THE ARCHITECTURE OF SOFTWARE SYSTEMS AND COMPUTING CURRICULA

Bazil Pârv

Abstract. The concept *software architecture* becomes more and more important in the software development process. As a matter of fact, [9] states that software design is considered 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. Consequently, there is a need for updating computing curricula with new disciplines, related to software architecture. This paper contains such a proposal, including three new courses: *Design patterns, Software architecture*, and *Framework design*.

2000 *Mathematics Subject Classification*: 68U35.

1. Introduction

According to [7], *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 sketches how these issues can be implemented as different courses in a computing curricula. The last section contains some conclusions.

2. Software architecture: main issues

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.

This section briefly introduces three main topics related to software architecture: architectural views and styles, design patterns, and frameworks.

2.1. Architectural views, styles, and models

High-level design of a software system can be discussed in different ways. This sub-section introduces two approaches: architectural views and architectural styles.

According to Phillipe Kruchten [5], the high-level design of a software system is composed by five views (Figure 1).

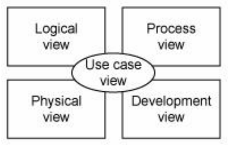

Figure 1: The 4+1 View Model

Each of the views in Figure 1 highlights some specific elements of the software system, intentionally suppressing others.

The four views in the "4+1" view model are: logical view, development view, process view, and physical view. The *logical view* deals with behavioral requirements and shows the decomposition of a software system into a set of abstractions, using among others UML class diagrams and interaction diagrams. The *development view* describes (using UML package diagrams) how the components of the system are nested. The components are bigger building blocks than classes and objects: packages, modules, subsystems, class libraries and so on. Using UML activity diagrams, the *process view* describes the system's processes and how they communicate, while the *physical view* illustrates how the application is installed and how it works in a computer network, using UML deployment diagrams and taking into account non-functional requirements like availability, reliability, performance, and scalability.

The additional view in the "4+1" view model is the *use-case view*, which describes the functionality of the system, using UML use case diagrams and textual/structured descriptions of scenarios. Figure 1 shows this view interacting with all other views, in both directions: use cases are used to explain the elements described by the other views, and other views also utilize use cases.

The concept of *architectural style* was introduced by Shaw and Garlan [6]. 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.

In their paper, Show and Garlan describe mainly six pure architectural styles: pipes + filters, data abstraction + object-oriented organization, event-based + implicit invocation, layered systems, repositories, and table-driven interpreters, as well as some heterogeneous ones.

The *architectural model* of a system may conform to a generic architectural model or style. Unfortunately, most large systems are heterogeneous: they do not follow a single (pure) architectural style. Moreover, the architectural styles, as introduced by Shaw and Garlan, do not cover all facets/views of a software system discussed by P. Kruchten [5]. Sommerville (2004) considers that subsystems and modules are different: the subsystem is a system operating independent of the services provided by others, while the module is a system component which 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 organisation, modular decomposition, and control strategy.

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

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

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.

In the *centralised control*, 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* is applicable to sequential systems: the control starts at the top of the component (subroutine) hierarchy and moves downwards. *Manager model* is applicable to concurrent systems: one system component (subsystem, monitor) controls the starting, stopping, and coordinating all other subsystems (system processes).

Systems employing *event-driven control* use externally generated events, which are not under the control of the subsystems that processes them. 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.

2.2. Design patterns

If architectural styles and models describe the high-level organization of a *whole* system or subsystem, *design patterns* (a.k.a. *microarchitectural patterns*) provide the solution for a *part* of a system/subsystem.

Design patterns are intended to record experience in designing object-oriented systems by providing a common framework which names, explains, and evaluates important and recurring designs.

Design pattern concept was introduced in a well-known book by E. Gamma, R. Helm, R. Johnson and J. Vlissides (a.k.a. Gang of Four, GoF) [3]. This book is the first catalog of design patterns, classified by two criteria (see Table 1).

The first criterion, *purpose*, denotes what a pattern does. [3] catalog contains three classes of patterns: creational, structural, and behavioral. *Creational* patterns are used to create objects, *structural* patterns deal with the composition of classes or objects, while *behavioral* patterns characterize the ways in which classes or objects interact and distribute responsibility.

The second criterion, *scope*, specifies the target of the pattern: class or object. Class patterns deal with relationships between classes and subclasses, i.e. *inheritance*, while object patterns deal with relationships between objects, i.e. *object composition.* Class relationships are static - fixed at compile time, and object relationships are more dynamic - they can be changed at run-time.

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Intrpreter Template method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsability Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Table 1. GoF design patterns

The classification in Table 1 is orthogonal: most of the 23 design patterns discussed belong to a single combination of the two criteria. So, *creational class patterns* defer some part of object creation to subclasses, while *creational object patterns* defer object creation to other object; *structural class patterns* use inheritance to compose classes, while *structural object patterns* describe ways to compose objects; *behavioral class patterns* use inheritance to describe algorithms and control flow, while *behavioral object patterns* describe how a group of objects cooperate to perform a task.

Also, [3] introduces two important object-oriented design principles which lie behind design patterns: (1) *Program to an interface, not an implementation* and (2) *Favor object composition over class inheritance.*

Manipulating objects solely in terms of the interface defined by abstract classes has two major benefits: (a) clients remain unaware (i.e. independent) of the specific types of objects they use, as long as these objects comply to the interface that clients expect and (b) clients remain unaware of the classes that implement these objects.There are two major techniques for reusing functionality in object-oriented systems: class inheritance and object composition.

Class inheritance allows the definition of one sub-class in terms of other already defined classes. Reuse by subclassing is known as *white-box reuse*: with inheritance, the implementation of parent classes is often visible to subclasses. Main advantages of this reuse technique are: simplicity, ease of use, direct support provided by programming languages, ease of change. The major disadvantages are: the inheritance relationship is statical - at compile time, and "inheritance breaks encapsulation" - the implementation of a subclass needs to know and is tightly bound to the implementation of the base class(es) - any change in the parent's implementation will force the subclass(es) to change. This is known as the *fragile base class problem.* In short, implementation dependencies can cause problems when reusing a subclass.

Object composition requires that the objects subject to composition have well-defined interfaces. With object composition, complex functionality is obtained by composing (assembling) objects; this kind of reuse is known as *black-box reuse*: no implementation details are needed. Object composition is defined at run-time, and uses objects with well-defined interfaces; accessing objects only through their interfaces, encapsulation is not broken and objects can be replaced dynamically if they implement the same interface. Moreover, the use of interfaces reduces the number of implementation dependencies.

Design patterns solve many everyday problems in object-oriented design:

finding appropriate objects during decomposition of a system, determining object granularity, specifying object interfaces and implementations, relating run-time and compile-time structures, and designing for change. They introduce a common design vocabulary and improve substantially the design process.

2.3. Application programs, toolkits and frameworks

Software engineering process is producing a large variety of object-oriented software: application programs, toolkits, and frameworks. In each such a subclass, the reuse of software has a different meaning.

In the case of *application programs*, high priorities are internal reuse, degree of maintainability, and degree of extensibility. Internal reuse means parcimony during software construction: only desired features are designed and implemented. The use of design patterns can greatly reduce dependencies, provide loose coupling and encapsulate specific operations and representations.

Toolkits are predefined classes from one or more libraries, designed to provide useful, general-purpose functionality. An example of a toolkit is the Java java.util package, or C++ Standard Template Library (STL) class library. Toolkits are used to develop application programs: they emphasize *code reuse*, just providing useful functionality, and their use do not impose a particular design on the application being developed.

Of course, toolkit design is harder than application design - their larger degree of reuse makes the design process more difficult. The designer of a toolkit does not know in advance the whole spectrum of its client applications - the more important is to avoid all assumptions and dependencies that can limit the toolkit's flexibility and applicability.

Frameworks [2] are sets of cooperating classes that make up a reusable design for software systems belonging to a specific application domain. Examples of application domains are: building graphical editors for different domains - artistic drawing, music composition, mechanical CAD, building compilers for different programming languages and target machines, and building financial modeling applications.

A framework is not a full application program. It provides the architecture of the application, defining its overall structure, its decomposition into classes and objects, the ways classes and objects collaborate, and the thread of control. The framework design decisions are common to its application domain, emphasizing *design reuse over code reuse* (as in the case of toolkits). Of course, a framework will usually include some concrete subclasses ready to work.

The designer of the framework takes into account all these design parameters, allowing the designer of the application program to concentrate on its specifics. The work of application designer is to write concrete subclasses (whose base classes are usually abstract classes defined by the framework).

The main principle behind frameworks is known as *inversion of control* (IoC, Figure 2, [4]). When the application programmer uses a toolkit (or a conventional subroutine library), he/she writes the main body of the application and calls the code he/she wants to reuse. This is the *direct flow of control*, calls being made from main program to toolkit/library. When the application programmer uses a framework, the main body is already part of the framework and he/she needs to write the code *called by the main body.* This is the *inverse flow of control*: calls are made from framework to concrete classes written by the application programmer. The number of design decisions left to application programmer is reduced, because all operations/classes to be implemented have particular names and calling conventions already specified in the framework.

The use of frameworks has significant benefits: increased productivity in developing application programs, and similar structure (architecture) of the software systems produced, enabling maintainability and consistency.

Of course, framework design is a hard task, harder than toolkit design. A framework needs to be both extensible and flexible, and its design must be based on a deep knowledge of the application domain. It is recommended to start the design of a framework after several successful attempts to develop domain-specific applications.

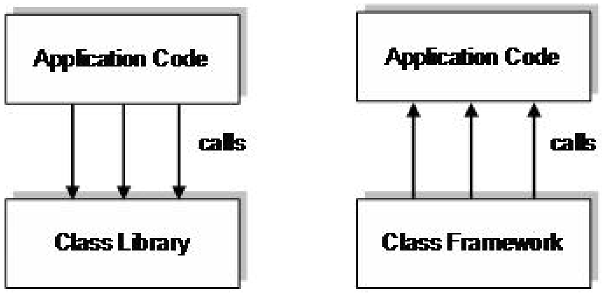

Figure 2: Inversion of control in frameworks

3. Software architecture and computing curricula

This section proposes a bottom-up approach of introducing software architecture disciplines into computer science curricula, part of undergraduate and graduate programs provided by our Department of Computer Science. The proposal starts with a design patterns course at undergraduate level, while software architecture and framework design disciplines will belong to graduate - master level programs.

3.1. Design patterns

The *Design patterns* course was introduced in our computer science curricula since 2003-2004 academic year, as an optional course for senior students in Computer Science. Its syllabus (see [10]) is based mainly on [3] book and covers five chapters: *Introduction* (definitions, templates, and a catalog for describing design patterns, design problems solved by using design patterns, how to choose and use design patterns in real situations), *Case study* (the description of a real application - a document editor - employing some design problems which can be solved by using design patterns: document structure and formatting, user interface, many look-and-feel standards and windowing systems, user operations and analytical processing), *Creational design patterns* (Factory Method, Abstract Factory, Builder, Prototype, Singleton), *Structural design patterns* (Adapter, Bridge, Composite, Decorator, Façade, Flyweight, and Proxy), and *Behavioral design patterns* (Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, and Visitor).

Each design pattern description follows the template: intent, the problem to be solved, the structure (including OMT/UML diagrams), examples (a structural and at least one real-life example). Lab activities include three mini-projects, dealing with creational, structural, and behavioral design patterns respectively. Each problem solution will be implemented in two different programming languages (Java, C#, Visual Basic 6.0, Visual Basic .NET, Python, etc).

3.2. Software architecture

The *Software architecture* course was introduced in the academic year 2004-2005, as an optional course for senior students in Computer Science program. Starting with AY 2008-2009, this course will belong to master program in Component-Based Programming.

The course (see [12]) introduces core concepts and principles of software architecture, focusing on software architecture definition, architectural styles and models, architecture definition languages, and tool support for architectural design. All theoretical concepts are introduced by using case studies, taken from real applications. The main chapters are: *Introduction*: from programming languages to software architecture, *Common architectural styles* (pipes and filters, data abstraction and object-oriented organization, event-based + implicit invocation, layered systems, repositories, table-driven interpreters, heterogeneous architectures), *Case studies* (discussion of some model problems and their solutions using different architectural styles: KWIC, Instrumentation software, Mobile robots, Cruise control, etc), *Shared information systems* (Database management systems, Software development environments, CAD in civil engineering), *Client/server architectures* (introduction and evolution, two-tier architectures, three-tier and n-tier architectures), *Architectural models and elements* (model elements and types, traditional architectural notations, architecture description languages, 4+1 view model - Kruchten, Zachman framework, data warehouse), *Architectural notations* (informal diagrams, module interconnection languages, SGL - Software Glue Language, ADL - Architecture Description Language, modeling and specification Languages - UML, ModeChart).

During lab activities, students are grouped in teams of 3 to 5 members. Each team will solve a model problem, and each team member will implement a different architectural style.

3.3. Framework design

The *Framework design* course is intended to be introduced starting with AY 2008-2009, as an optional course offered to computer science master programs provided by our Department of Computer Science. Its planned objectives (see [11]) are to explain the design techniques used to develop application frameworks, to discuss some existing lightweight frameworks as a novel approach for building enterprise applications, and to offer students the opportunity to design a new framework during didactical activities, using the specific tools for the development of lightweight frameworks dedicated to several application domais.

Its contents include: Basic GoF design patterns used in framework design, Eclipse plug-in development, An MDA approach for designing Web user interfaces, Developing a framework for Web User Interfaces, Designing a framework for Web Services, and Lightweight frameworks for applications with layered architectures.

4. Conclusions

Current computing curricula deals mainly with *programming-in-the-small* - related disciplines, like algorithms and data structures, programming languages, databases, operating systems, and so on.

Due to rapid evolution and growing importance of software architecture disciplines, we believe that computing curricula needs to be updated accordingly. Students need to be aware of the role of high-level design issues in software development, and they must have basic knowledge regarding important architectural concepts and principles developed in the last two decades. All these belong to what do we call programming-in-the-large, the old name for software architecture.

Our proposal includes three new disciplines to be added to Computer Science curricula. Two of them were already included in our undergraduate program, and the results encourage us to continue in this direction. We believe that the importance of software architecture will grow in the future, bringing into attention new disciplines.

References

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

[2] L. P. Deutsch, Design reuse and frameworks in the Smalltalk-80 system. In Ted J. Biggerstaff and Alan J. Perlis, editors.

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

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

[5] P. Kruchten, Architectural Blueprints - The “4+1“ View Model of Software Architecture, IEEE Software 12 (1995), No. 6, 42-50.

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

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

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

[9] IEEE. Software Engineering Body of Knowledge, 2004. http://www.swebok.org/ironman/pdf/SWEBOK_Guide_2004.pdf

[10] Design Patterns syllabus, Department of Computer Science, Babes-Bolyai University http://www.cs.ubbcluj.ro/files/curricula/2007/disc/rtf/emid0016.rtf

[11] Framework Design syllabus, Department of Computer Science, Babes-Bolyai University http://www.cs.ubbcluj.ro/files/curricula/2008/disc/rtf/emid1012.rtf

[12] Software Architecture syllabus, Department of Computer Science, Babes-Bolyai University

Author:

Bazil Pârv
Department of Computer Science
Faculty of Mathematics and Computer Science
Babes-Bolyai University
1, M. Kogãlniceanu, Cluj-Napoca 400084, Romania
email:*bparv@cs.ubbcluj.ro*