Cambridge (2016)
52. Rodriguez, P., Haghighatkhah, A., Lwakatare, L.E., Teppola, S., Suomalainen, T., Eskeli, J., Karvonen, T., Kuvaja, P., Verner, J.M., Oivo, M.: Continuous deployment of software intensive products and services: a systematic mapping study. J. Syst. Softw. **123**, 263–291 (2017)
53. Shaw, M.: Architectural issues in software reuse: it's not just the functionality, it's the packaging. Softw. Eng. Notes **20**, 3–6 (1995)
54. Shaw, M., Clements, P.: A field guide to boxology: preliminary classification of architectural styles for software systems. In: Proceedings of the Twenty-First Annual International Computer Software and Applications Conference (COMPSAC 1997), pp. 6–13 (1997)
55. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Upper Saddle River (1996)
56. Shaw, M., DeLine, R., Klein, D., Ross, T., Young, D., Zelesnik, G.: Abstractions for software architecture and tools to support them. IEEE Trans. Softw. Eng. **21**(4), 314–335 (1995)
57. Taylor, R., Tracz, W., Coglianese, L.: Software development using domain-specific software architectures. ACM SIGSOFT Softw. Eng. Notes **20**(5), 27–38 (1995)
58. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. Wiley, Hoboken (2009)
59. ThoughtWorks Inc.: Technology Radar (2015). http://www.thoughtworks.com/radar
60. Vaisman, A., Zimányi, E.: Data Warehouse Systems: Design and Implementation. Springer, Berlin (2014)
61. van Hoorn, A., Frey, S., Goerigk, W., Hasselbring, W., Knoche, H., Köster, S., Krause, H., Porembski, M., Stahl, T., Steinkamp, M., Wittmüss, N.: Dynamod project: dynamic analysis for model-driven software modernization. In: Proceedings of the 1st International Workshop on Model-Driven Software Migration (MDSM 2011), vol. 708, pp. 12–13. CEUR, Aachen (2011)
62. van Hoorn, A., Waller, J., Hasselbring, W.: Kieker: a framework for application performance monitoring and dynamic software analysis. In: Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012), pp. 247–248. ACM, New York (2012)

63. von Massow, R., van Hoorn, A., Hasselbring, W.: Performance simulation of runtime reconfigurable component-based software architectures. In: Crnkovic, I., Gruhn, V., Book, M. (eds.) Software Architecture (Proceedings ECSA 2011). Lecture Notes in Computer Science, vol. 6903, pp. 43–58. Springer, Heidelberg (2011)
64. Waller, J., Ehmke, N.C., Hasselbring, W.: Including performance benchmarks into continuous integration to enable DevOps. SIGSOFT Softw. Eng. Notes **40**(2), 1–4 (2015)