MATCH
Communications in Mathematical and in Computer Chemistry

MATCH Commun. Math. Comput. Chem. **60** (2008) 869-882

ISSN 0340 - 6253

THE ARCHITECTURE OF SOFTWARE SYSTEMS FOR MOLECULAR TOPOLOGY

BAZIL PÂRV
Department of Computer Science, Faculty of Mathematics and Computer Science, Babes-Bolyai University, 1, M. Kogălniceanu, Cluj-Napoca 400084, Romania
e-mail: bparv@cs.ubbcluj.ro

(Received April 19, 2008)

ABSTRACT. The concept *software architecture* becomes more and more important in the software development process. Software system design is a two-step process: architectural design and detailed design. Architectural design describes top-level structure and organization of a software system, identifying its components, and is considered today the most important part of the overall design process. At least three criteria can be used to discuss different architectural patterns (aka architectural styles): system organization, modular decomposition, and control strategy. According to Sommerville, system organization is concretized in three architectural styles: shared data repository, client-server, and layered model; there are two main models answering to modular decomposition: object, and dataflow; finally, control strategy has two styles: centralized and event-driven. The paper presents in detail all these architectural styles, their advantages and drawbacks, and their use in molecular topology software.

1. INTRODUCTION

Software architecture is a relatively new software engineering discipline, having an increased impact on the ways a software system is designed and implemented today. We believe that its important concepts, principles, and models need to be known by all people involved in the development of software systems, including scientific software.

According to [Shaw96], *software architecture* defines a software system in terms of its structure and topology (components and interactions) and shows the correspondence between the system requirements and elements of the constructed system, providing some rationale for design decisions. The *computational components* of a system are clients, servers, databases, filters, layers and so on, while *interactions* among those components can be either simple (procedure call, shared variable access) or complex, semantical rich (client-server or database access protocols, asynchronous event multicast, piped streams, and so on). Relevant *system-level issues* at the architecture level are capacity, throughput, consistency, and component compatibility.

Architectural models can be expressed in several ways, from box-and-line diagrams to architecture description languages, and clarify structural and semantic differences among components and interactions. They answer questions like: how components can be composed to define larger systems, or how individual elements of architectural descriptions are defined independently, so they can be reused in different context, refined as architectural subsystems and implemented in a conventional programming language.

This paper is organized in four sections, including this one. Second section introduces three important issues considered to be part of software architecture discipline, while the third contains some remarks regarding the software architecture of molecular topology programs. The last section contains some conclusions.

2. ARCHITECTURAL STYLES

High-level design of a software system can be discussed in different ways, like architectural views (see [Kru95]) and architectural styles. This section refers to the latter ones.

Architectural design is a creative process, depending on the type of the target system. However, there are a number of common decisions that span all design processes. Among the high-level design questions enumerated by Sommerville (2004), the most important are referring to the general organization of the system, its decomposition into subsystems and modules, and its control strategy.

The concept of *architectural style* was introduced by Shaw and Garlan [Shaw94]. They consider the architecture of a software system as a collection of components and connectors describing the interactions among components, described as a graph with components as nodes and connectors as arcs. Following this approach, an architectural style defines a family of software systems with the same structural organization. In other words, an architectural style is defined by three sets of constructs: (a) the *components*, (b) the *connectors*, and (c) the *constraints* on how components and connectors can be composed.

Sommerville (2004) considers that subsystems and modules are different: the subsystem is a system operating independently of the services provided by others, while the module is a system component that provides/uses services to/from other modules, but is not considered as a separate system. He describes five types of models that define a system's architecture: *static* (structural model, showing major components of the system), *dynamic* (process model, showing the process structure of the system), *interface* (defining subsystem interfaces), *relationships* (showing subsystem relationships, and *distribution* (showing subsystem distribution across network). In this more general framework, architectural styles are organized in three main groups, depending on the main question they answer: system organization, modular decomposition, and control strategy.

2.1. Architectural styles reflecting system organization

System organisation reflects the basic strategy used to structure a software system. The most important architectural styles are: shared data repository, shared services and servers (client-server model), and abstract machine (layered model).

2.1.1. Shared data repository

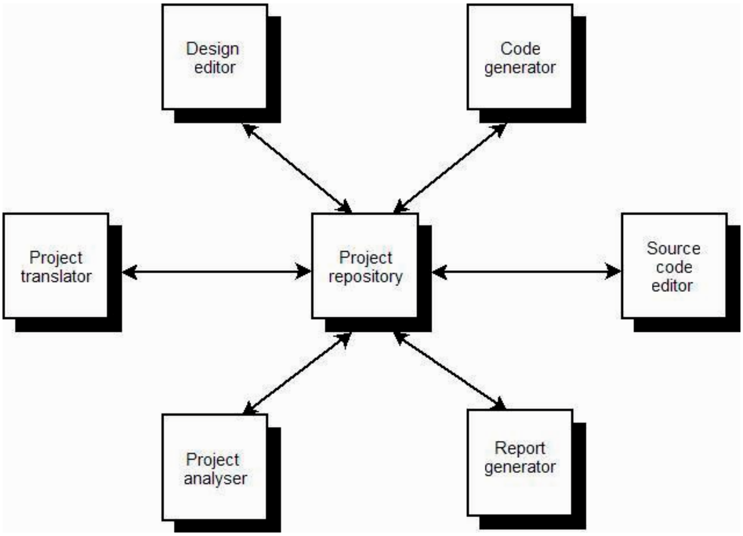

Figure 1. Shared data repository model

Shared data repository model (Figure 1) has two categories of components: a central data structure (data store) and a set of independent components (subsystems) that operate on the data store. The connectors between subsystems and the data store are interactions, i.e. different types of transactions.

The set of constraints is represented here by the control discipline. There are two variations here: *traditional database model*, in which types of transactions in the input stream trigger selection of the process to execute (according to the *pull programming model*), and *blackboard model*, in which the current state of the central data structure is the main trigger of selecting process to execute (according to the *push programming model*).

Figure 1 above represents the architecture of a CASE toolset [Som04], in which the central data store is the project repository.

Advantages of this architecture are straightforward. It represents an efficient way to share large amounts of data between its subsystems, which need not to be concerned with how data is produced. In order to consume and produce data, the subsystems need to know the repository schema. The data model represents the main drawback of this architecture: all subsystems must agree on a common representation of data, usually a compromise. Moreover, any change in this model is difficult and expensive.

2.1.2. Client-server organization

Client-server organization (Figure 2) corresponds to the *request-response paradigm*. Its main components are the (database) server(s) and the client applications; the connectors are procedure calls. A client issues a request at a time to the server (using a procedure call, i.e. explicit invocation); the server is faced to deal simultaneously with many clients, processing their requests and sending back responses to them.

Clients and servers cooperate in order to cover all logical functions (or layers) of an interactive application [Fowler02]: *presentation logic* (presenting data to the user, capturing user interaction), *application logic* (application-specific processing), and *data management* (storing and retrieving persistent data).

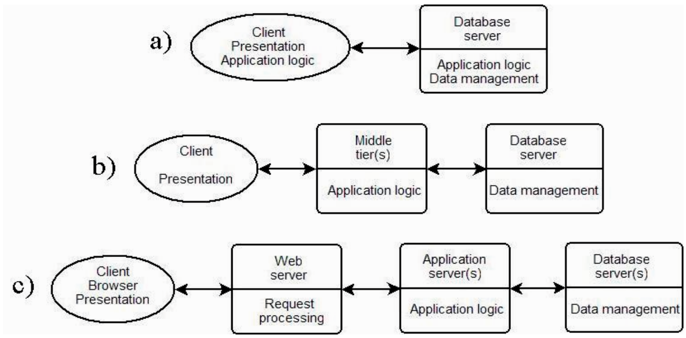

Figure 2. Client-server model: (a) two-tier, (b) three (n)-tier, (c) web-based

All variations sketched in Figure 2 consider that presentation logic is located in the client application, while data management functions reside on the server side. They differ in the ways components implement application logic: *two-tier client-server* splits application logic between the client and the server, while *three* (*n*)-*tier client-server* considers intermediate (middle) tier(s) dedicated to application logic, freeing clients and servers of application-specific duties. *Web-based* approach belongs to the latter scheme, employing a thin universal client (running a Web browser), always connected to a Web server. In this case, Web server connects to one or many application server(s), which on their turn make calls to database server(s).

Usually, client-server architecture is a common method for distributing computer power within an enterprise, where many users (clients) connect to and share some processing resources and data (servers). As Duchessi et al. [Duc98] noted the main benefits are: improved integration of shared data, improved accessibility, and reduced costs, by using cheaper client hardware. In the same time, there are some technical problems to be addressed when such architecture is intended to be used, related to computer architecture, management and organization, and conversion and maintenance. There is no shared data model, so different logical subsystems (clients) use different data organization. Additionally, because there is no central register of names and services, all clients need to know exactly what servers and services are available.

2.1.3. Layered model

In the layered model (Figure 3), each layer represents a component, while the connectors are, as in the client-server organization, procedure calls. Layers are seen as abstract machines, each defining its own application programming interface (API) to be used by its clients, i.e. other layers. The constraints are of topological nature, devising two alternatives: strict layering and non-strict layering.

In the strict *layering variant*, each layer communicates only with the adjacent layers, the other (inner or outer) layers being hidden. The *non-strict layering* approach means: (a) certain functions of inner layers are exported to the outer layers, *or* (b) the inner layers are not hidden from the outer layers, *or* (c) some connectors are used to determine how layers will interact.

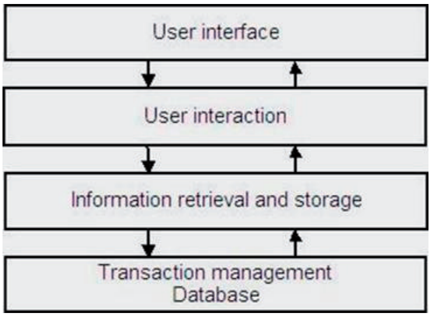

Figure 3. Layered model

Figure 3 above depicts the strict layered architecture of an interactive application, illustrating the functionality of each layer.

Layered systems provide good support for design, maintenance, and reuse. By supporting design based on increased level of abstraction, they allow designers to decompose a complex problem into a sequence of incremental steps. Because each layer interacts with a few other layers, the coupling between them is kept to a minimum. Abstract machine view of a layer (i.e. defining standard interfaces or APIs) allows its different implementations to be used interchangeably.

The drawbacks of layered architecture are related to performance and design issues. Considerations of performance may require tighter coupling between layers (i.e. logically high-level functions and their lower-level implementations). Moreover, not all systems can be easily structured in a layered fashion; usually it is difficult to find the right levels of abstraction.

2.2. Architectural styles reflecting modular decomposition

Modular decomposition deals with the decomposition of subsystems into modules, and main decomposition models are *dataflow* (pipe-and-filter) *model* (the system is decomposed into functional modules which transform inputs to outputs) and *object model* (the subsystem is decomposed into a set of interacting objects).

2.2.1. Pipe-and-filter model

In this model (known also as *dataflow model*), shown in Figure 4, filters are computational components, while pipes represent connectors, serving as communication media between filters.

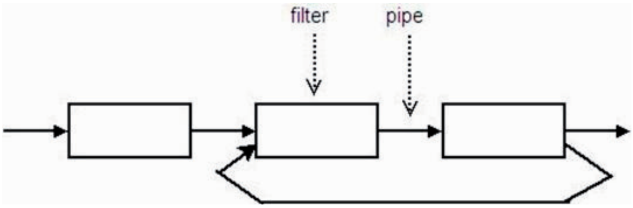

Figure 4. Pipe-and filter model

Filters are independent entities; they don't know their neighbors. A filter starts processing when input data are available on incoming pipes. Pipe-and-filter architecture has three common specializations: *pipelines,* linear sequence of filters; *bounded pipes*, where the amount of data that reside on a pipe is restricted; and *typed pipes*, which require that the date passed between two filters have a well-defined type. *Batch sequential systems* represent a special case of this architectural style, where each filter processes all of its input data as a single entity, and the pipes no longer serve the function of providing a stream of data. Usually this style is treated separately.

Benefits of this architecture, prescribed by the functional decomposition, are related especially to the enhanced reusability and maintainability of filters. They allow the designer to understand the overall input/output behavior of a system as a simple composition of the behaviors of the individual filters. Any two filters can be connected, provided they agree on the data that is transmitted between them; new filters can be added to the existing systems and old filters can be replaced by new and improved versions. Finally, this architectural style supports concurrent execution, each filter being implemented as a separate task, potentially executed in parallel with other filters.

Disadvantages of this architecture are related to the data representation. Because pipe-and-filter style is function-oriented, data representation is a second-class citizen. Changing data representation may produce the need to maintain correspondences between two separate, but related streams. In addition, this architecture is not good at handling interactive applications, because of its transformation character.

2.2.2. Data abstraction and object-oriented organization

This architectural style considers *objects* as computational components, and *messages* (method invocations, explicit invocation) as connectors. Objects are instances of classes (implementations of abstract data types). An object-oriented program is a structured collection of communicating objects.

Each object is responsible for preserving its state; object representation is hidden and inheritance does not have a direct architectural function. Inheritance and object composition are main mechanisms for code reuse.

Benefits of object systems are related especially to two object design principles: *separation of concerns* (each object has a well-defined role) and *program to an interface*. Object implementation be changed without affecting its clients (object representation is hidden), and the resulting object-oriented program is a collection of interacting agents. This leads to more flexible, maintainable and extensible systems.

Of course, all above advantages have a cost. Each class of objects in the system needs a separate design, implementation, and testing process. In addition, in order for one object to interact with another, the first must know the identity of the second. Whenever the identity of an object changes, all other objects that use it need to be notified in some way. This is in contrast to pipe and filter systems, where filters are totally independent.

2.3. Architectural styles reflecting control strategy

Control strategy is concerned with the control flow between subsystems, which is distinct from the system decomposition model. Two main control styles are employed: centralised control and event-based control.

2.3.1. Centralized model

In the *centralised model*, components are subsystems, while connectors represent control flow. One of the subsystems has the overall responsibility of the system, managing the execution of all other subsystems. Centralized control comes in two flavors: call-return model, and manager model. *Call-return model* (Figure 5) is applicable to sequential systems only: the control starts at the top of the component (subroutine) hierarchy and moves downwards. *Manager model* (Figure 6) is good for concurrent systems: one system component (subsystem, monitor) controls the starting, stopping, and coordinating all other subsystems (system processes).

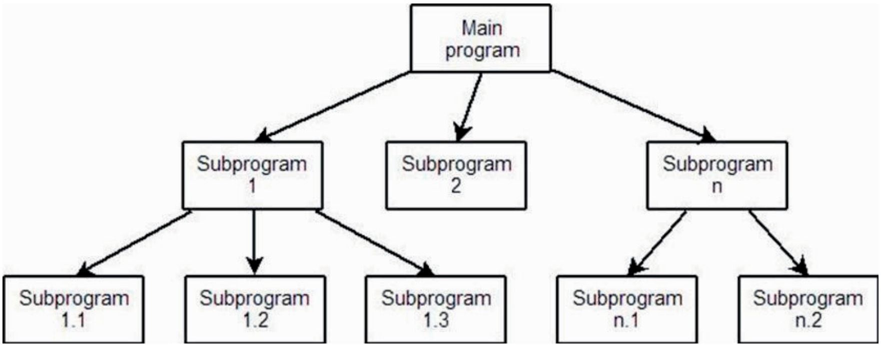

Figure 5. Call-return model

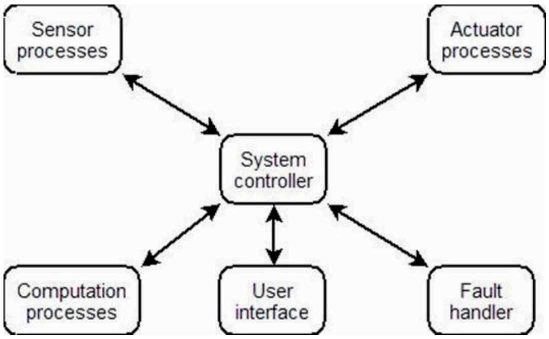

Figure 6. Manager model

In the call-return model, subsystems (subprograms) are invoked using traditional subprogram calls. This was the main control model driven by FORTRAN programming language.

2.3.2. Event-driven control

Systems employing *event-driven control* have no master controller. Subsystems react to externally generated events, which are not under their control. Two main event-driven models are used: broadcast models and interrupt-driven models. In *broadcast models*, the event is broadcast to all subsystems, and any of the subsystems can handle it. *Interrupt-driven models* are used in real-time systems, and use hardware interrupts processed by interrupt handlers.

From our viewpoint, event-driven systems (Figure 7) are of a particular interest. Their *components* are modules, whose interfaces provide a collection of methods (incoming interface) and a set of events (outgoing interface). Methods pertaining to incoming interface are called by explicit invocation, while events that constitute the outgoing interface are subject to implicit invocation (connectors): registered procedures will be called when the corresponding events occur at runtime. The announcer of events does not know which components will be affected by those events: event consumers are dynamically registered to event sources.

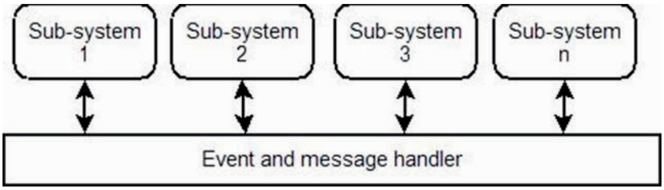

Figure 7. Event-based model - broadcast

Benefits of event models are their strong support for reuse and for system evolution. Any new component (event consumer) can be introduced into a system (event source) by simply registering it for the events of that source. Moreover, a component may be replaced by other components without affecting the interface of other components in the system.

Disadvantages of this model are related to the undefined response to events, data exchange, and reasoning about correctness. When a event source fires an event, it cannot assume event consumers will respond to that event. In the case there are registered more event consumers and they react to an event, the order in which they respond is undefined. Finally, traditional reasoning about procedure calls does not apply in the case of implicit invocation.

3. SOFTWARE ARCHITECTURE OF MOLECULAR TOPOLOGY PROGRAMS

Molecular topology (a.k.a. chemical graph theory) is an interdisciplinary science that using tools taken from graph theory, set theory, and statistics, attempts to identify structural features involved in structure-property-action relationships.

Typically, molecular topology software involves a lot of processing and usually some standard (reusable) data, regarding the structure and constituents of a chemical molecule, made up from atoms and bonds. Usually, chemical molecules are represented in graph form (a.k.a. *molecular graph*), with atoms as vertices and covalent bonds as edges. Of course, structures that are more complex need special attention.

From a logical viewpoint, a software system has three

This section contains some remarks concerning high-level design of molecular topology software. First, we have to sketch the main features of such computer programs, in terms of their data representation and processing. After that, we can issue some remarks regarding optimal choices for their architecture.

3.1. Data representation

As [Fogh05] remarks, there is a lack of standards for data storage and exchange in scientific software. The problem lies not only in defining and maintaining the standards, but also in convincing scientists and application programmers with a wide variety of backgrounds and interests to adhere to them.

Traditional bookkeeping approaches, such as the laboratory notebook, do not scale the multiuser and multisite environments. In [Paj05], two possible approaches to allow the data to be shared between users and applications are discussed. The first is to define a general and unique data format to transfer data between applications (i.e. a single format meeting the needs of all users/applications), while the second is to build a data model.

The first approach is supported by some well-known computer applications, e.g. HYPERCHEM, which impose a de facto standard in data representation. This way, the researcher can use them (i.e. HYPERCHEM) for specific processing of the chemical copounds he/she proposes. For example, we designed the output of our experimental programs (TORUS) generating new toroidal structures in the HYPERCHEM format, which allowed us to visualize them in a HYPERCHEM window. This is an example of a *pipeline* architecture, where filters are TORUS and HYPERCHEM, and the pipe is represented by a HYPERCHEM-complaint data file.

The second approach involves a more dedicated work. The idea is to identify and define the classes of information of interest along with their relationships, in order to produce a structural description (static view) of all the data, i.e. a *data model*. Such a general data model provides well-defined interfaces for data manipulation and can be used by different computer applications in order to perform specific tasks. This way, the data model and the processing are separated and can evolve in an independent way.

The use of a data model has implications in application and data management layers, as they are defined in 2.1.2. In this case, application layer will contain application-specific objects, while data management layer is responsible with the maintenance of the data model contained in the persistent store.

From our architectural viewpoint, the use of a data model suggests either *repository* architecture or a *client-server* one. In the shared data repository model, the central data store contains the data model, while independent components (subsystems) are dedicated to specific processing needs. In the client-server model, the database server performs data management functions, while client applications are designed to cover specific processing functions.

3.2. Typical processing

Molecular topology processing involves matrix and graph algorithms (see [Diu00] for an exhaustive review). The main categories of outputs are: topological matrices, topological indices, symmetry and similarity studies, and property modeling using QSPR (Quantitative Structure-Property Relationship)/QSAR (Quantitative Structure-Activity Relationship), as well as molecular modeling. All these results involve a lot of computational power and internal storage.

Chemical graphs can be represented in list form (as a sequence of numbers), ploynomial form, or matrix form. There are many matrix representations, collectively known as topological matrices: adjacency matrix, Laplacian matrix, distance and distance-extended matrices, detour matrix, 3D - distance matrix, path and distance-path matrices, Wiener, Szeged, Hosoya, Schultz, and Cluj matrices, reciprocal and walk matrices, layer and sequence matrices.

The characteristic polynomial of a graph is the most popular and most extensively used polynomial in molecular topology. There are many variations of it, such as the matching polynomial, the μ-polynomial and the β-polynomial. They are studied in order to find some specific properties of their roots.

Other processing needs are related to the graphical visualization of chemical structures and to the data management functions.

3.3. Recommended architectures for molecular topology software

In what follows, we briefly discuss some ideas regarding the architecture of three important classes of molecular topology applications and subsystems: processing-intensive, data-intensive, and user interaction-centered.

Processing-intensive components and applications involve usually few data and many processing power. This is the case when the specific processing involves the computation of molecular topology matrices, indices, or polynomials. In addition, graphical visualization involves a lot of matrix computation. The recommended architecture in this situation is *dataflow* (best for sequential and well-defined processing steps), where each filter computes a specific topological matrix or index. In addition, where the computation complexity is a big issue, each filter can be organized in a *layered* fashion.

A *central data store* is recommended for data-intensive applications and components. This situation is typical when the same large amounts of static data are used again and again in different processing steps and moments. Because of the static nature of such data, it is recommended to consider data entry as a separate processing step. Such an example is presented in [Paj05]

Finally, user interaction-centered components and applications are best designed by employing *implicit invocation* architectures. They provide the necessary flexibility and reliability in the presentation of data and in capturing and processing the user interaction.

REFERENCES

[Bass98] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998.

[Deu89] L. P. Deutsch, *Design reuse and frameworks in the Smalltalk-80 system*. In T. J. Biggerstaff, A. J. Perlis, editors, *Software Reusability, Volume II: Applications and Experience*, Addison-Wesley, Reading, MA, 1989, 57–71 pp.

[Diu00] M.V.Diudea, I. Gutman, J. Lorentz, *Molecular topology*, Nova Science Publishers, 2000.

[Duc98] P. Duchessi, I. Chengalur-Smith, Client/Server benefits, problems, best practices, *Comm. ACM* **41** (1998) 87-94.

[Fogh05] R.H. Fogh et al., A framework for scientific data modeling and automated software development, *Bioinformatics* **21** (2005) 1678-1684.

[Fowler02] M. Fowler et al., *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2002.

[GoF95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[GS03] J. Greenfield, K. Short, *Models, Frameworks, and Tools*, Wiley, 2003.

[Paj05] A. Pajon et al., Design of a Data Model for Developing Laboratory Information Management and Analysis Systems for Protein Production, *Proteins: Structure, Function, and Bioinformatics* **58** (2005) 278-284.

[Shaw94] M. Shaw, D. Garlan, *An Introduction to Software Architecture*, CMU/SEI-94-TR-21, 1994.

[Shaw96] M. Shaw, D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

[Som04] J. Sommerville, *Software Engineering*, 7th ed., Addison-Wesley, 2004