Software Architecture Fundamentals

Whitepaper

Randolf Rotta, rottaran@b-tu.de
Vladimir Nikolov, vladimir.nikolov@uni-ulm.de
Lutz Schubert, lutz.schubert@uni-ulm.de

19th April 2016

Contents

1 Overview

A *software system's architecture* is the set of principal design decisions about the system and lays out the path for the system's construction and evolution. The design decisions consider all aspects of the system including the basic structure, the behavior, the interactions, as well as non-functional properties [GS94, TMD09]. Architectures are a description of elements ("what"), form ("how"), and rationale ("why") [PW92]. Architectures are about the reuse of ideas, patterns, and experience, whereas product families exercise the reuse of components, structure, implementations, and test suites. The idea of software components was first expressed on a conference sponsored by the NATO Science Committee in 1968 [MBNR68].

The composition and interplay of processing, data (information and state), and interaction needs to be described in some way. Software architectures and architecture styles provide vocabularies for this purpose. The next section introduces the basic vocabulary. The next two sections review often used architecture styles and connector types, mostly based on [TMD09].[1] Then, computational models with focus on parallel and distributed systems are discussed. The last section reviews best-practice design guidelines that address non-functional requirements of software architectures.

[1]See also `http://csse.usc.edu/classes/cs578_2013/`

2 Components, Architectural Styles, and Patterns

Software components encapsulate processing functionality and/or data. Components may just manage data without processing, provide processing for external data, or combine both. Access to their functionality and data is restricted through well defined interfaces and usage contracts. All dependencies to the execution environment are explicitly defined. *Software connectors* are architectural building blocks that regulate the interactions between components. Connectors can be much more sophisticated than method calls and shared data access, for example streams, event buses, multicasts, and adaptors. The *system's configuration* is a set of specific associations between components and connectors and the *architectural topology* governs design rules about these associations.

Architectures can exist at quite different levels of abstraction. For example, the World Wide Web is an component-based architecture although no single peace of software implements the whole architecture. The takeaway herein is, that components do not necessarily have to be exactly the objects that are provided by some object-oriented programming language. Neither does such a programming language provide a reasonable architecture – except for very narrow minded languages. More general purpose languages simple defer more design decisions into the software architecture.

An *architectural pattern* is a set of decisions that are applicable to a recurring design problem and can be reused in many different architectures. In contrast, an *architectural style* is a named collection of architectural design decisions and constraints that provide the foundation for architectures. Based on experience and reasoning, they result hopefully in better architectures with respect to functional and non-functional objectives.

An architectural style consists of a vocabulary of design elements, composition rules, and a semantic interpretation. The vocabulary is a set of high-level types that categorize components, connectors, and data elements into, for example, clients, servers, pipes, sensors, actors, and regulators. The composition rules are a collection of allowable structural patterns that provide essential invariants of the style, for example layered versus peer-to-peer composition. The semantic interpretation provides a computational model that describes the behavioral rules of the design elements and translates the allowed compositions into well-defined meanings, for example a processing pipeline or a control circuit.

The main benefits of defining and using a style are design reuse by applying the style's solutions to new problems, code reuse by exploiting the style's invariant aspects, and comprehensibility of the system structure through the style's semantics. By combining the component and connector types, the composition rules and the semantic interpretation, style-specific analyses become possible. This can even extend to style-specific visualizations of the system structure.

A software system has actually two different architectures at any given point in time: The prescriptive (as-intended) architecture and the descriptive (is-implemented) architecture. Due to the human nature of programmers, the implemented architecture can drift away from the intended architecture (without violating prescriptive decisions) or erode by violating prescriptive decisions. Thus ideally, the evolution of software should first adapt the prescriptive architecture and just then change the actual implementation.

Most architecture styles exhibit a mismatch between their elements and the capabilities of the target programming language and operating environment. *Architecture frameworks* bridge this gap by, for example, providing reusable connectors for communication, generators for glue components, and suitable activity schedulers. They provide the infrastructure "middleware" that helps to implement architectures. As such, frameworks are usually not directly present in the application's architecture. For example, a comparison of different frameworks for the C2 style can be found in [MMMR02].

3 Existing Architectural Styles

Two traditional language influenced styles are the *main program with subroutines* and the *object-oriented style*. In the latter, the system is decomposed into objects as main component type. Method calls via object references are used as connectors. The objects are responsible for the consistency of their internal state and hide details of the state to the outside. In practice this style is too general, leaving a lot room for incompatible solutions, for example how the objects become connected to other objects and which side effects are allowed for method invocations.

Component-oriented systems without any language-level objects exist. For example, the components of aviation control systems are usually physical boxes with literally hard-wired connectors between boxes. Still, many software architectures benefit from using objects as building blocks for more sophisticated components and connectors.

The following paragraphs review prevalent architectural styles. The review will begin with basic styles and conclude with heterogeneous styles that are a combination of simpler styles. Further examples of distributed and concurrent architectural styles can be found in [Kha02, SEHT04, Ere06].

Data-Flow styles. The batch sequential style is probably one of the oldest architectures, dating back to the tabulating machines of Hollerith [Hol94]. Each component processes a large batch of data. After completing the batch, the resulting output is passed to the next component. Components may be connected over sneakernet (humans that carry around stacks of punch cards or the like) with considerable bandwidth [Tan89]. The latest revision, known as Google's MapReduce, uses slightly more sophisticated infrastructure.

The pipe and filter style improves upon the high latency of batch sequential architectures. The components are filters that are connected with pipes. The pipes stream the application's data from one component to the next. The filter components are independent as they share no state and have no knowledge about the source and destination of the streamed data, including other filters in the pipeline. This enables the concurrent and interleaved execution of filters and the analysis of throughput and latency.

Client/Server and Layered styles. The components are categorized into clients and servers. The important characteristic of this style is that the servers do not know the identity and number of clients. Only the clients know the servers they use. Both are connected by request-response mechanisms like remote procedure calls.

The layered style extends the client/server style hierarchically. Each layer acts as a server to higher layers and as a client to lower layers. Connections between components of the same layer are often deemed a bad practice. One advantage of this style is, that it is sometimes possible to replace whole layers without changing the interface to above layers. A layer that fully decouples layers above from all layers below is a virtual machine.

Blackboard style. This style has two component types. The central blackboard components store the system state and all other components operate on the blackboard. The control flow is driven by be blackboard. Examples are the Linda tuple spaces and the model components in Model-View-Controller patterns. Shared memory can also serve as blackboard.

Implicit Invocation styles. In these styles, components emit events instead of calling specific methods of other components. The emitter of an event does neither know which components will handle it nor which effects this will have. Listener components register at event sources by providing methods that handle specific events. Because components can assume only little about the effects of their events, they should be quite reusable. Control about the order of execution is moved completely to the system. In practice, it can become quite difficult to reason about cross-component interaction without analyzing the system's configuration at runtime.

A widely used form is the publish/subscribe style. Components register at other components for specific events and published events are broadcasted to all subscribers. Another style is the event bus. All components connected to an event bus can emit any event to it and receive all events on the bus. Receivers may filter events according to own criteria. Thus, components communicate only with event buses directly. TACO's object groups can be interpreted as event buses.

4 Peer-to-peer style

The state and behavior is distributed over several components (peers) and each component can act as client and server to other peers and components. As a decentralized structure, every peer has is own control flow. State data may be replicated or partitioned but such details are hidden to the outside. One example are the service fleets of the factored operating system fos [WA09].

Finally, the following three styles review examples of heterogeneous styles that combine more basic strategies.

C2 style. The C2 style [TMA+96] is based on layers that are connected through implicit invocation mechanisms. Higher layers can be informed about state changes by emitting notification messages and actions can be requested from lower layers by emitting request messages. The components are not connected directly between layers. Instead, all messages are routed through connectors, such as message queues, event buses, or publish/subscribe connectors.

Myx style. The architectural elements of the Myx style[1] are components and connectors. The imported and exported interfaces of each element are classified into a top versus bottom domain and interfaces from the top domain can be linked only to interfaces from the bottom domain. Cyclical links are prohibited. Calls towards a bottom interface are allowed to be synchronously blocking but all other calls have to be asynchronous, that is use implicit invocation. The top to bottom linking constraint enforces a layered architecture where asynchronous communication promotes loose coupling and prevents deadlocks. Still, the one-directional synchronous calls enable good performance for most of the local communication.

Quasar architecture style. The Quasar development methodology, among many other things, provides an example architectural style that categorizes components and interfaces into five types 0, A, T, AT, and R. The *0* type is used for basic reusable components and data types like containers and strings. Type *A* components represent

[1]See http://isr.uci.edu/projects/archstudio/myx.html

application specific code and type *T* represents components that are determined by a technical programming interface, for example a file system or resource management interface. Accordingly, *AT* components mix application-specific and technology-specific code. In order to improve the chances for later reuse, such components should be strongly avoided because they most certainly do not fit in a different application or technology context. Instead, Quasar advocates the introduction of type *R* components. These provide the small glue between A and T components and may sometimes be generated. To connect A and T components without creating a direct AT dependency, standard interfaces (type 0) and adapters from the R category may be used.

The 0,A,T,R-partitioning can be refined into domain specific component types. The motivation is, that general categories are more easy to identify than individual components. Then, the allowed dependencies between these categorical types are defined as an acyclic directed graph, quite similar to layered styles. Experience tells, that changes to components of a category affect the dependent categories and, thus, higher categories are more likely to change more often. And conversely, changes in lower categories get longer deferred and stronger suppressed the more components depend on it.

Service Oriented Architecture style. There is a lot of hype and buzz around SOA, especially in proximity to web services.[2] Hidden beyond all that noise are a couple of useful architectural ideas: The system's components are partitioned into the application and several services. Each service is designed to be autonomous (maintained and developed independently), distributable (can be placed anywhere), and loosely coupled (independent from other services). This style aims at the reuse of services between multiple application while still being able to evolve each service independently.

[2]"The Large Hadron Collider was created to help unlock the secrets of the universe. And also to create a working SOA implementation." http://soafacts.com/

5 Connectors

Connectors are, like components, architectural elements. Unlike components, their purpose is to model application-independent interaction mechanisms instead of providing new application features. They provide the architecture's communication channels and are used to transfer data elements, control flow, or both. While connectors are first-class elements of the architecture with own identity and dedicated specifications, they often have no identity inside software implementations. Instead, they are usually provided by the language or architecture framework and their implementation can be distributed through the whole code.

The explicit modeling of connectors has a couple of benefits. They provide means to abstract interaction patterns, which improves reusability by reducing the direct dependencies between components. As predetermined breaking points, they aid the distribution of components in a distributed system and enable the dynamic replacement of components and whole subsystems. This also simplifies the migration of components in distributed systems. Finally, their clear semantic interpretation can be helpful for system analysis.

The simplest connectors are method calls and access to shared data. As can be seen in the previous section, many architectural styles include more specific and more constrained connectors like, for example, data stream pipes, blackboards, publisher/subscriber mechanisms, and event buses. Taylor et al. [TMD09] provide a categorization of connectors with respect to their main role in the system (Communicators, Coordinators, Converters, and Facilitators) and with respect to basic types (procedure calls, data access, events, streams, linkage, distributor, arbitrator, and adapter).

Communicators separate communication from computation by transporting information. Communicators can be used to abstract over hardware-dependent communication mechanism and to constrain the communication structure, for example in order to enforce security measures. Examples are procedure calls, events, shared data access, and streams.

Coordinators separate control from computation by steering the control flow between components and the order of concurrent interactions. Note that all connectors are coordinators to some degree. Examples are procedure calls, events, and arbitrators.

Converters are used to bridge mismatched interfaces when connecting independently developed components. Examples are wrappers and adapters.

Facilitators improve and mediate the interactions between components that, in contrast to converters, were intended to interact by design. They can be used, for example, for load balancing and for synchronization of critical sections. Examples are linkage, arbitrators, and distributors.

6 Behavior in Parallel/Distributed Systems

Each component has an internal state that is encoded by the component's data, including any data that is shared between components. Interaction between components triggers some activity at the target component with following possible effects: Change of the component's state; Creation of new, possibly temporary, components; And further interactions with other components. The concrete activity depends naturally on the component's state as well as the type of interaction. The actual behavior is constrained by a computational model that defines when and in which order, if any, these effects become visible. It also defines how the behavior of a component, i.e. its activities in dependence to state and interaction, can be described.

The connectors provided by the architecture describe the available types of interaction. Shared data connectors play a special role because they expose parts of a component's internal state and allow to change it without supervision by any affected component. Shared data hands control from the component to the connector. Without care, concurrent activities will result in inconsistent and unpredictable behavior. As a consequence, most behavior models heavily restrict the use of shared data or abandon it completely.

Fork-Join model. Functions serve as components, which have no internal state, usually. The only interaction between functions is by function call, which creates an activation record for execution at some unspecified time and place. Lacking internal state, the behavior of functions depends only on the arguments passed through the call. The behavior is constrained to spawning function calls and, then, wait for the completion of all of them. Thus, only cycles of compute-fork-join-compute are possible, which enables very efficient data management, low memory footprint, and distributed execution by work stealing. However, reacting to events from external sources does not fit into this model or at least prevents many significant optimizations.

Because of the strict parent-child dependencies created by the forks, the control flow can never deadlock.

Examples: Cilk, and less ideological: OpenMP.

Communicating Sequential Processes (CSP). The components are logical threads of execution, called processes. Internally, they might employ any behavior model but traditionally it is very simple single threaded execution. Processes interact by messages. Each process is responsible for receiving and handling its incoming messages explicitly. Some variants address messages to specific target processes, others send messages into and receive from logical communication channels without knowing the other processes connected to the channel. Beyond peer-to-peer message exchange, process groups and multicast channels provide the functionality of bus connectors. One important advantage is that this model does not rely on shared memory and thus results in good performance on distributed memory systems. However, mapping the inevitable shared information onto message exchange is completely up to the user.

Processes can deadlock when waiting mutually for a message before sending the awaited message. One work around are non-blocking communication primitives.

Examples: MPI, go language, Barrelfish OS, LogP model.

Shared Memory Programming. Components can be functions and objects. The functions are executed by several sequential processes similar to the CSP model. But instead of messages, all interactions inside processes are expressed by function/method calls and between processes by direct access to shared data. Calls are executed immediately in the callers control flow. Thus, a component's internal state can be manipulated concurrently. The immanent risk of data inconsistencies lead to elaborate shared data connectors that coordinate concurrent access, for example mutual exclusion, read-copy-update, transactional memory, and special purpose concurrent data structures (associative maps, sets, arrays).

Processes can deadlock when mutual exclusion mechanisms are used. This is circumvented by non-blocking and wait-free shared data connectors.

Examples: PRAM, Linux.

Actor model. The components are called actors. Each actor has an internal state and a message handler. Actors communicate via dynamically typed asynchronous messages and messages to the same actor are handled sequentially in an arbitrary order. In reaction to an message, the actor can create new actors, send messages to other actors, and set the handler that will be used for the next message. The order of these actions is arbitrary and can even be parallel. In the pure form everything is an actor and the handler is the only internal state. Pragmatic implementations support additional state data that is manipulated directly in order to reduce the message count. The message handler parses the message by pattern matching. The current state is usually not analyzed. Instead, the handler for the next message is switched.

The inversion of control in the actor model is notable: Unlike method calls between objects, the caller of a service does just specify what he needs and the target activity specifies when that call will be handled. Messages to different actors can be handled in parallel, for example by a thread pool. Because all messages are sent asynchronously, handlers will never block nor wait and, thus, programs cannot deadlock. However, the message queues of actors can become arbitrarily (but finite) long during the execution of well-defined but appropriately crafted programs. System analysis can eliminate this situation.

The original actor model is actually a Turing-complete formal model of computation. Unlike many other models it does not rely on the composition of sequential processes. For reasons of efficiency and interoperability with existing software, many implementations extend the model more or less.

Examples: Erlang, Actors in Scala, libcppa, Ptolemy II.

The event-reactive programming model REFLEX is quite near to the actor model but has some own twists. Interaction is described by typed events that are emitted to an event sink of matching type, which immediately processes the event. Most sinks just store the event's data by changing their internal state and may then enqueue an activity in a global task queue. The activities may be executed in any order determined by a scheduling strategy. Sinks are combined into a component and represent the component's interface. Their activities share the component's state, but activities are not allowed to block and thus concurrent state access is not possible. To eliminate arbitrarily long messages queues, sinks can drop or merge events and suppress the repeated enqueuing of activities.

Distributed Objects. This is an object-oriented refinement of CSP with objects as components. The objects are assigned to a process and global pointers act as one-directional message channels that may cross processes. The objects interact via, possibly remote, method calls. The execution of local and incoming method calls can be immediate or distributed over a pool of local worker threads, depending on the implementation and configuration.

Without worker threads, programs can easily deadlock. With worker threads, the model inherits concurrent manipulation of component-internal state from the shared memory model with all its challenges. Due to the structural similarities to the actor model, programs can produce arbitrarily long messages queues. This can also manifest in arbitrary many blocked threads that wait for some future result. On a side note, CILK demonized future variables for exactly this reason.

Examples: CORBA, Java RMI, .NET Remoting, X10, TACO

The popularity of this model comes from its similarity to sequential local object-oriented programming and the assumed ease of porting existing object-oriented code into this model. A very good discussion of the inherent problems can be found in [KWWW94]. Four aspects of distributed systems differ significantly from local systems, namely latency, memory access, partial failures, and concurrency. The effects of latency and the semantic differences of remote and local memory access can be hidden through languages and frameworks in exchange for performance. However, partial failures and concurrency issues cannot be hidden. Unified distributed object models know no difference between local and remote objects. In consequence, they either ignore these problems or enforce complicated handling of failures and concurrency even on simple local tasks. Kendal et al. conclude that reliable distributed objects systems cannot be based on the *same kind* of objects all the way down. Instead, there has to be a visible line between local and remote interactions.

7 Non-functional Properties

Non-functional properties are the result of architectural design choices and constrain how easy it will be to implement a system and how well it will work. While most of the design work usually focuses on the functional requirements alone, the non-functional properties will determine the overall system's success in the end. Design guidelines may help to achieve desired non-functional properties. This section reviews guidelines derived from [TMD09].

Examples for non-functional properties are the following quality attributes: modifiability, efficiency, and dependability. *Modifiability* measures the ease of use for the programmer. This will be influenced by the size of interfaces, the number of architectures elements to choose from, the implementation's complexity in number of lines, and the system's support for heterogeneity, adaptability, and extensibility. The analyzability of a system can improve its modifiability by facilitating debugging tasks. The *efficiency* concerns the system's performance in dimensions like, for example, processing time, memory and cache usage, and communication bandwidth utilization. The performance scalability can be studied with respect to the input size, to event frequencies, and to the system size, that is the number of software components or the number of cores and processors. A system is *dependable* when it works as intended without failure. This is a collection of properties like reliability, robustness, fault-tolerance, and safety. The reliability considers only the systems design limits and correct inputs. It can be quantified, for example, by the mean time between failures. In contrast, the robustness considers the behavior with respect to unexpected inputs outside of the system's specification.

- Partition the system's functionality hierarchically along concerns into separate components. Separate data, meta-data, and processing into dedicated components. For example, the Model-View-Presenter pattern moves all state information out of the presenter (as controller) into model components (as data containers). It is a good idea to make processing components independent from data format changes to some degree.

- Make dependencies explicit and avoid unnecessary dependencies. This can be achieved by moving interaction concerns out of components into explicit connectors. Independent interaction concerns should be split into separate connectors. Appropriate connectors can be chosen to increase scalability. Use connectors to control the dependencies between components. Avoid cyclic dependencies by all means.

- Separate components from instantiation, linking, and configuration concerns. Dependencies like references to other components and system-dependent configuration can be provided from the outside by constructor arguments and configuration interfaces. This approach is known as dependency injection principle.

- Keep components and interfaces simple and compact – the KISS principle. Interfaces can often be simplified by providing task-specific interfaces to the same functionality. Components that implement several related interfaces reduce the component count. Consider the possibility to compose connectors.

- Use asynchronous interactions when possible. This inversion of control lets the callee decide when and how to process requests and, hence, reduces control flow dependencies.

- Be careful with broadcasts and distribution transparency with respect to performance. Frequently interacting components should be placed closely together with respect to the number of intermediate components and the distribution over processors.

- Include exception and fault handling concerns in the architecture design.

- Make the system observable: Non-intrusive health monitoring, for example about the utilization of memory and bandwidth resources, can help to identify bottlenecks and resource management mistakes early. Some simple reflection capabilities also can help during debugging, at least can be used to generate meaningful exception reports.

Another collection of guidelines is the SOLID principle[1]: Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion. Most are covered above already. The open-closed principle says that components should be open for extension but closed for modification in such a way that its behavior can be altered without changing its code. The Liskov substitution demands that

[1] `http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod`

subtypes do not tighten the base type's contract, which is usually defined through interfaces. And interface segregation prefers many client-specific interfaces in favor of convoluted general-purpose interfaces.

Finally, Kendal et al. [KWWW94] concludes some guidelines with respect to distributed systems: Accept the fundamental differences between local and remote interactions by classifying interfaces into "local use only" and "remote usable". Only the design of remote interfaces has to consider message frequencies, serialization of message data, and reliably in presence of partial failures. The way interfaces are implemented, either by hand or by IDL compilers, should exploit the knowledge about local-only vs. remote use. While implementing components, the programmer should differentiate explicitly between local and remote interactions. Making the difference visible will help to avoid mistakes.

8 Technical Aspects

This subsection reviews technical aspects that usually come up during design and implementation.

Interfaces. Interfaces can be defined and used at many different places inside the architecture. Therefore a naming convention is helpful: A component takes the *importer* role with respect to an interface if it makes method calls to that interface. A component is an interface's *exporter* if it implements its methods. *Standard interfaces* are defined externally, that is separately from its importers and exporters. In comparison, *submitted interfaces* are defined by the exporter and *requested interfaces* are defined by the importer. In order to connect a requested interface to a submitted interface, obviously an adapter between both is needed.

Errors and Exceptions. *Errors* are an explicit part of the interface definitions. They are expected to occur regularly during normal operation. In comparison, *exceptions* are thrown by implementations when situations outside of the specification are detected. They are expected to occur almost never. One strategy is to introduce security facades in the design that catch such exceptions and trigger diagnosis and repair. Unfortunately, the exception mechanisms of languages like Java and C++ are also used by their standard libraries to signal regular errors. Thus, an architecture should define stricter rules for exceptions and errors.

Resolving cyclic depencencies. A cyclic dependency is a group of two or more components whose direct dependency graph contains a cycle. Such cycles reduce the composability of the system considerably because no single component can be replaced easily without also replacing the other components of the cycle. Depending on the behavior model, such cycles can also lead to deadlocks in the control flow. Finally, cyclic dependencies can make the compilation process much more difficult when virtual methods are replaced by templating mechanisms for higher efficiency.

Two transforms can be applied to break such cycles. It may be possible to arrange the components into a meaningful layered architecture without any two dependent components on the same layer. Then, at least one dependency of the cycle will have to go from a lower to a higher layer. These dependencies should be replaced by implicit invocation connectors like publisher/subscriber or event buses. In the case where two cyclical dependent components cannot be sorted into meaningful layers, the whole interaction between both components should be moved into a coordinator component above both. Then, only the coordinator has a direct dependency to both components.

Inheritance versus Composition. Extension by inheritance is based on the idea that an application derives a specialized class from a class that was provided by the environment. Here, code reuse focuses on inheriting previously defined behavior and extending it with small pieces of application-specific behavior. The interface(s) and connections as seen by the environment cannot be extended. Without generic programming mechanisms like C++ templates, inheritance provides a strict one-to-one relation: The implementation results in a single new class and instances of the base class (i.e. components) can be replaced transparently by instances of the derived class.

Extension by composition focuses on the interaction between components instead of how they are implemented. The system is extended by changing the connections between its components and by inserting further components. Here, code reuse focuses on the reuse of existing components and interfaces, although inheritance may still be used to implement new components.

Composition enables a combinatorial explosion that is only limited by the mismatch between imported and exported interfaces. As such it lies the foundation for highly flexible program families. On a historical note, for example even the huge Enterprise JavaBeans framework moved in its third version radically from extension by inheritance to extension by composition and related strategies. Obviously, a major architectural challenge is to organize the components and interfaces in a way that does not prevent recombination because of too strong dependencies.

Component Containers. Extendable architectures that are based on the composition of components and that apply the dependency injection principle, end up with one or several configuration components. These encode specific deployments of the system by describing which components are instantiated and how they are created, composed and configured.

This results in very simple repetitious code that can be generated to a high degree. *Component containers* are off-the-shelf components for this task. They use a simple standardized description language that is more compact and easier to read than manually written code. From this description, static initialization code can be generated or it is interpreted dynamically. Also, helpful visualizations of the system structure can be generated from this description.

Most container implementations provide a life-cycle management for their components. Advanced component containers can include constraint solvers in order to automatically figure out how to fulfill open dependencies and configure requested components. Dynamic containers may also generate glue components or compile specialized components on demand. This can be used, for example, to replace virtual method interfaces by concrete type information at load time in order to reduce run-time overhead.

Examples are the PocoCapsule/C++ container for embedded systems[1]. Not so obvious containers are the Linux depmod module loading mechanism and the loader for dynamic shared object libraries.

[1] See, for example `http://www.pocomatic.com/docs/whitepapers/pococapsule-cpp/`

Bibliography

[Ere06] Justin R Erenkrantz. Architectural styles of extensible rest-based applications. In *Institute for Software Research, Report UCI-ISR-06-12*. Citeseer, 2006.

[GS94] David Garlan and Mary Shaw. An introduction to software architecture. Technical report, Pittsburgh, PA, USA, 1994.

[Hol94] Herman Hollerith. The electrical tabulating machine. *Journal of the Royal Statistical Society*, 57(4):678–689, 1894.

[Kha02] Rohit Khare. Decentralized software architecture. Technical report, DTIC Document, 2002.

[KWWW94] Samuel C Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. A note on distributed computing. 1994.

[MBNR68] M Douglas McIlroy, JM Buxton, Peter Naur, and Brian Randell. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98. sn, 1968.

[MMMR02] Nenad Medvidovic, Nikunj Mehta, and Marija Mikic-Rakic. A family of software architecture implementation frameworks. In Jan Bosch, Morven Gentleman, Christine Hofmeister, and Juha Kuusela, editors, *Software Architecture*, volume 97 of *IFIP — The International Federation for Information Processing*, pages 221–235. Springer US, 2002.

[PW92] Dewayne E Perry and Alexander L Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[SEHT04] Girish Suryanarayana, Justin R Erenkrantz, Scott A Hendrickson, and Richard N Taylor. Pace: An architectural style for trust management in decentralized applications. In *Software Architecture, 2004. WICSA*

2004. Proceedings. Fourth Working IEEE/IFIP Conference on, pages 221–230. IEEE, 2004.

[Tan89] Andrew S Tanenbaum. *Computer networks*, volume 4. Prentice-Hall Englewood Cliffs (NY), 1989.

[TMA$^+$96] Richard N Taylor, Nenad Medvidovic, Kenneth M Anderson, E James Whitehead Jr, Jason E Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. A component-and message-based architectural style for gui software. *Software Engineering, IEEE Transactions on*, 22(6):390–406, 1996.

[TMD09] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[WA09] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, April 2009.