

A Survey on Software Architecture Domain

Prepared by

Kamran Sartipi

Software Engineering Group

February, 1997

Dept. of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

Contents

1	Introduction	1
2	Software Architecture	2
2.1	Terminology	4
2.2	Software Architecture Research Areas	6
2.2.1	Foundation of Software Architecture	6
2.2.2	Architecture Description Languages	8
2.2.3	Architectural Design Process	10
2.2.4	Architectural Analysis and Evaluation	11
2.2.5	Architectural Reuse (Styles)	13
2.2.6	Architecture Recovery	13
2.2.7	Architecture Development Environment	15

1 Introduction

Computer systems have been around for more than 50 years. They are supposed to make the life easier for human by solving scientific problems, controlling production processes, facilitating user interaction and control of complex equipment, providing entertainment, and assisting in the development of more sophisticated computer systems which closes this loop. The old computer systems in 60's and 70's, despite their inefficiency and unreliability, seemed to be less problematic than their (hundreds times) more efficient and reliable offspring. Why? The level of automation they offered was limited and for the most part, a human made the final decisions.

Now, we can see the indication of automation in every aspect of our life. As automated processes and controlled equipment become more sophisticated, the size and complexity of software increasingly grows (it has nearly doubled every decade [33]), and takes a more critical role. Today's most challenging software projects exceed millions of lines of code, and require the highest degree of reliability, efficiency, and maintainability. The software is often geographically dispersed and has real-time constraints. Examples of such software include: *flight simulators* for Air Force fighters [19], *naval defense systems* [38], *air-traffic control, telecommunication systems*, etc. The software in the automatic baggage-handling system at Denver's International Airport controls 4000 independent telecars, 5000 electric-eyes, 400 radio receivers, 56 bar-code scanners, and connects a network of 100 computers [33]. The project costs \$193 million, and because of bugs in software, the project was not completed as scheduled, causing a nine month delay on grand opening of the airport. The delay caused an estimated \$1.1 million a day in interest and operating costs. Many examples of large human and financial losses due to software failures exist [33].

The following statistics [33] indicate that *Software Engineering*, despite its half-century age, is not yet a mature engineering discipline: for every eight large software projects, two of them are cancelled; most software projects exceed their time-schedules by about 50 per cent; approximately 75 per cent of large systems malfunction after completion, or they are not used at all. Other studies [63] show that the detecting and correcting of errors in software systems takes more than 50 per cent of the development time and effort; and that the maintenance cost of a system is 60 per cent of its entire cost. The above problems are the result of the engineers' inability to overcome and manage the size and complexity of software. This situation, known as *Software Crisis*, is getting more serious as the software industry builds larger systems to fulfill the requirements of modern technology.

Software engineering, with its collection of techniques, tools, and processes, has tried to master this crisis for about 30 years. We have learned how to plan, design, implement, test, and maintain our systems. There is no doubt that applying the software engineering directions and tools yield a higher quality software system. But the above evidence indicates that this discipline has not been successful in dealing with large and complex projects. Thus, software engineering, in its state of the art and technology, is unable to solve all of the problems of today's software systems. The failure of large projects is caused partly because humans have limited abilities to master and comprehend all of the details of a project.

One solution to this problem is to increase the level of abstraction and let the designer

and developer worry about gross-grained entities of a system such as software components (e.g., subsystems with non-trivial functionalities) and their interactions. This view of a system's software, that of the *Software Architecture*, assists us in understanding, managing, and analyzing large and complex systems.

2 Software Architecture

As we discussed earlier, software engineering is not mature enough to deal with the existing chaos in developing large and complex systems. The reasons include:

- The human's limited abilities to understand all of the details of a large system.
- The variety and non-standard software used in the construction of systems.
- The *legacy* of systems, that is, the problem of maintaining, evolving, and integrating old systems which lack adequate design documentation.
- The shortcomings of the applied development process.

Researchers are seeking a higher abstract view of software systems, as a basis for managing the inherent complexity of large systems, in order to better understand, develop, and analyze software systems. In an attempt to solve the above problems, software engineers have employed software architecture to provide higher levels of design abstraction, reusable architectural styles, architectural recovery, and new methods of development process [17], respectively.

Software engineers use the term *architecture* to emphasize the similarities between the structure of a software system and that of a building [52]. A building architect works with different plans of a building (e.g., one set for the customer with exterior views, rooms, and stairs; and one set for the builder with details of plumbing, electrical wiring, heating, and air conditioning). A software architecture is also presented with different views for different stakeholders. The notion of style in building construction conveys the insight of how the building will look, its uses (e.g., a house, business center, or restaurant), its composite materials, and the constraints that are imposed on the shape of the building by the style. The same prescriptions are exactly applicable to the concept of style in software architecture. Choosing proper materials is of great importance in creating a particular building style. The same correspondence is also essential in software architecture style, with the resemblance of materials to software elements.

Being a young research area, software architecture still lacks a universally accepted definition. A few researches believe that this uncertainty is due to the variety of software architectures, rather than being a difficult concept to understand. Roughly speaking, software architecture is the gross allocation of software's functionalities to structural elements, and the definition of the elements' relationships. A few researches have attempted to give a precise definition of software architecture. Some of these definitions are presented below:

- “Software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.” A definition by Mary Shaw and David Garlan [60].
- “The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.” An academic definition, developed in a discussion about software architecture among a team of engineers at the SEI, Carnegie Melon University in 1994 [31].
- “A collection of software and system components, connections, and constraints; a collection of system stake-holders’ needs statements; and a rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stake-holders’ needs statements.” This definition is used in Nortel’s Software Engineering Analysis Lab. It is adapted from [27], focusing on the evaluation properties of a software architecture and the role of stakeholders.
- “Architecture is a formation or construction of components for building. From an information technology perspective, an architecture provides a structure from which functional and technical components can be effectively defined and related to meet business objectives. It addresses business functions, data, application information, hardware, system software to provide a unified and coherent structure of information system components and interfaces. We use the term ‘architecture’ to cover all the functional and technical elements which must be assembled to make the system work, and the relationship between them.” This definition has a practitioner flavor and is used by the Center for Strategic Technology Research, Anderson Consulting [51].

The above four definitions, addressing different applications of the architectural view of software systems, have been selected from nine different definitions that I have collected from the literature. Other definitions have been collected by researchers of Carnegie Melon University [15]. Despite the different objectives among these definitions, they have a common core definition: software architecture defines a software system’s structural components and connectors, and the form of connecting the components through connectors. Software architecture has many aspects to be elaborated on. This is why the software architects collectively reject the term “*the*” architecture of a software system. They believe that a system has several architectural views (see section 2.2.1).

Software architecture domain consists of several research areas which are actively pursued. Typical problems in this domain include:

- *How to capture architectural descriptions of a system?* (addressed by different architectural design processes)
- *What architectural information should be captured?* (the subject of research on architecture description languages)
- *How to evaluate the functional and extra-functional qualities of a large system?* (addressed by the architectural analysis methods)

- *How to keep different architectural representations consistent as the system evolves?* (e.g., traceability between source code and software architecture)

In the following sections, important concepts in software architecture are defined. This is followed by a presentation of different research areas in this domain.

2.1 Terminology

In this section, software architecture and its basic elements (components and connectors) and styles are defined. In the last section, four definitions of software architecture were presented to address the lack of a general definition. I believe that the second definition is compressive enough to serve as an abstract definition, hence, I repeat it below:

“Software architecture: the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.” [31]

The following definitions of architectural components and connectors have been adapted from the book of Shaw and Garlan [60]:

Components represent the encapsulation of system’s computation corresponding to the compilation units of the programming languages. A component’s interface defines the signature and functionality of the services provided by the component. Examples of components include: filter, process, layer, client, server, memory, etc.

Connectors represent the encapsulation of interactions among components. The Connector’s protocol defines the form of communication, the type and order of the transported data, and the way of assuring the protocol execution. Connectors can be viewed as definers of the roles that must be played by some services inside the components. Examples of connectors include: RPC, event broadcast, pipe, etc. A connector can be implemented in a variety of ways such as: built-in mechanism in programming languages; system calls in operating systems; calls to library code; etc.

The above components and connectors are primitive. A composite component comprises of a set of well-defined primitive components and their mediator connectors, defined in an *architecture description language* (ADL). Composite connectors have not been defined yet [60].

Architectural styles

As with software architecture, architectural style suffers from the lack of a unique definition. Some definitions are presented below:

- *“An architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined”.* A definition by Garlan and Shaw [32].

- “*By architectural style, we mean a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done*”. Developed in an attempt to classify different architectural styles [59].
- “*An architectural style describes the structure and common properties of a family of systems rather than of an individual system. A style provides a vocabulary for describing components and a set of (possibly parameterized) connectors that can be used to compose the components into configurations (which are instances of the architectural style). In addition to the introduction of vocabulary, an architectural style may constrain how configurations are assembled*” [20]. Developed in a DoD project to perform military mission simulations among distributed members of a federation [2].
- “*Architectural styles capture broad properties, vocabulary, and configuration constraints that apply to all instances of a family of systems*”. Defined in an activity to develop a customizable architectural design environment [46].

As a summary of the above definitions, a *style* can be defined as a class of systems with a restricted set of elements and a constrained topology. Examples of well known architectural styles include *pipe and filter*, *pipeline* and *batch sequential* (simplified forms of pipe and filter), *client/server*, *implicit invocation*, *layered*, *blackboard*, *object-oriented*, *interpreter*, *state transition*, and *main program/procedure* [60]. These architectural styles are briefly described below.

Pipe and filter style is a combination of filters which are connected through pipes. Filters incrementally transform a stream of data received from a pipe, and deliver it to the next filter via another pipe. In an ideal situation, this style allows concurrency among filters with a distributed control mechanism. However, other characteristics of the pipes and filters may restrict this concurrency. *Pipeline* and *batch sequential* are restricted forms of pipe and filter. The former has a linear topology, and the latter disallows any concurrency among filters. Examples of pipe and filter style include UNIX shell, ancient compiler technology, and signal processing.

Client/Server is a common architectural style in distributed systems, in which a number of clients, using the remote procedure call mechanism, request service from a server. In most cases, the clients acquire the identity and the types of services of a server from a name-server [22].

Implicit invocation allows components to register events for the services they provide. A manager receives request-events from the components which need specific services, and propagates the request-events to the components that have registered those services. This style supports reusability and system evolution, but suffers from poor predictability and testability. Examples include the Field system [56] and HP SoftBench [24].

Layered systems provide hierarchical layers of service, in which a layer uses/provides service from/to its adjacent lower/higher layer. To compensate for the performance inefficiency, bridging over layers is done in some systems. This style supports reuse, abstraction of the system functionality, incremental enhancement of the design, and locality of changes. The style is ideal for systems whose structures are potentially hierarchical such as the ISO/OSI reference model [7].

Data repository style is divided into *database* and *blackboard* systems, depending on the control mechanism used by the repository (knowledge base) to trigger the processes which share and modify the repository. The repository triggers the processes based on the type of transaction (database), or the current state of the repository (blackboard). Examples of blackboard style include modern compiler technology and speech/pattern recognition systems [50],

Object-oriented style encapsulates an object's state and the implementation of operations on that state. It provides a mechanism for isolating object usage from its implementation, and inheritance capabilities, which support hierarchical design. Aesop, an architectural construction environment, is an example of this style [60].

Other styles include: **Interpreter** style, which provides an illusion of the underlying machine (virtual machine) to the user (example: programming languages). **Main program/subroutine** style, in which the main program (driver) dispatches the sole thread of control to subroutines, in a particular sequence. **State transition** style, in which the system transitions to different states according to some stimuli (example: reactive systems [34]).

Domain specific software architectures (DSSA) refer to a number of families of industrial systems that have reached a level of maturity to possess specific software architectures (known as *reference architectures*) for those domains. Examples of DSSAs are found in: avionics [8], command and control (C2) [9], and vehicle management systems [10]

2.2 Software Architecture Research Areas

The following areas are actively pursued in software architecture domain.

2.2.1 Foundation of Software Architecture

Software architecture foundation addresses the fundamental definitions and concepts which assist us in exploring and understanding the software architecture domain.

Architectural views

All knowledge of an architecture can not be captured by a single architectural representation. A *view* of an architecture, first introduced by Zachman [65], embodies a certain type of information concerning a different stakeholder. Each view reveals a certain type of detail of

an architecture which helps the developer or user to understand, describe, use, or implement the system. A number of developers of large software systems have reported the significance of views in capturing, communicating, and discussing software architectures in their projects [42, 53]. The lack of an agreed-on set of views in software architecture domain has lead industrial developers to define views of their own interest. Example views for describing a system are [42]:

- *logical view*, in which architectural elements are described in terms of objects and classes for the components, and inheritance and association for the connectors;
- *process view*, in which the components are processes and threads, and the connectors are inter-process communication and synchronization methods;
- *implementation view*, in which the components consist of modules and subsystems, and connectors constitute compilation/construction relations;
- *deployment view* describes the hardware platform topology of the system, such as physical processors/interconnection networks, and scheduling of processes/threads to these processors; and
- *user-interface view*, *data view*, *security view*, etc. describe other aspects of a system.

Characteristics of architectural elements

A number of research activities are focusing on the fundamental characteristics of architectural elements, as a basis for classifying them for purposes such as architectural construction, analysis, reuse support, etc. In this direction, an approach for defining “*mechanism matching*” criteria employs a set of core characteristics of elements in order to identify similar elements to be used interchangeably. In an abstract view, two elements are matched when they have the same functionality and signature. The elements can be abstractly characterized by their “*temporal*” and “*static*” properties as follow [41]:

- Temporal characteristics deal with the run-time properties of the receipt or passing control and data. For example, elements can be classified based on the number of threads or the instances they can receive/pass control and data.
- Static properties encompass a number of features which are fixed over time. A static scope is defined for both data and control passing (e.g., over virtual space of a computer or across a network). Other features include: whether the element transforms data, whether it blocks the elements linked to it, the type and number of the elements connected to it, etc.

The above form of describing an element using its properties, does not distinguish between components and connectors. In order to match elements, the designer defines a set of criteria of elementary properties and searches through the architectural elements to find an element with a complete or partial match. Element matching enables an architect to assemble several elements to obtain another element with equivalent characteristics. For example, a set

of procedures and a state-retaining element (e.g., a shared memory or file) can be used as a substitution of an object. Hence, this approach facilitates the task of architectural development, reuse, or delaying design decisions.

2.2.2 Architecture Description Languages

An Architecture Description Language (ADL) is used to document architectural information in order to describe and analyze a system. From the linguistic point of view, an ideal ADL is characterized by six properties [60]:

- *composition*: integration of elements into larger subsystems, or decomposition of a system into its constituents
- *abstraction*: the ability to define abstract views of high-level or low-level design
- *reusability*: the property of using generic patterns (not an instance) of elements
- *configurability*: the capability of changing a software's structure, independent of the components
- *heterogeneity*: the ability of integrating different styles in one system (or integrating modules written in different languages)
- *analysis*: the ability of reasoning about the system

The high level of abstraction provided by the architectural view of a software system completely discriminates ADLs from conventional programming languages. Conventional programming languages deal with details such as algorithm and data structure design. As a candidate for ADL, these languages have shortcomings in all aspects cited above. The highest abstraction level they provide is module (or object); module interconnections are performed in restricted forms; communication protocols are implemented inside the module; analysis is limited to simple type checking; and they fail to isolate modules from each other (e.g., module interfaces rely on *provide/require* services with explicit name bindings).

Module Interconnection languages (MIL) [54] are an attempt to provide glue for integrating separately developed modules, and have had some success in relaxing the configurability and composition of the systems. But still they have many shortcomings to be considered as an ADL, particularly, they are based on name binding and support restricted interactions (procedure call and shared memory).

Carnegie Mellon University (CMU) is currently studying existing and experimental ADLs, to classify ADLs according to their features [16]. The taxonomy is intended to aid ADL researchers or industry in choosing their desired language, or to seek ways of improving the features. A number of ADLs and tool integration environments have been introduced, including [14]: UniCon [5], Wright [6], ACME [1], Gestalt [18] (CMU), Rapide [4] (Stanford University), ArTek [10] (Tecknowledge Corporation), ControlH and MetaH [3] (Honeywell Technology Center).

Features

A set of important features of modern ADLs are presented below:

- the capability of abstractly defining components by the functions or behaviors of the services provided through their interfaces (component abstraction);
- explicitly defining connectors as encapsulation of protocols and data formats required for component interactions (communication abstraction);
- hierarchically refining an architecture, for both components and connectors (abstraction);
- preserving constraints among a set of related architectures, e.g., controlling expansion in a set of hierarchically related architectures (heterogeneity);
- supporting integration of architectural patterns (heterogeneity);
- representing architectures (or styles) as classes, and systems as instances of these classes (reusability);
- simulating architectural prototypes, analyzing the simulation, and generating code (analysis);
- supporting dynamic architectures, i.e., allowing the number of the components or connectors vary during the execution of the system (configurability); and
- architectural recovery from source code (analysis).

A brief introduction of two popular ADLs follows. Rapide is a concurrent event-based simulation language that supports: component and communication abstraction; architectural classes; and dynamic architecture. Its analysis mechanism and architectural simulation are based on using partially ordered sets of events (*poset*). Posets keep track of the time dependencies of the components' behaviors during the parallel execution. Rapide hierarchically expands an architecture, or part of it, using a mechanism called *event pattern mapping*. This mechanism allows us to relate complex behavior of a subsystem to a simple behavior in a higher level of abstraction.

UniCon is a language and a set of tools that support component and communication abstraction, a large set of component and connector types, code generation of the architecture, analysis based on the types of the elements, architectural recovery, and integration with external tools.

Dynamic architecture

The problem of *dynamically* describing an architecture is increasingly important, and some ADL developers seek ways of augmenting their languages with this feature. Except for a very few languages (e.g., Rapide), current ADLs lack any facility for changing the architecture at run-time (i.e., they are *static*). One related approach, “*self-organizing software architectures*”

[43], relies on architectural components to dynamically change the architecture. Darwin is an ADL which allows *simple* bindings of the components through their *required/provided* services. Each new component, upon creation, checks whether its existence is consistent with the architectural description or not. Constraints can be locally evaluated by the components, using sufficient information and actions to perform “self-organizing” operations. In an example of a client-server architecture, each client, upon creation, checks the current number of the clients, and attaches itself to the server if the number of clients does not exceed a threshold.

Architecture interchange language

The variety of characteristics and notations among different ADLs suggests a generic architecture description language. An *architecture interchange language* is a means for constructing a unified environment to be used by different ADLs as a mediator for interchanging architectural design. The language should be a carefully designed intermediate representation of the software architecture in order to obtain the acceptance of a large body of ADL developers as a standard. ADLs translate the architectures into and out of this intermediate model, hence, they must agree on its semantics. A simple and mathematically concrete specification semantic serves as a basis for reasoning about different software architectures described by ADLs. ACME is an architecture interchange language [64] whose kernel allows the definition of primitives such as: components, connectors, and systems comprising of components connected via connectors. Outside the kernel, other facilities such as *templates* for defining styles and aggregates of systems are available.

2.2.3 Architectural Design Process

Architectural design process (or architectural process) is the description of steps whose outcome is the architectural description of a software system.

Problem domain and solution

There are a few approaches to defining a software architecture as a *solution* to a set of related *problems*. One modeling uses Domain Specific Software Architectures (DSSA) to solve a restricted problem and defines the concept of “*problem families*” [25]. A problem is defined as a set of functional or extra-functional requirements for a particular system. A problem family is then a group of problems that have some principal characteristics in common but differ in details. Decomposition of a domain into a set of problem families is performed by classifying the systems of the domain into groups with common requirements. The groups of systems differ in that each has a different “*key requirement*”, which is unique and central to all systems in the problem family. This key is a guide to find the right problem family to search when we have a particular problem definition to solve. After finding the corresponding problem family, the solution architecture is the outcome of mapping the main section of the problem family to the corresponding sections of the DSSA, and implementing the rest of the requirements of the problem family separately. Other approaches, focusing on narrow problem domains, also exist as well as those that propose a general architecture to solve a

more general model of the problem.

Style-based architectural refinement

One solution to the problem of designing an architecture with a specific style, is to transform the existing architecture to a one in different style. The capability of performing inter-style refinement is important in cases when the available analysis tools are specific to a particular style that differ from the system's style, the system is used in an environment adapted to a different style, or it is required to relax the architectural complexity by decreasing the number of its constituent styles. Techniques for converting systems from one style to another, rely on mapping mechanisms as a means for transformation among parts of the systems. Analysis tools then check that the system's consistency has been preserved in this transformation. For example, Rapide uses this mechanism for mapping two sets of events between two systems. Style-centric refinement techniques cover a large number of systems, and are based on a set of rules for style transformation which must be applicable to all transformations among instances. The generality of this method makes it difficult to define mapping rules.

In avoiding the above generality, an approach proposes to work on sub-styles of an style (by defining more constraints on systems) to restrict the set of systems for transformation. The approach claims that there is no unique definition of the refinement, and each transformation should preserve a particular property such as: behavioral (e.g., Rapide), performance, etc [30]

Architectural behavior representation

By capturing the interaction information among the components of an architecture, the designer can simulate the dynamic behavior of the system. Dynamic behavior of an architecture can be represented using an state transition diagram implemented as a data structure. The architectural constraints are enforced by preventing the state machine from entering particular states. The method is known as *Relational Abstraction*. A communication mediator attaches itself between two architectural components (controlled by an analysis tool), in order to monitor the components' interactions and update the state of the data structure. The connector is implemented in the operating system of the host machine, hence, provides a transparent interaction. This mechanism works in conjunction with a particular ADL. The possibility of using ACME (an architectural interchange language) as the ADL of this mechanism is under study, to enable the use of a broad range of ADLs and analysis tools [21].

2.2.4 Architectural Analysis and Evaluation

Research on architectural analysis aims at devising methods, tools, or environments for evaluating software architectures. Novel architectural analysis and evaluation techniques have shown some promise in analyzing large systems for various purposes such as preventing large investments of the software projects with a high risk of big losses [39] and detecting the complexity and problems of a software design [40].

Tools are an essential means for analyzing software architectures. A typical architec-

tural analysis tool is expected to possess capabilities such as [37] describing architectural elements graphically (most commercial tools use objects as components and method invocation as connectors), associating semantics to elements or aggregates of elements, storing the architectural elements in a repository to serve as a database or *data dictionary* for various inquiries, analyzing the architecture against some metrics, and providing useful modeling of the architecture (e.g., control and data flow graphs) for further analysis of the attributes.

SAAM [39], a Software Architecture Analysis Method, assists us in evaluating and ranking the architectures of competing systems in a particular domain. The major task of the analyst is to define a set of carefully chosen task-scenarios in relation to the main stakeholders in the domain. The task-scenarios should reflect the main functional or extra-functional qualities of the systems. The rest of the analysis consists of checking the degree of fitness of these task-scenarios to each competent architecture, and developing a subjective evaluation of the architectures accordingly. SAAMtool, possessing most of the above cited features, is an analysis tool for supporting the SAAM method [37].

Architectural complexity

An approach to the evaluation of architectural complexity involves determining the proportion of the architecture covered by particular patterns (architectural regularity), and the number of different types of patterns composing the architecture [40]. The method uses reverse engineering to extract high-level patterns which are connected graphs of architectural elements. The elements in architecture and patterns are described using their abstract features, and pattern matching is a direct application of mechanism matching described in section 2.2.1. This pattern recognition system is an interactive toolkit for graphically composing patterns and architectures. It analyzes the regularity of architecture and explores some properties such as: fan-in and fan-out of the elements (i.e., the number of elements that control, or are controlled by, an element, respectively), and particular design problems (e.g., layer bridging).

Architectural analysis techniques are not abundant, and most of the existing ones are based on measuring coupling and cohesion of the components [39], or their fan-in and fan-out [36].

Concurrent design and analysis

In an attempt to perform concurrent design and analysis, Argo [61] (an architectural development environment) provides an analysis method based on a set of “*critics*” which are concurrent agents, embedded into the environment [57]. Each critic checks a set of conditions in the evolving architecture, and reports the violated conditions to the designer. A control mechanism manages the reports from all critics in a menu-driven list. Selection of an item for verification causes all information about the violated condition to appear, thereby, providing an interactive analysis mechanism. Different types of critics can be defined to report the violation of syntax and semantics of the design notation, design consistency and completeness. The critics also assist the designer in providing design alternatives.

2.2.5 Architectural Reuse (Styles)

One important cause of the complexity of large software systems is the variety of non-standard building blocks used in the construction of systems. Software standardization, and hence reusability, is a solution to this problem which assists us in constructing reliable, cost efficient, and more understandable systems from off-the-shelf software components. A comprehensive collection of generic software encapsulation (known as software *handbook*), encourages the practitioners to reuse software. Software architecture, by introducing architectural elements as the encapsulation of standard functionalities, encourages reusability in system construction. Concepts such as design patterns and architectural styles imply standardization and reusability of generic constructs in high levels of abstraction. Section 2.1 contains a set of definitions for architectural style from the literature.

In an attempt to address the significance of reusability, DoD CARDS [13] *systematic reuse* project, focuses on ways of collecting and organizing the scattered knowledge on software architecture, specially DSSAs, in a centralized knowledge repository to be used by researchers and practitioners.

Architectural style and design pattern

Design patterns define a collection of guidelines and rules for designing useful models that are repeated in systems [11, 29]. Design patterns have reached a level of popularity and maturity that a rich collection of handbooks, papers, conferences, and special issues of Journals have been dedicated to them [12]. Most design patterns are written in object-oriented languages, but there is no direct correspondence between a pattern and object-oriented design paradigm.

Object-oriented design paradigm, design patterns, and architectural styles are system design facilities that provide different levels of abstraction, from low to high, respectively. An architectural style is a model for generating several systems, whereas, a pattern provides a solution to a particular problem within a system, and an object implements low-level algorithmic problems. In this view, a style resembles a *pattern language* which is a comprehensive collection of patterns to construct a system architecture. The patterns can provide well defined, understandable, and easily analyzable design solutions to be used as building blocks of an architectural style. Not all patterns are intended for mid-level design solutions; pattern handbooks include a set of patterns to be used for architectural level designs [47].

In an attempt to ease the mapping of the problem domain to a design solution, similar to that of the design patterns, a series of activities are underway to classify architectural styles according to the data and control interactions among the architectural elements [59]. The classification criteria include the type of components or connector, the form of sharing, allocating, and transferring control among the components, the form of data communication, and the type of reasoning specific to a style.

2.2.6 Architecture Recovery

Architectural recovery encompasses the various methods for extracting the architectural information from some lower level architectural representation, such as source code.

In recent years, engineers have been confronted with the problem of *legacy* systems, which are old software systems that continue to be used. On average, a software system is operational for 10 to 15 years before it is replaced [63]. Large and expensive projects, such as the \$2-billion American airline reservation system SABRE written in the 70's, have longer lifetimes. These old systems are difficult to maintain, evolve, and integrate with new systems, since they usually lack adequate design documentation. Most of these systems have low level (if any) design documentation which is not sufficient for making systematic changes to the system during their evolution over time. Architectural recovery methods and tools are an effective means for extracting high level design information and lower levels of abstraction views of a legacy system, to assist the engineer in making a change or integration [48].

Reverse engineering

In dealing with the architecture recovery from legacy systems, reverse engineering tools are central. Different reverse engineering techniques can be classified based on methods and tools used in their two main phases. In the first phase, an *extraction* tool converts the source code into an intermediate representation such as relational entities or an abstract syntax tree. In the second phase, an *analysis* tool constructs a high-level view of the system from its intermediate representation. The taxonomy of the existing reverse engineering tools follows [45]:

Filtering and clustering framework: A parser extracts a relational source model from source code and stores it in a database. Then the engineer, using a series of cascading operations on database, provides higher levels of information. This is done by clustering components based on some criteria such as low-coupling and high-cohesion among them. The method has a drawback: the conventional parsers can not extract complex relationships, such as IPC and RPC, among the components, hence, they restrict the level of architectural recovery of the tools. Rigi is an example [62].

Compliance checking framework: The extraction phase is identical to that in the above method (with the same limitation). In the analysis phase, the engineer first defines his/her assumed high level model of the software in an appropriate form (e.g., modules and interrelation, an inheritance hierarchy, a design pattern, or an architectural style). The tool then checks the degree of conformance between the proposed model and source model. Software relaxation model tools are examples [49].

Analyzer generators framework: A parser generates an abstract syntax tree, and stores it in a repository. A query language assists the engineer in generating analysis tools (as a series of queries written in the query language) that makes queries from the source model. Abstract syntax trees are language dependent, hence, the engineer must be familiar with the language syntax. This method is more flexible than the previous ones and is used in conjunction with other methods which follow. Genoa is an example [26].

Program understanding framework: This method uses the knowledge of the engineer about the system to automatically generate high level view of the system's *function-*

ality. The knowledge of the engineer is captured in a knowledge-base, and the source code is represented in an abstract syntax tree. A recognition engine then searches through the source model and knowledge-base with the goal of finding match parts. The result is a hierarchy of recognized patterns which are the user-guided views of the system. DECODE is a tool in this method [55].

Architecture recognition framework: This method employs a series of architectural style *recognizers* that search the abstract syntax tree (source model) for the style (or styles) of the software. A recognizer is written as a set of queries in a query language. The number of the recognizers indicates the richness of the method. If a system consist of several styles, this method can find them one at a time; hence, there is no support for integrating the styles [35].

A relational source model, as the result of the extraction phase, provides ease of the analysis phase and a user-friendly interface. These source models are incapable of providing complex architectural relationship, as an abstract syntax tree does. On the other hand, abstract syntax trees are difficult to manipulate. A new approach suggests a second extraction phase to benefit from the advantages of both [45]. In the first extraction phase, an abstract syntax tree is generated. In the second phase, primitive architectural information about the components and their interactions are produced, using the same methods that recognition engines use to recognize architectural styles. This information is stored in a relational database. The analysis phase can be performed using the first or second method described above, to retrieve architectural information at different levels.

Reverse engineering methods and tools are excellent facilities for architectural recovery, but some researchers believe that this is not sufficient and that a combination of reverse engineering and domain knowledge is necessary. In this direction, the researchers in Vienna University employ the reverse engineering method *architectural recognition framework* (studied above), as well as three complementary architectural recovery methods [28]. The complementary methods aim at providing additional information about the architecture, such as the release history of the software, state transition diagrams (for reactive systems), and matching library templates.

2.2.7 Architecture Development Environment

Research on designing development tools and ADLs has yielded a number of style-specific system development environments and methods, including event-based environments [24], object-oriented design [58], black-board shells [50], dataflow [44], and control systems [23]. Despite employing sophisticated frameworks and efficiency, these systems have some drawbacks: they are expensive, take years to be built, and must be designed from scratch.

In an attempt to avoid the above drawbacks, Aesop [60] was constructed. Aesop uses a generic object model to represent an architecture, and creates an style-specific architecture construction environment by specializing the generic object model (using subtyping). The object model includes a few styles for direct designing of those styles. The created environment provides a set of component and connector types corresponding to the employed style,

a control mechanism to ensure the consistency of the composed elements of the developed system with the applied style, analysis tools to analyze the system's attributes, and a graphical visualization mechanism to view and design the artifact. Aesop is still experimental, but its approach has had some success in developing small scale development environments.

References

- [1] Architecture Description Language, ACME. Web site, URL = <http://www.cs.cmu.edu/Groups/able/acme-web/>.
- [2] DMSO. Web site, URL = <http://www.dmsomil/docslib/hla/>.
- [3] Architecture Description Language, ControlH & MetaH. Web site, URL = http://www.htc.honeywell.com/projects/dssa/dssa_tools.html.
- [4] Architecture Description Language, Rapide, URL = <http://anna.stanford.edu/rapide/rapide.html>.
- [5] Architecture Description Language, UniCon. Web site, URL = <http://www.cs.cmu.edu/People/UniCon/>.
- [6] Architecture Description Language, Wright. Web site, URL = <http://www.cs.cmu.edu/Groups/able/able.html>.
- [7] ISO/OSI Reference Model. Web site, URL = <http://cougarnet.byu.edu:80/acd1/ed/InSci/Projects/osilinks.html>.
- [8] Loral, Architecture Avionics Domain Application Generation Environment. Web site, URL = <http://www.ai.mit.edu/projects/adage/adage.html>.
- [9] DSSA Perspective and Contributions, Command and Control (C2). Web site, URL = <http://www.isi.edu/software-sciences/dssa.html>.
- [10] Teknowledge's DSSA & ProtoTech Projects. Web site, URL = <http://www.teknowledge.com/DSSA/>.
- [11] J. O. Coplien. Software Design Patterns: Common Questions and Answers. Web site, URL = <http://st-www.cs.uiuc.edu/users/patterns/papers/>.
- [12] Patterns Home Page. Web site, URL = <http://st-www.cs.uiuc.edu/users/patterns/patterns.html>.
- [13] Department of Defense ARPA, CARDS Project. Web site, URL <http://dealer.cards.com/>.
- [14] Carnegie Mellon University, SEI. Web site, URL = <http://www.sei.cmu.edu/architecture/adl.html>.
- [15] Carnegie Mellon University, SEI. Web site, URL = <http://www.sei.cmu.edu/architecture/definitions.html>.
- [16] Carnegie Mellon University, SEI. Web site, URL = <http://www.sei.cmu.edu/architecture/>.

- [17] Department of Defense, USA. Web site, URL = <http://www-ast.tds-gn.lmco.com/arch/arch-008.html#foundations>.
- [18] Architecture Description Language, Gestalt. Web site, URL = <http://www.sei.cmu.edu/architecture/IWSSD8.Gestalt.paper.html>.
- [19] G. D. Abowd, L. Bass, et al. Structural modeling: An application framework and development process for flight simulators. Technical Report CMU/SEI-93-TR-14, Carnegie Mellon University (Software Engineering Institute), Pittsburgh, Pennsylvania, 1993.
- [20] R. Allen. Hla: A standards effort as architectural style. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 130–133, 1996.
- [21] R. Balzer. Enforcing architectural constraints. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 80–82, 1996.
- [22] A. Berson. *Client/Server Architecture*. McGraw Hill, 1992.
- [23] P. Binns and S. Vestal. Formal real-time architecture specification and analysis. In *Tenth IEEE Workshop on Real-Time Operating Systems and Software*, New York, May 1993.
- [24] M. R. Cagan. The HP softbench environment: An architecture for a new generation of software tools. *Hewlett-Packard journal*, pages 36–47, June 1990.
- [25] J.-M. DeBaud. Viewing a dssa in context: Problem versus solutions. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 19–23, 1996.
- [26] P. T. Devanbu. Genoa - a customizable, language and front_end independent code analyzer. In *Proceedings of the 14th ICSE*, pages 307–317, May 1992.
- [27] C. Gacek, A. Abd-Allah, B. Clark, and B. Boehm. On the definition of software system architecture. In *ICSE 17 Software Architecture Workshop*, April 1995.
- [28] H. Gall, M. Jazayeri, R. Klosch, W. Lugmayr, and G. Trausmuth. Architecture recovery in ares. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 111–115, 1996.
- [29] E. Gamma, R. Helm, et al. Design patterns: Abstraction and reuse of object-oriented design. In O. M. Nierstrasz, editor, *ECOOP'93 Object-Oriented Programming (7th European Conference)*, pages 26–30, Kaiserslautern, Germany, July 1993. Springer-Verlag.
- [30] D. Garlan. Style-based refinement for software architecture. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 72–75, 1996.
- [31] D. Garlan and D. E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274, April 1995.

- [32] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39. World Scientific Publishing Company, 1993.
- [33] W. W. Gibbs. Software’s chronic crisis. *Scientific American*, pages 86–95, September 1994.
- [34] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
- [35] D. R. Harris, H. B. Reubenstein, and A. S. Yeh. Reverse engineering to the architectural level. In *Proceedings of the 17th ICSE*, pages 186–195, April 1995.
- [36] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transaction on Software Engineering*, 7(5), September 1981.
- [37] R. Kazman. Tool support for architecture analysis and design. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 94–97, 1996.
- [38] R. Kazman. Software architecture: Definitions, examples, and analysis short course. The Institute for Computer Research, University of Waterloo, January 13-14 1997. Waterloo, Canada.
- [39] R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. *IEEE Software*, pages 47–55, November 1996.
- [40] R. Kazman and M. Burth. Assessing architectural complexity. In *CSMR*, pages 104–112, 1998.
- [41] R. Kazman, P. Clements, G. Abowd, and L. Bass. Classifying architectural elements as a foundation for mechanism matching. In *Submitted to COMPSAC 1997*, 1997.
- [42] P. Kruchten. Software architecture – a rational metamodel. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 5–7, 1996.
- [43] J. Magee and J. Kramer. Self organizing software architectures. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 35–38, 1996.
- [44] V. W. Mak. Connections: An inter-component communication paradigm for configurable distributed systems. In *Proceedings of the International Workshop on Configurable Distributed Systems*, London, UK, March 1992.
- [45] N. C. Mendonca and J. Kramer. Requirements for an effective architecture recovery framework. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 101–105, 1996.

- [46] R. T. Monroe. Capturing design expertise in software architecture design environments. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 87–89, 1996.
- [47] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural styles, design patterns, and objects. *IEEE SOFTWARE*, 14(1):43–52, January/February 1997.
- [48] G. C. Murphy. Architecture for evolution. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 83–86, 1996.
- [49] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion model: Bridging the gap between source and higher-level models. In *In proceedings of the 3rd ACM SIGSOFT SFSE*, pages 18–28, October 1995.
- [50] H. P. Nii. Blackboard systems. *AI Magazine*, 7(3):38–53, 1986.
- [51] J. Q. Ning. Where does architecture research meet practice. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 138–142, 1996.
- [52] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40–52, October 1992.
- [53] J. S. Poulin. Evolution of a software architecture for management information systems. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 134–137, 1996.
- [54] R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6(4):307–334, November 1986.
- [55] A. Quilici and D. N. Chin. Decode: A cooperative environment for reverse-engineering legacy software. In *Proceedings of the 2nd Working Conference on Reverse Engineering*, pages 156–165, July 1995.
- [56] S. P. Reiss. Connecting tools using message passing in the field program development environment. *IEEE Software*, July 1990.
- [57] J. E. Robbins, D. M. Hilbert, and D. F. Redmiles. Using critics to analyze evolving architectures. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 90–93, 1996.
- [58] J. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliff, New Jersey, 1991.
- [59] M. Shaw and P. Clements. Toward boxology: Preliminary classification of architectural styles. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 50–54, 1996.
- [60] M. Shaw and D. Garlan. *Software Architecture*. Prentice-Hall, 1995.

- [61] R. N. Taylor et al. A component and message-based architectural style for gui software. *IEEE Transactions on Software Engineering*, to appear.
- [62] S. R. Tilley, H. A. Muller, M. J. Whitney, and K. Wong. Domain-retargetable reverse engineering. In *Proceedings of the International Conference on Software Maintenance*, pages 142–151, September 1993.
- [63] E. Wallmuller. *Software Quality Assurance: A Practical Approach*. Prentice Hall, New York, 1994.
- [64] D. Wile. Semantics for the architecture interchange language, acme. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 28–30, 1996.
- [65] J. A. Zachman. A framework for information systems architecture. *IBM Systems Journal*, 26(3):276–292, 1987.