Chapter 26
A Heuristic Approach to Architectural Design of Software-Intensive Product Platforms

Carlos O. Morales

Abstract This chapter introduces a heuristic approach for the analysis, architecting, and design of software-centric product platforms. The central role of software architecture is stressed by highlighting its relationship to the analysis of new product domains. Several case studies are used to illustrate key concepts, including a more detailed case on the design of an object-oriented application framework as platform for a family of products that control industrial processing machines. Case studies and methodology are linked to important software engineering design principles. At the end of the detailed case study, an approximate measure of code reuse and its economic impact is presented, which can serve to support the business case of making the significant investment required by a software platform for a family of related products. This chapter builds on fundamental software engineering concepts introduced in Chap. 21.

26.1 Introduction

Each new generation of high-technology products is smarter and more sophisticated than the previous one, mainly as a result of their advanced software features. Voice recognition and synthesis, automated suggestions for new purchases, smart assistants that anticipate the user's intention, devices that learn the user's preferences and lifestyle, expert systems that can perform automated diagnoses and classifications, or predictions of the future cost of airline tickets and stock prices. All of these technologies are manifestations of the ever-increasing complexity of software.

Software is inherently complex, and its malleability coupled with the sheer number of degrees of freedom (DOF) of a software system (one DOF per line of code) makes it a fragile and fertile source of product defects. However, as software

C.O. Morales (✉)
Animas Corporation, Johnson & Johnson Medical Devices, 200 Lawrence Drive, West Chester, PA 19380, USA
e-mail: carlos.o.morales@ieee.org

engineering continues to mature as a discipline, new methodologies, tools, processes, and design approaches aid in the development of more solid and robust software technologies for this kind of products.

Mission-critical systems, in particular, require a high level of discipline and formality during their design, implementation, and testing. It is of the utmost importance to spend enough time analyzing and understanding the specific domain for which the new product is targeted. Examples of mission-critical systems include implantable medical devices, life-support equipment in hospitals, weapon systems, avionics, spacecraft, telecommunication satellites, nuclear reactor controllers, and automobile control computers, just to name a few. Nevertheless, the same process and concepts can also be applied to other products like cellular phones, industrial controllers, or consumer video equipment, where overall quality of the product is increasingly judged by the quality of its software content.

These sophisticated products must be built on a solid foundation if they are to perform safely and effectively, and the best way to accomplish this is to design and implement a software platform for a family of related products, which is known in software engineering as a domain-specific framework (Fayad and Schmidt 1997). One of the most notable qualities of these frameworks is that their robustness is improved every time this platform is reused to derive a new product, since the framework constitutes a reusable reference design and a very valuable, thoroughly tested, reusable code base.

Domain-specific application frameworks, or enterprise frameworks, are neither easy nor cheap to develop. The task requires a dedicated team of professional software engineers led by an architect with a deep understanding of the problem domain and with the experience of having previously developed several applications in that same domain. For more information on enterprise frameworks, their challenges, and economic justifications, the reader is referred to CACM (1997).

In this chapter, we use examples from several domains to illustrate the analysis and design of software systems that could serve as generalized solutions, or platforms, for families of related products.

26.2 Definitions of Framework in Software Engineering

The term "framework" is heavily used in software engineering, but it can be confusing sometimes. For clarity, we shall define the following terms: "enterprise architecture framework," "software infrastructure framework," and "family platform framework."

26.2.1 Industry-Standard Enterprise Architecture Framework

IEEE Standard 42010–2011, "Systems and Software Engineering: Architecture Description" (IEEE 2011) is now an international standard that has also been adopted by ISO and IEC. This document specifies *the manner in which architecture* *descriptions of systems are organized and expressed.* This includes specifications for *architecture viewpoints, architecture frameworks, and architecture description languages for use in architecture descriptions.* This standard defines the term "architecture framework" as *conventions, principles, and practices for the description of architectures established within a specific domain of application and/or community of stakeholders.*

One of the earliest publications on the topic of enterprise architecture frameworks was authored by John Zachman, of IBM (Zachman 1987). The Zachman Framework continues to be in use today, and it paved the way for the creation of architecture frameworks for many domains, most notably for the defense industry, e.g., DoDAF, the US Department of Defense Architecture Framework standard on how to document architectures. Further discussion on architecture frameworks is beyond the scope of this chapter, and the interested reader is referred to Clemens et al. (2011) for an accessible overview of DoDAF and other architecture frameworks. For a survey of all known architecture frameworks currently in use, please see ISO (2011).

The main idea to remember here is that industry-standard architecture frameworks are not executable code and refer mainly to documentation requirements.

26.2.2 Software Infrastructure Framework

Examples of these frameworks include Microsoft Foundation Classes (MFC®), Microsoft NET Framework®, Java EE®, and frameworks for creating graphical user interfaces (GUI toolkits) such as Qt, Motif, Swing®, or Adobe Flash®, just to name a few. This class of frameworks has been called by many names, including *application frameworks and architectural frameworks.* These frameworks are collections of software libraries that provide infrastructure services for other applications to use in the form of software objects. Some notable classes provided by these frameworks are GUI widgets, networking services, web services, e-mail, and other messaging services.

These frameworks provide executable code that is intended for general purpose, and they can be used to build sophisticated software applications by using its high-level functionality. However, they do not solve problems that pertain to a particular domain or business, which is the next level up in the software abstraction scale, as described below. A software infrastructure framework can be thought of as a general-purpose toolbox.

26.2.3 Family Platform Framework

In this work, we refer to enterprise application framework as a former name for family platform framework, since the term helps to make the connection with the literature. However, we prefer to use the term family platform framework because it is more accurate and descriptive regarding its intention and use.

Enterprise application frameworks, or family platform frameworks, implement solutions for a specific domain. This kind of framework is a reusable, semi-complete application that can be specialized to produce custom applications. They are designed for particular businesses such as data processing, telecommunications, or other industrial domains. A platform framework reuses not only code but design as well. It describes how the system is decomposed into cooperating objects and how custom applications must be implemented based on this infrastructure. Platform frameworks are the software foundation for families of related products, i.e., they serve as software product family platforms.

A good introduction to enterprise application frameworks can be found in the literature (CACM 1997). For a deeper study of all that entails to embark in their design and development, see the referenced works by Fayad et al. (1997, 1999, 2000), Schmidt and Fayad (1997), Schmidt et al. (2000), and Johnson (1997).

The term "family platform framework" specifically refers to an object-oriented enterprise application framework that serves as the foundation for a family of related products and that has been developed according to the specific software architecture and design process described herein.

Later, a detailed case study is introduced to illustrate the design and implementation of a software platform for a family of industrial machines and the economic impact of creating software family platforms. The scope of this chapter is to present the thought process for analyzing a newly planned product family and for mapping the results of that analysis to the family architecture and its detailed software design.

26.3 General Architecture of Software-Intensive Products

This chapter uses, and builds on, the concepts presented in Chap. 21 on software design principles (Morales 2013). It is recommended to read that chapter first, if the reader is unfamiliar with some of the terms used below.

Most modern software-intensive products are embedded devices, but not necessarily. As shown in the following examples, these products can be found hidden in vehicles, in the form of high-volume consumer devices, high-cost customizable industrial equipment, or just as pure software residing on remote servers providing service to clients over the Internet (also known as "the cloud").

The best way to tame the complexity of most software-intensive modern products is with the application of the age-old principle of *Divide and Conquer*. Figure 26.1 shows a typical layered architecture for software applications, based on the *LAYERS* architectural pattern (Buschmann et al. 1996), which is widely used in industry and even standardized for some applications like networking. The main advantage of this architectural pattern is that each layer specializes in a particular aspect of the application. Each layer is highly cohesive and is loosely coupled with its adjacent layers. Typically, dependency among layers only flows in one direction,

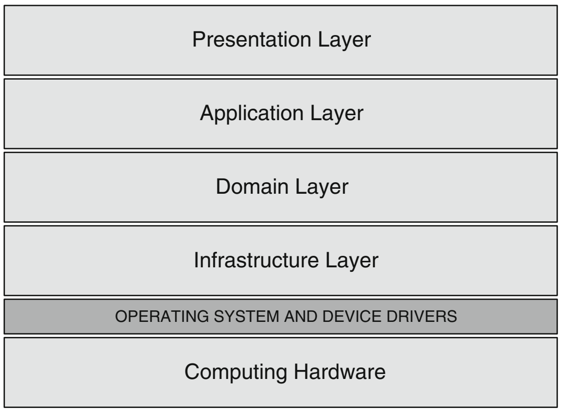

Fig. 26.1 Layered architecture

i.e., downwards. This means that upper layers can typically initiate interaction with the lower layer at any time by requesting services through its interface.

In the classical structure of a layered architecture, the Infrastructure software layer provides generic technical services that enable access to the system's resources, while the Domain layer encapsulates the business-specific concepts and rules. The Application layer implements particular jobs that the software is intended to do, but it does not include any business knowledge. It accomplishes its task by delegating processing and coordinating cooperation between objects in the Domain layer. The Presentation layer is typically the user interface.

Let us use an example to illustrate how this structure is applied. In an online banking application, the Presentation layer would be implemented within the customer's web browser, through which she navigates the system's functionality and requests transactions. The Infrastructure layer would provide user authentication services, access to the bank's databases, and encryption for a secure connection between the bank and the customer, among many other facilities.

In the case of a fund transfer operation, for instance, the customer would enter information through the Presentation layer (GUI), which in turn would send it down to the Application layer. The Application layer is responsible for validating that input and then sending it down to the Domain layer for processing. Data sent would include the amount to be transferred, currency type, source account number, and destination account number, and that is the end of its responsibility. The Domain layer implements the essential business rules for the banking business, like accounting, for example. In accounting, one of the fundamental business rules is "Every credit must have a matching debit"; thus, it performs the operation by modifying all the necessary tables in the bank's database through requests to the Infrastructure layer, which provides these services. As each operation in the lower layers is completed, the upper layers are notified, until the end result is finally presented back to the customer at the top layer.

When we embark on the task of designing a platform for a product family, maximizing software reuse is one of the main objectives. Although designing and

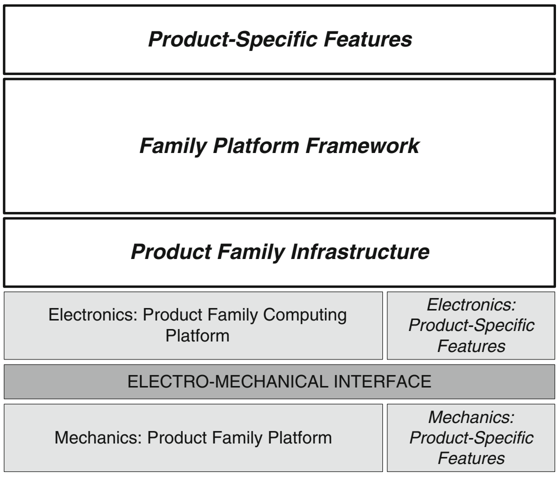
Product-Specific Features
Family Platform Framework
Product Family Infrastructure
Electronics: Product Family Computing Platform
Electronics: Product-Specific Features
ELECTRO-MECHANICAL INTERFACE
Mechanics: Product Family Platform
Mechanics: Product-Specific Features

Fig. 26.2 General architecture for a software-intensive product platform

implementing a platform take time, it later pays off significantly. Maximizing code reuse ensures that once the platform is finished, new products can be implemented in a very short time, and it allows engineers to focus on what makes the new product unique, i.e., they don't have to reinvent everything with each new project.

In order to maximize code reuse, we design and implement an object-oriented application framework to work as the Domain layer, which here we call the *family platform framework*. We call it by a different name because it not only implements the core functionality that comprises the essence of the product family as well as its overall design philosophy but, at the same time, it imposes certain rules for the implementation of features that differentiate individual members of the family, both in software at the top layer and in hardware at the bottom layer. At the top, we take the layered architecture described above and compress the Application and Presentation layers into one, which we call the *product-specific features*. Individual product-specific features could be optionally included or excluded in order to create different products within the family, even though their structure and basic behavior are dictated by the platform framework architecture.

The advantages of merging the two top layers into one will become clear with some examples, as described later, but the main reason is that the framework takes control over most things in the product family, and user interfaces (Presentation) become just one more product-specific feature that must comply with the interfaces prescribed by the platform.

At the bottom of the software layer hierarchy we have the *Product Family Infrastructure* layer, which is slightly different from the classical banking example presented above. In the case of a product family platform, the Infrastructure layer must not be static, nor monolithic, but modular and extensible to accommodate the evolving hardware needs of individual products within the family. This is illustrated with the partitioned hardware layers shown in Fig. 26.2, where we can appreciate a standardized hardware platform for the family, plus additional modules that are product specific.

This highly cohesive domain encapsulation centered on the family platform framework brings along a technical risk. If the platform is not designed correctly, then the number of products that can be easily derived from the platform would not be as prolific as expected. Investment on a product platform is justified only when it has a long life span, consistently generating a long sequence of derived products. Two key principles that will help us achieve a correct design of the platform are *Abstraction* and *Design for Change* (Morales 2013).

Figure 26.2 shows a generalized layered architecture for a software-intensive product platform, which is based on the same concepts of the *LAYERS* architectural pattern, but tailored to suit the specific needs of a software platform for a family of related products. This figure describes a platform for embedded devices more accurately, although it also applies to software-only products if we remove the bottom layer representing the product's mechanisms. The key concept here is *abstraction*, clearly *separating concerns* into specific containers that segregate technologies and encapsulate those features that belong to the common product platform and those that distinguish individual products in the family.

Each layer is abstracted away from the adjacent layers by means of their interfaces. The framework exposes an Application Programming Interface (API), which prescribes the behavior and data expected from the software modules that comprise the Product-Specific Features' layer. The electromechanical interface specifies how the electronics and mechanical subsystems fit within the system in order to interact with the external world. Compared with the reference *LAYERS* architectural pattern, the operating system and device drivers' layer, which isolates the software from its computing platform, have been merged into the family platform framework.

In order to design an effective and efficient product family platform, we must make design choices related to the computing platform that will ensure software compatibility across all derived products in the family and maximize reuse. Specifically, the platform specification should prescribe a particular processor core for the whole family, e.g., an ARM Cortex or an Intel Atom, or some other specific processor. However, this choice does not prevent product designers from having a rich assortment of peripherals around the processor core that can be very different from product to product within the same family—as long as they keep the same core. Furthermore, choosing a single operating system for the whole family makes software reuse even easier and more cost-effective.

A successful family platform framework typically implements approximately 80 % of the functionality for each product derived from the family, and it prescribes with precision how the remaining 20 % (product-specific features) are to be designed and implemented. Although it sounds restrictive, it is in fact very effective and efficient, since the enforcement of the family architecture and its behavior encapsulated in the platform forces and enables software engineers to follow a clear and consistent process for the implementation of derivative products based on the platform. This phenomenon will be explained in more detail later in this chapter.

26.4 Domain Analysis

Deep knowledge about a product domain doesn't come easy. The best situation is to have previous experience designing stand-alone products for that domain before one commits to designing a successful product family platform, or at least, to have a multidisciplinary team of product designers and consultants with significant experience in that particular domain. The reason for this concern is related to the technical risk mentioned above: if the product family framework is not designed correctly, then the exercise will result in a significant expense that does not meet expectations. This is not a typical software project and has many subtleties in designing and building object-oriented enterprise application frameworks (Fayad and Johnson 1999; Fayad and Schmidt 1997).

Figure 26.3 shows a UML activity diagram that describes the high-level process for synthesizing the core essence of a product domain and incorporating that knowledge into a new product family platform. Each step is described below.

26.4.1 Develop Product Platform Use Cases

When designing a platform for a new product family, the first thing we need to know is a clear definition of its scope. It is necessary to specify the behavior and functionality that these new products must exhibit in order to satisfy the needs of all stakeholders.

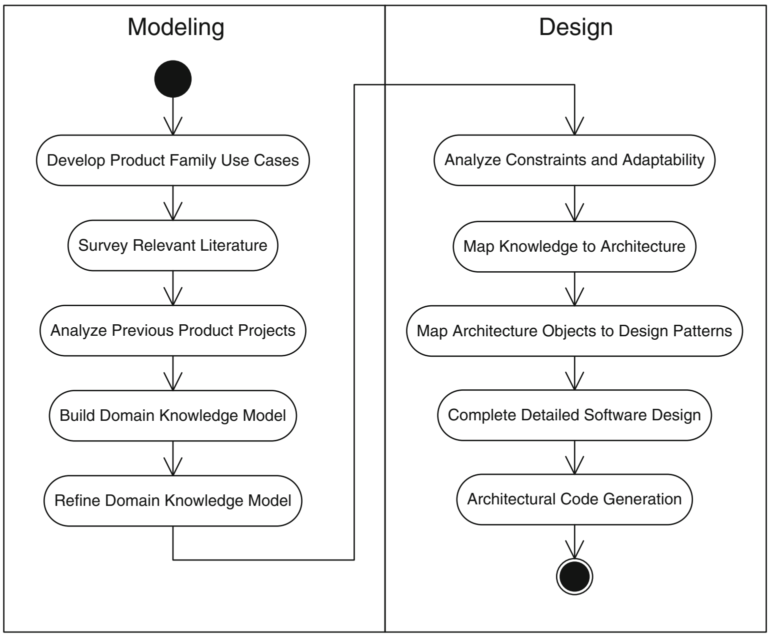
Modeling
Design
Develop Product Family Use Cases
Survey Relevant Literature
Analyze Previous Product Projects
Build Domain Knowledge Model
Refine Domain Knowledge Model
Analyze Constraints and Adaptability
Map Knowledge to Architecture
Map Architecture Objects to Design Patterns
Complete Detailed Software Design
Architectural Code Generation

Fig. 26.3 High-level process for synthesizing a product family platform

One of the best ways to obtain this understanding is to analyze the expected use of the product family through use cases (Jacobson et al. 1992).

The scope of the use cases must remain at a high level of abstraction, where the goal is to obtain all the scenarios in which each type of stakeholder will interact with a generic product of this new family in order to satisfy their goals. Some specific needs may be satisfied by one product derived from the platform, but at this point, the analysis should be kept at the product platform level. Our goal at this stage is to abstract the essential use cases that make up the core behavior of the product family. An excellent manual for analyzing use cases is Cockburn (2000).

26.4.2 Survey Relevant Literature

This activity is particularly important (and indispensable) for software designers that are newcomers to the particular domain of interest. Collect all relevant literature about the domain, which includes books, journal publications, presentations, technical descriptions, and specification documents for similar products that can be used as precedents. Reliability of all sources should be confirmed, ensuring that they represent the consensus of subject matter experts. Extract and abstract the essential knowledge about the domain, summarizing the essential concepts in tables or some other tool.

26.4.3 Analyze Previous Product Projects

Fred Brooks said that, when designing a new kind of system, software teams *will* throw one system away whether they want it or not (Brooks 1975). His argument was in favor of using pilot projects to ensure that we can ultimately deliver exactly the software system that we all want. There is certainly great value in having the opportunity to perform forensic analysis of previous projects that have attempted to solve the same problem that we are now facing, since this improves the odds that the team will arrive at a better design the next time. However, our focus here is not on the software design or the code. What we are looking for are abstract concepts: those ideas that represent the essence of the kind of products for which we want to design a platform and those characteristics, functions, and behavior that comprise the essence of being a member of that family of products. For example, we want to discover those essential attributes and behavior that constitutes being a smartphone, or a pacemaker, or a human-size robot arm.

Essentially, this is still part of the information survey that began with the relevant literature review, but now our sources are more heterogeneous and focused on the actual product needs. We have to continue extracting and abstracting important knowledge about the domain from these new sources and add the most relevant concepts and relations to the compilation we started with the literature review.

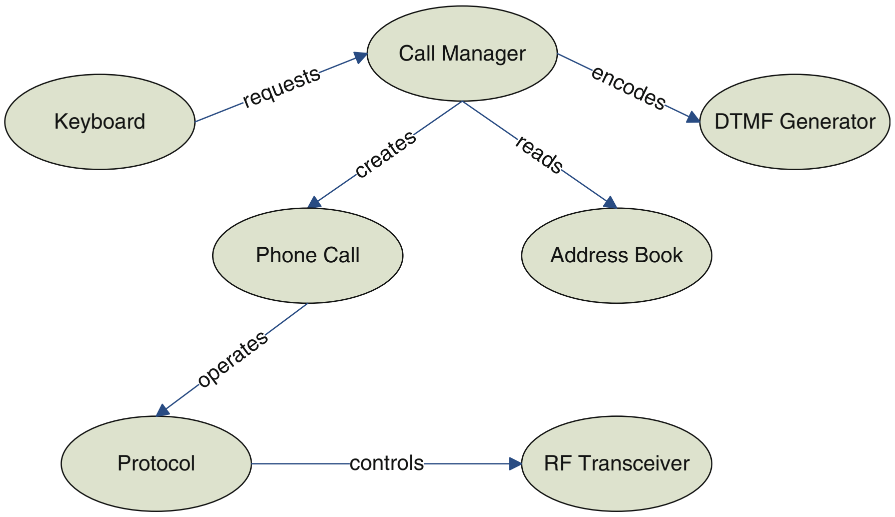

Fig. 26.4 Knowledge graph partially representing the mobile phone domain

26.4.4 Build Top-Level Domain Knowledge Model

Using the knowledge derived from the literature and product information surveys, we now build a knowledge graph (Fayad et al. 1999). Knowledge graphs are a form of the more widely known Semantic Networks (Russell and Norvig 2009).

A knowledge graph is made up of nodes and edges. Nodes are those concepts that appear consistently and in similar ways, in the domain knowledge survey. These essential domain concepts are typically nouns, and each node represents an indispensable concept for the product platform. Edges represent relations among the different concepts, or nodes, and they are typically represented with verbs indicating actions that one node performs on other nodes.

As an example, Fig. 26.4 shows a knowledge graph that abstracts and represents some of the concepts and relations that could be used to design a generic product in the mobile phone domain. Notice that the nodes represent concepts that are indispensable for the product to be considered a mobile phone. In other words, they constitute the essence of the product family.

The key design principles here are *Separation of Concerns, Abstraction, and Generality* (Morales 2013).

26.4.5 Refine Domain Knowledge Model

Once we have collected the main concepts and relationships in the top-level domain model, it is necessary to investigate the internal structure of each node in order to gain a deeper understanding of the platform design needs. This refinement of the

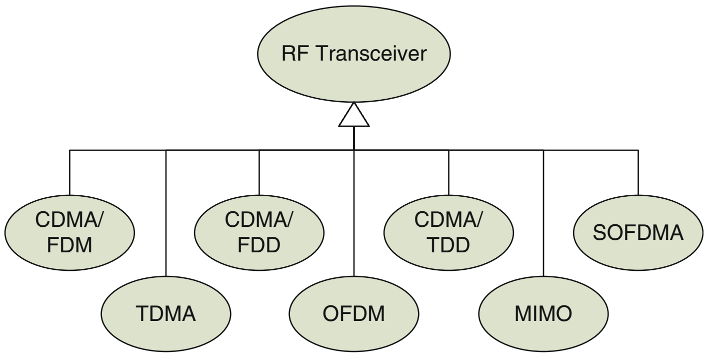

Fig. 26.5 Refined knowledge graph for the RF Transceiver node in the mobile phone domain

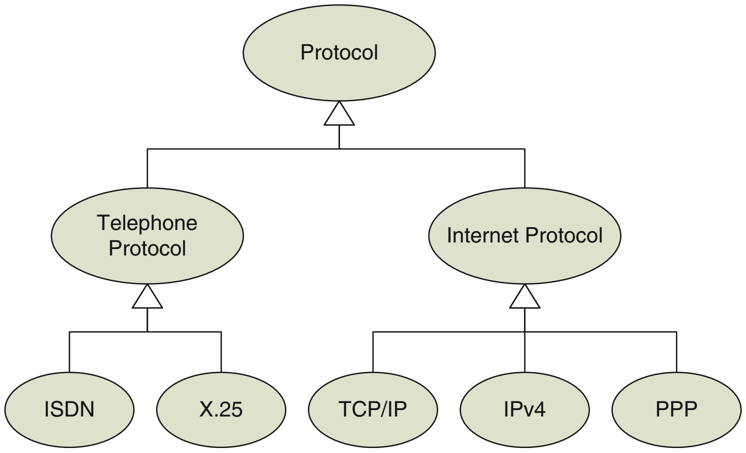

Fig. 26.6 Refined knowledge graph for the Protocol node in the mobile phone domain

domain knowledge model reveals more information that is less abstract and more suitable for use in the design of the product platform.

Figures 26.5 and 26.6 show the concept hierarchies for the RF Transceiver and Protocol nodes. The white triangle denotes specialization of the root concept, similar to inheritance in object-oriented software. The root node can be thought of as similar to an abstract class and the leaves as similar to concrete derived classes.

26.4.6 Analyze Constraints and Adaptability

This is a critical step in the process of abstracting knowledge about a domain and using that knowledge to design a successful product family platform that can evolve over time and be useful to generate a prolific family of derived products.

The key design principle here is *Design for Change* and its various forms of expression: *Change of Algorithms, Change of Data Representation, Change of Abstract Machine, Change of Peripheral Devices,* and *Change of Social Environment* (Morales 2013).

Each node in the knowledge graph must be analyzed for its potential need of change in the future due to foreseeable technological progress or to enable different features for distinct members of the product family. All anticipated changes can be compiled in tables, supported by reference documents.

For example, let us continue using the mobile phone example, and oversimplifying for the sake of clarity, we say that it could be anticipated that the "RF Transceiver" node will change. This node represents the radio transmitter and receiver used by the mobile phone to connect to Base Stations, and it could use any of two multiplexing technologies, namely, code division multiple access (CDMA) or time division multiple access (TDMA), depending on the target market of the particular product. This is an indicator of the need for *Modularity* in the implementation of this feature in the product platform. Compliance with each of these standard technologies is also a constraint on the system.

Similarly, and oversimplifying again, we could see that the "Network Protocol" node should be easily adaptable to work with potentially non-compatible new technologies. Experience shows that mobile network technology evolves very quickly, and thus, for the investment on a long-life span product platform to be worthwhile, it should be able to adapt to these new technologies as they come along. In this manner, we have seen three generations of mobile networks based on the Global System for Mobile Communications (GSM), including general packet radio service (GPRS) for second-generation networks (2G), Universal Mobile Telecommunications System (UMTS) for third-generation networks (3G), or Long-Term Evolution for fourth-generation networks (4G LTE). These are examples of *Constraints* on the platform design and *Change of Peripheral Devices.*

Other concepts are not as clear-cut, however. For example, a keyboard could have different embodiments in two products derived from the same platform. Let's say that one product could use a membrane keyboard driving the electronics directly, and another could implement it as a soft keyboard on a touch screen, driving a software device driver instead (*Change of Peripherals* or *Change of Data Representation*). However, at the product platform level, the abstract concept of a keyboard is exactly the same. Therefore, the actual implementation of the keyboard in a particular product is a secondary matter. Along the same lines, the abstract concept of an address book is exactly the same at the platform level, even though it might be implemented as a remote web page that is accessed over the Internet or as a local database file, which, as described above, would constitute secondary concepts and not essential details for the platform.

The dual-tone multi-frequency (DTMF) generator can be implemented in one way only, since it is a real function based on an international industry standard with which every phone, regardless of its technology, must comply. This is a *Constraint* on the system design. In contrast, the concept of a "Call" is completely abstract, completely defined internally, and its representation could be unique to the product platform.

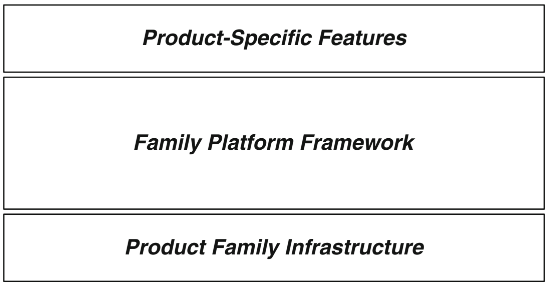
Product-Specific Features
Family Platform Framework
Product Family Infrastructure

Fig. 26.7 Layered software architecture for a product family platform

Although the user interface is not shown in the knowledge graph, we can say that the product platform software should be designed to represent all the concepts of a user interface in an abstract form within the platform, e.g., using label IDs instead of actual text strings, and anticipate a *Change of Social Environment* by designing the platform is such a way that different languages can be easily implemented through loosely coupled software components that are external to the platform.

26.4.7 Map Knowledge to Architecture

With a greater understanding of each node and its internal structure in the domain knowledge graph, we now proceed to find the right place for each node in the software architecture. Figure 26.7 shows the software section of the generalized architecture presented earlier. Assignment of each node to the appropriate software layer will ensure the construction of layers that have the very important properties of *high cohesion* within them and *low coupling* between them (Morales 2013).

In general, the Product Family Infrastructure layer should provide access to the physical resources of the system, the Family Platform Framework layer should encapsulate all the system behavior and functionality that does not change from product to product, and the top layer should encapsulate those features that distinguish each product from the rest of the family.

In the mobile phone example, it is clear that abstract keyboard and RF Transceiver refer to physical resources and, therefore, belong in the Infrastructure layer. The call manager, phone call objects, Internet session objects, protocol handlers, address book, and DTMF generator all belong specifically to the mobile phone domain, i.e., in the family platform framework. Note that none of the nodes in the top-level knowledge graph is actually mapped to the Product-Specific Features layer directly, but as revealed by the Constraints and Adaptability analysis above, some of the nodes have deeper hierarchies where some of the sub-nodes are expected to change over time or change from product to product in the family. In other words, some of the sub-nodes should be allocated to the top layer where they function as product differentiators. Those sub-nodes that are not expected to change, regardless of the particular product instance, should be allocated to the family platform framework to be encapsulated. The best way to produce a robust arrangement of software objects is to use a catalog of best design practices and time-proven solutions known as design patterns (Gamma et al. 1995).

26.4.8 Map Architecture Objects to Design Patterns

The Constraints and Adaptability analysis guides the selection of the best design pattern for each case. For example, some of the classic design patterns from Gamma et al. (1995) are the following: "Bridge," which decouples an abstraction from its implementation so that both can vary independently; "Proxy," which provides a surrogate for another object in order to control access to it; and "Strategy," which defines the interface for a family of algorithms, encapsulating them and making them interchangeable. After the landmark work of Gamma, Helm, Johnson, and Vlissides, many other books on design patterns have been published. Some of them are new catalogs of design solutions for software in general, and some others are aimed at particular domains (e.g., Johnson 1992; Buschmann et al. 1996; Schmidt et al. 2000; Fowler 2002; Douglass 2002, 2011; Alur et al. 2003; Daigneau 2011).

As an example, let us assume that our family of smartphones has a built-in image enhancement tool designed to improve the quality of images taken with the phone's built-in camera. A digital image is represented in software as a matrix of pixels with color values. Users can perform automatic or manual enhancements on a single image by running image filters to enhance sharpness, contrast, modify color saturation, or exposure (brightness). Since this is a feature that all products in the family will have, these image-enhancing mechanisms will be allocated to the Family Platform Framework layer. All image filters operate as convolutions on the original image and produce a new image as output, with modified values on each pixel according to the requested operation. Figure 26.8 shows a UML class diagram that implements a family of image processing algorithms according to the "Strategy" design pattern (Gamma et al. 1995).

26.4.9 Complete Detailed Software Model Design

Applying design patterns is only the beginning of detailed software design. Design patterns are coarse-grain building blocks, but each software system has nuances and peculiar concepts that need to be represented in detail, as well as the interconnection among design pattern structures.

The ever-increasing complexity of each new generation of high-technology products requires the use of appropriate tools to handle it and enable designers to

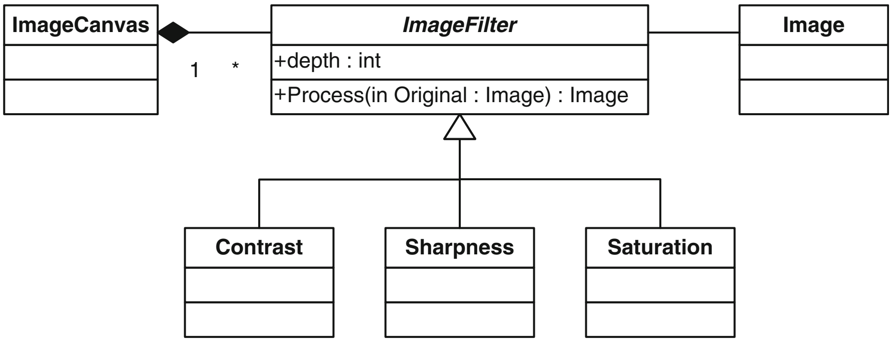

Fig. 26.8 An image processing feature for a family of mobile smartphones based on the Strategy design pattern

look at the system from different perspectives. Abstraction is the strongest feature of model-based development, also known as Model-Driven Architecture (MDA). By slicing the system into distinct perspectives that show its structure, dynamic behavior, interfaces, and internal states, designers can develop a complex system that maintains consistency and correctness throughout the product development life cycle. The Unified Modeling Language, or UML, was specifically developed for this purpose. For more detailed presentations, the reader is referred to the following: Fowler (2003), Booch et al. (1998), Rumbaugh et al. (1998), and OMG UML (2011).

26.4.10 Architectural Code Generation

An additional benefit of modern software modeling tools based on UML is that most of them provide code generation facilities, as well as two-way code engineering. We use the term architectural code generation to refer to the automatic generation of source code in an industry-standard programming language like C++, Java, C#, or any other. This automatically generated code typically includes all software interfaces defined as properties and methods of a class, as well as processor and project-specific header files. Since the Object Management Group (OMG) released the new specification for UML 2.0, executable models are now possible, and code generated automatically also includes source code that implements state machine behavior directly from UML State Diagrams (also known as Statecharts).

Two-way code engineering tools allow software developers to maintain the software model and source code synchronized by having the tool automatically update the code every time the model is changed. Likewise, it can also update the model whenever the source code is modified, e.g., when a function parameter changes name or data type.

26.5 Case Study: Software Platform for a Family of Industrial Machines

26.5.1 System Overview

We now present a more detailed case study from an actual project to illustrate the design of a software platform for a product family. Here, we review the characteristics of the system and the design decisions that made it a successful platform. The purpose of this system was to serve as a generalized solution that addresses the specific requirements of a family of products in the industrial automation and test domain.

The main purpose of this platform is to provide the software foundation for a family of automated machines that are used in industrial manufacturing and test. Typical applications of this product family are stand-alone units, or cells, that are integrated into fully automated or semiautomated manufacturing lines. Many of these cells include one or more robots for product handling, machine vision systems for robot guidance and automated visual inspection, general-purpose digital and analog inputs and outputs and programmable power supplies, waveform generators, current and voltage meters, and other instrumentation equipment. The same robots, digital cameras, power supplies, and meters can be used across many different projects, for example, assembling handheld blood glucose meters, testing miniature endoscopy equipment, calibrating vibrating mirrors, or simultaneously performing functional tests on a batch of two hundred computer hard drives. Below we take a closer look at some of the most important aspects of this design.

26.5.1.1 High-Level Organization

A major design goal for this system was to serve as a solid, reusable product platform, applicable in a variety of automated test and industrial automation solutions. In general, its implementation is extendable through external and interchangeable software components.

Software components conforming to the interface specified by this framework will, as a consequence, be reusable components as well, which may be stored in a company repository for use in future projects. This component-based approach to software development enables consistent reuse of the components described below.

PATF Engine

PATF stands for production automation and test framework. The PATF engine component encapsulates the general operations and algorithms required by most applications in this domain, including but not limited to sequencing of operations, manage requests to and from external devices, manage internal data representation, system configuration, basic interaction with the user, interaction with external software components representing devices under test (DUT), and basic database management using a default format that may be overridden or translated by an external software component (see below). The engine provides two main classes: first, we have the Scheduler class, a *singleton* (Gamma et al. 1995) responsible for assembling the software system at run-time and for orchestrating overall execution. Then, we have the TaskProcessor class, implemented according to the *Command Processor* design pattern (Buschmann et al. 1996). This class provides an arbitrary number of TaskProcessor instances that are responsible for executing the procedures prescribed within AppSequence classes.

ActiveDevices

These software components are more than just device drivers. They not only implement communication and control over ActiveDevices, but they also include graphical user interfaces and persistent configuration facilities for each ActiveDevice. This set of software components becomes part of a common library of extendable components for use in future projects. These components may be initially built by implementing minimum functionality and then incrementally extend their services, while complying with the ActiveDevice interface as specified by the platform.

Independent ancillary software components describing the current application-specific details include:

1. Application sequence component. Library file that contains code for the application-specific functions or sequence of operations. This component encompasses the main differences that distinguish each product in the family derived from this platform.
2. System configuration Component. Executable file responsible for launching the PATF engine and customizing it for a particular application.
3. Application-Specific GUI Component. Required software component that defines application-specific GUI labels and panels.
4. DutCollection. Optional software component that implements application-specific computation algorithms for devices under test (DUT). Examples are DUT product-specific communications protocols, decision-making algorithms, parameter limits testing, and data-exchange protocol translators.
5. External DB controller. Optional software component that implements interfaces for a specific database, for example, interfacing with remote database servers, generating files that are compatible with other applications like Microsoft Excel® or Word® processors, and exchanging data with other applications.

Figure 26.9 is a UML component diagram that shows the main software components of the PATF platform and their mapping to the corresponding architectural layers of the framework.

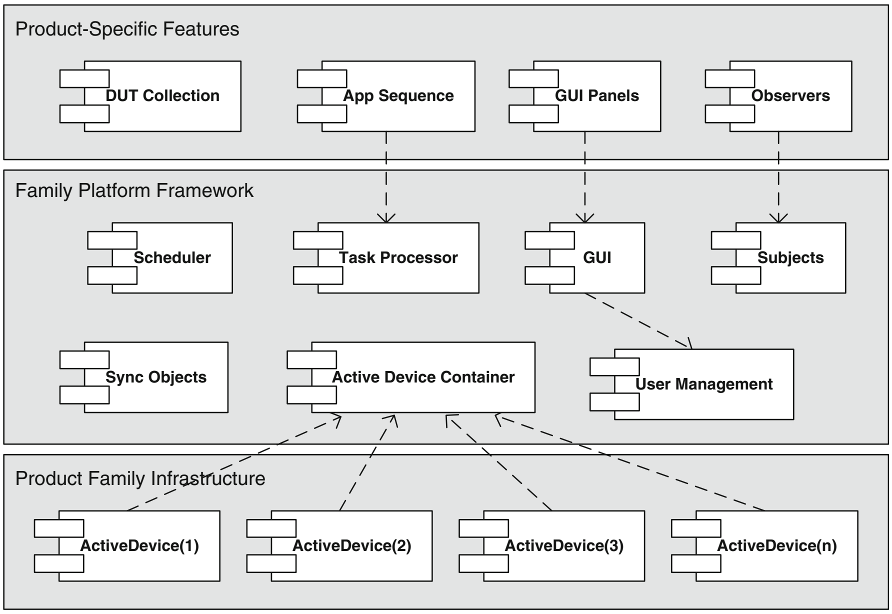

Fig. 26.9 Software components and architecture of the PATF product family platform

26.5.1.2 High-Level System Behavior

The proposed domain *abstraction* classifies operation modes of all automated industrial machines as being either manual mode or automatic mode. The system's behavior is driven by a finite state machine, and under these two operation modes, there are six top-level system states. Figure 26.10 shows these states and the events that cause transitions between them.

Manual Mode

This mode is used for initializing the machine, modifying operational parameters, and for performing manually controlled operations like jogging a robot, for example. Safety devices are of utmost importance in industrial automation machines, as they play a critical role in preventing operators from getting injured by the machine. In manual mode, however, these safety devices are overridden to enable a qualified technician to perform certain operations that require full control over the machine, as is the case of machine troubleshooting or other maintenance operations. This operation mode is hard coded into the system's engine, but before entering this mode, users are authenticated and only allowed in if they carry the

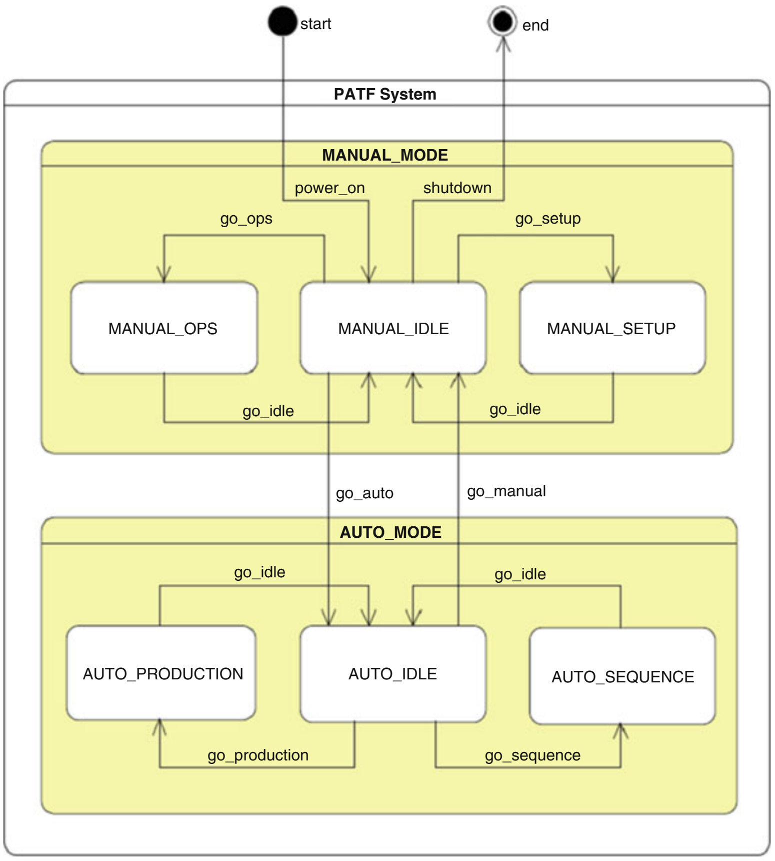

Fig. 26.10 PATF system state diagram

proper credentials, as determined by the User Management component. All safety device signals are ignored and the machine is allowed to run, although displaying a warning message on the user interface through the main GUI component.

Automatic Mode

This is the normal operation mode of this type of machines. In contrast to manual mode, all system devices and subsystems are engaged, and a built-in safety monitoring system watches for incoming alarms from input/output signals configured as safety devices and automatically calls an emergency shutdown procedure when flagged. Automatic mode also allows users to perform maintenance operations where full speed and safety device enforcement are required, for example, robot coordinate system calibration or product load and unload.

26.5.2 Relevant Design Decisions in PATF

Let us review the most important design decisions that were put into this framework. A significant feature is that this software application gets assembled at run-time as opposed to compile time. This is possible due to a clear *Separation of Concerns* and *Modularity*. As shown in Fig. 26.9, software components are compiled separately into multiple executables, not as a monolithic application. At the binary level, they are completely decoupled and independent, establishing their relationships and dependencies during execution only. This means that each software component in the Product Family Infrastructure and Family Platform Framework layers is always reused in binary form, not linked as source code libraries at compile time.

Another aspect of significance is that the architecture implements an *Abstraction* of the domain in the shape of behavior, as shown in Fig. 26.10, imposing *Generality* in the behavior of all derived products while still allowing limited customization through the use of externally defined configuration algorithms for each state and substate, which will be explained in more detail later.

The most relevant features, however, revolve around the principle of *Design for Change*, which enables the construction of a wide variety of products. Although they all belong to the same family and share similar characteristics, they can also be applied to a wide range of dissimilar applications across many industries. A more detailed presentation of these features follows below.

26.5.2.1 Design for Change

For the purpose of illustrating the application of the principle of *Design for Change* (Morales 2013), we now discuss some of the *Evolvability*, *Reusability*, and *Anticipation of Change* features that went into the design of the PATF software product platform.

Product Family Infrastructure Layer

Among several needed flexibility features that were observed during the Constraints and Adaptability analysis performed for this project, it was noted that most industrial automated cells would use robots, machine vision, and instrumentation equipment in a consistent manner. The differences between applications were limited to the specific settings and configuration for each infrastructure device, and the sequence of actions that were needed to assemble one product, or test some other. For that reason, all of these components (robots, cameras, instruments, etc.) were packaged as black-box reusable components called ActiveDevices. ActiveDevices were designed to be implemented in two parts: an ActiveX control and an ActiveX EXE (Microsoft 1994). Both ActiveX components are reused in binary form regardless of the application in which they are used.

Configuration for each ActiveDevice is performed by reading an INI text file containing parameter values for a particular application. When software engineers produce a new application based on the PATF platform, they configure their Infrastructure layer by executing a simple operation of "drag and drop" of ActiveX controls onto the "ActiveDevice Container," as illustrated in Fig. 26.14, later in this chapter.

As will be seen later on, this approach allowed code reuse of the Product Family Infrastructure layer to reach 100 % across all products in the family.

Family Platform Framework Layer

It was also observed during the Constraints and Adaptability analysis that a single graphical user interface based on the state machine of Fig. 26.10 could satisfy the needs of all applications in the industrial automation domain, as long as it provided a way to include general-purpose panels in which derived software products could display customized information based on their particular application. This design problem required the provision of a mechanism that could insert customized panels into the Main GUI at run-time.

The solution was to implement the GUI completely as part of the platform framework so it would be reusable in binary form (black box) and add a separate product-specific feature called "GUI panels." Reusing the GUI along with the system state machine and its related User Management component with its own database represented a major contribution to the overall code reuse ratio, which will be elaborated on later in this chapter.

The family platform framework also supplies generic "Subject" objects which can be instantiated from a GUI panel to work together with an "Observer" object. Observers are described in the next section.

As mentioned previously, the main difference among members of the product family is the sequence of operations they perform on their physical resources to assemble products using a robot and machine vision or the sequence of events when generating electrical stimuli to their devices under test (DUT) on which they perform automated measurements and apply pass/fail criteria. Therefore, a fundamental flexibility need for the platform was the ability to execute one or more simultaneous tasks in a multithreaded environment, effectively implementing an explicit *Separation of Concerns* (Morales 2013). This was achieved by having the platform framework provide generic "Task Processor" objects that would take algorithms encapsulated in a generic object whose class was defined externally, i.e., as a product-specific feature. Thus, a sample application would define a sequence of events for moving a robot about its work cell, operating its gripper, and controlling its speed and joint position, all in a single class of type AppSequence (see below).

Multiple Task Processors can run simultaneously either independently or collaborating through Sync objects (also provided by the platform framework) which enable processors to exchange messages and data, as well as performing a rendezvous that may lead them to alternate paths of execution based on their data exchange, e.g., a test station sending Pass/Fail results to the robot handling the product, which can then drop the product off into a reject bin or pack it into a shipping box, for instance.

Product-Specific Features Layer

The top layer of the product family architecture houses those features that make individual members of the family truly distinct from each other. The platform framework imposes what is known as *inversion of control* (Morales 2013). This means that all software components in the top layer must comply with the interfaces, protocols, and types expected by the platform framework underneath it. For that reason, this layer consists mostly of code templates. In other words, code reuse in the Product-Specific Features layer is white box. Form is imposed by the software platform, but content is completely determined by the application at hand, i.e., the specific product being implemented.

Whereas Task Processors are generic objects that will execute any task encapsulated in an AppSequence object, the AppSequence class is the product-specific complement to Task Processors. The software family platform specification prescribes some special requirements for the content of these classes, e.g., initialization sequences and exit sequences, but the new application is otherwise completely free to define the sequence of actions on ActiveDevices, to perform any calculations to determine results using external libraries if needed, to exchange data and synchronization points with other sequences (rendezvous), and to update the user interface at any time. During initialization, AppSequences receive references to the ActiveDevice and GUI panel containers in order to have access to the system's resources, and each sequence object announces to the Scheduler with what other AppSequences it needs to collaborate during execution.

GUI panels are plain object containers, which get modified and configured at design-time. All GUI elements like text boxes, list boxes, labels, buttons, tab containers, and frames that are needed by the application can be added freely without any other restriction but the real estate available, which cannot be modified. The number of panels available at run-time is also configured at design-time, being a minimum of one and a maximum of eight. Panels that are not used are hidden at run-time, and users can select which GUI panel is displayed by clicking a button on the screen, independently of the current state of the system (Fig. 26.10). AppSequences can access any GUI widget through the object reference to the container they all have.

Observers are also reused as code templates, and they are typically used to update the contents of GUI panels in an asynchronous manner. Subject and Observer objects are implemented according to the *Observer* design pattern

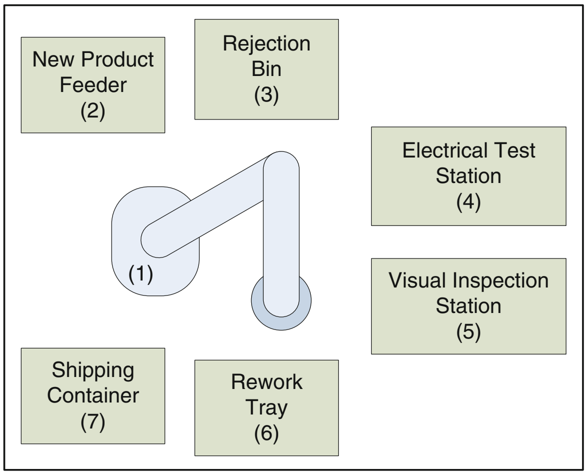

Fig. 26.11 A robotic cell based on the PATF product platform

(Gamma et al. 1995), and as described in the referenced book, each Observer subscribes to its corresponding subject during initialization. Subjects are typically updated by AppSequences.

DUT Collection is just another code template that provides a multithreaded execution environment for a collection of application-specific objects defined by a free-form class representing devices under test.

Final Product Test Example

Further elaborating on the discussion of *Separation of Concerns* as it relates to the implementation of AppSequences, we now present a concrete example to clarify and show how this approach helps to isolate the truly application-dependent features from the common product family functionality.

Figure 26.11 depicts a robotic cell application where we have a robot arm (1) with a gripper to pick up a finished product from the input feeder (2) and take it to an electrical functional test station (4) in the first step of the process. If the electrical test fails, the robot takes the product to a rejection chute (3) that guides the scrapped product to a bin and then picks up a new part from the input feeder. If the product passes the test, it continues in the process and is then taken to a machine vision station (5) for final cosmetic inspection. The visual inspection station consists of a digital camera and its illumination system. If the product fails the visual inspection, the robot takes the product to a tray (6) that is used to ship a batch of rejected products to an external rework station when it's full. If the product passes the visual inspection, then the robot packs the product in the shipping container (7).

This cell is required to keep the count of defective products, count of products sent for rework, and count of good products packed in shipping containers. Additionally, it is required to store the detailed results of all electrical and visual tests, associate them with the serial number of each product and compute statistical results, and process capability of the robotic cell.

The controlling software for this cell would be implemented as follows:

1. A new class named RobotSequence is derived from the AppProductSequence class (see Fig. 26.13) to implement the sequence of actions required to control the robot's movements to and from each position in the robot cell, according to the scenario described above.
2. Another class named ElectricalTestSequence, also derived from the concrete class AppProductSequence, is implemented to execute all the electrical tests at station (4) and to determine the final Pass/Fail result, based on the measurements from each product tested. This class must include a rendezvous point that is exchanged with RobotSequence to stop the robot at the electrical test station and pass in the test result, so the robot can proceed to either drop the product in the rejection chute (3) or continue on to the next test station (5). Measurement instruments are all ActiveDevices, as described above.
3. The third sequence class, called VisualInspectionSequence, is implemented with the image processing algorithms that are needed to perform the cosmetic inspection. These image processing functions are called from an external library acquired from a third party that specializes in machine vision algorithms. The sequence also handles digital inputs and outputs to control the illumination system. This class also implements a rendezvous point to synchronize with RobotSequence, stopping the robot in front of the camera and exchanging data representing the inspection results, upon which RobotSequence reacts subsequently by placing the product either on the rework tray or in the shipping box.
4. The fourth sequence class is implemented to keep track of the rework tray and automatically swap a full tray with an empty one using a mechanical actuator without having to interfere with the other components of the robot cell. This sequence does not need to synchronize with the robot because they are completely asynchronous and independent. The swapping action is indirectly triggered as the consequence of having the robot place a product in the last available spot on the tray.
5. A Subject object is instantiated and assigned an object of type ProductCount, which holds the current counts of products tested, rejected, reworked, and shipped.
6. Another Subject object is instantiated and assigned an object of type TestResults, which holds all the electrical parameter measurements for each product, plus the cumulative statistical results and metrics.
7. Two Observer objects are created based on the Observer class template. Each of them subscribes to its corresponding Subject objects: ProductCount or TestResults. Each observer is responsible for updating its own GUI panel, which is assigned to them at run-time.

Notice how convenient it is to focus on just one thing at a time (*Separation of Concerns*) when implementing a new application based on the PATF platform, even one as simplified as this example. This advantage becomes more evident as the complexity of the application increases. Additionally, this approach enables teams

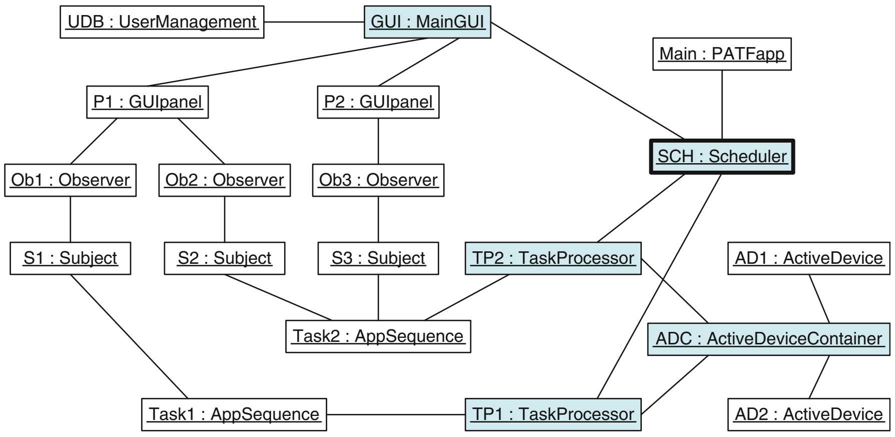

Fig. 26.12 Partial view of the PATF run-time structure (object collaboration)

to divide and conquer the complexity of the problem and to distribute programming work among team members without affecting the system's consistency, resulting in shorter development times.

26.5.2.2 Run-Time System Composition

One of the great benefits of creating a software platform for a family of products is code reuse. Black-box code reuse, in particular, has additional side benefits including robustness, known quality and reliability, and software maturity that improves with each new product that is implemented using the family platform.

Software can be reused in binary form if a multitude of compiled software components are assembled into a customized application at run-time. This is achieved through object composition, as described below. Figure 26.12 is a UML Object Collaboration Diagram showing a partial view of the run-time structure of a generic application built on the PATF product family platform. During initialization, the Main application instantiates the Scheduler, which is the orchestrator of a PATF application, and a chain of events is triggered, at the end of which the whole application is composed and ready to run.

The sequence of events during start-up is roughly as follows:

1. *Main* application starts, indicating a number of parameters and resource locations for the framework's use. *Scheduler* is instantiated, who takes control over the execution and suspends *Main*. From that point on, *Scheduler* is responsible for instantiating all the necessary objects and assembling them into a complete working application.
2. *Scheduler* instantiates *MainGUI*, which in turn instantiates *GUIPanels* and *UserManagement*. The latter component is responsible for providing user authentication services to *MainGUI* and can use a local database, a remote

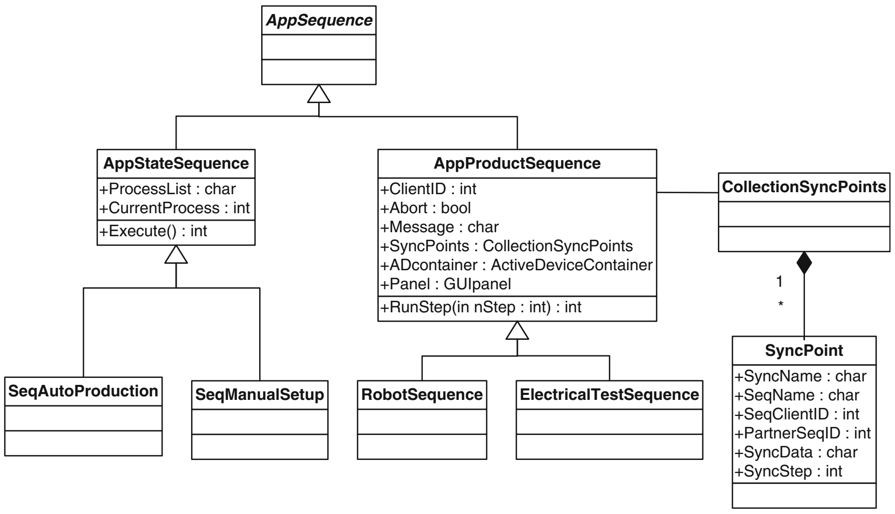

Fig. 26.13 Partial class diagram showing the structure of application sequences in PATF

database connection, or delegate requests to a corporate user repository, details of which are completely hidden from *MainGUI*.

3. Once *MainGUI* is up and running, *Scheduler* instantiates the *ActiveDevice-Container* object, which in turn instantiates all its children components, i.e., all the *ActiveDevices* available to the system in a particular application. At this point, the application objects are all fully assembled in memory, and the system waits in the MANUAL_IDLE state (see Fig. 26.10) for user input about the next action.

All applications built on top of the PATF platform behave according to the state machine shown in Fig. 26.10. When the user requests a change of state, the state machine executes the sequence of actions specified in the AppStateSequence class corresponding to that state. Figure 26.13 shows a UML class diagram partially showing the structure of AppSequences, with a faint reference to the robot cell example described above.

As shown in Fig. 26.13, there are two subclasses derived from the abstract class *AppSequence*: one that defines the actions the system must take during a state transition (*AppStateSequence*) and one that defines the actions to be taken while the system remains in that state (*AppProductSequence*). There are eight *AppState-Sequences* that are expected by *Scheduler*, and therefore, they must be implemented by the new product application. These eight classes correspond to each of the low-level system states shown in Fig. 26.10, plus *Initialization* and *Shutdown*. Each *AppStateSequence*class launches one or more *AppProductSequences* as required to be run in their corresponding state. *AppSequences* can be executed in either *Concurrent* or *Sequential* mode, and *AppSequences* may spawn other *AppSequences* in either execution mode. Once children sequences terminate, the parent terminates too. This scheme enables product designers to implement dynamic system behaviors of arbitrary complexity.

When an *AppProductSequence* object starts, it is responsible for attaching themselves to *Subject* objects so they can post the information that is to be displayed on GUI panels after any transformations are made by the corresponding *Observer* objects.

SyncPoint objects are instantiated by each *AppProductSequence* object that needs to rendezvous with another *AppProductSequence*. When execution of a sequence reaches the *SyncStep*point, the *SyncPoint* object is handed over to *Scheduler*, who takes the request and puts the sequence to sleep until its advertised partner *PartnerSeqID* is ready for rendezvous. At that point, the dormant sequence is awaken, and *Scheduler* swaps the *SyncPoint* objects and sends them back to the partner sequences, thus enabling information exchange between the partners, as described in the robot cell example above.

Run-time object composition as described above maximizes code reuse in binary form, thus ensuring that a more solid product is implemented in a very short time with great technical and economic benefits, as shown in the study of code reuse presented in the next section.

26.6 Code Reuse in Product Platforms Vs. Traditional Approach

The PATF product family platform was implemented and initially tested on three different applications within the field of automated industrial manufacturing and testing. More than twenty different products have been implemented and deployed since then.

The first application, System A, is an automated assembly cell for building microelectromechanical system (MEMS) devices. This task requires very high precision, high performance, and high flexibility. It includes a vision-guided SCARA robot, multiple servo-controlled linear actuators, and a range of other industrial automation devices.

The second application, System B, uses the same type of robot in a completely different robot cell platform for automatically testing the same MEMS devices. In this case, the system features multiple automated part feeders that enable uninterrupted operation. This system uses high-precision optical and instrumentation equipment and performs a series of mathematically intensive calculations.

The third application, System C, is a manual version of the robotic tester, using the same optical equipment and mathematical algorithms but manual part handling, interactive operation, and a different set of operator safety policies.

26.6.1 Implementation

The system was implemented to run on a Microsoft Windows® platform using Microsoft COM technology. Active objects were implemented as multithreaded COM servers (Microsoft 1994), which produce native code for faster execution.

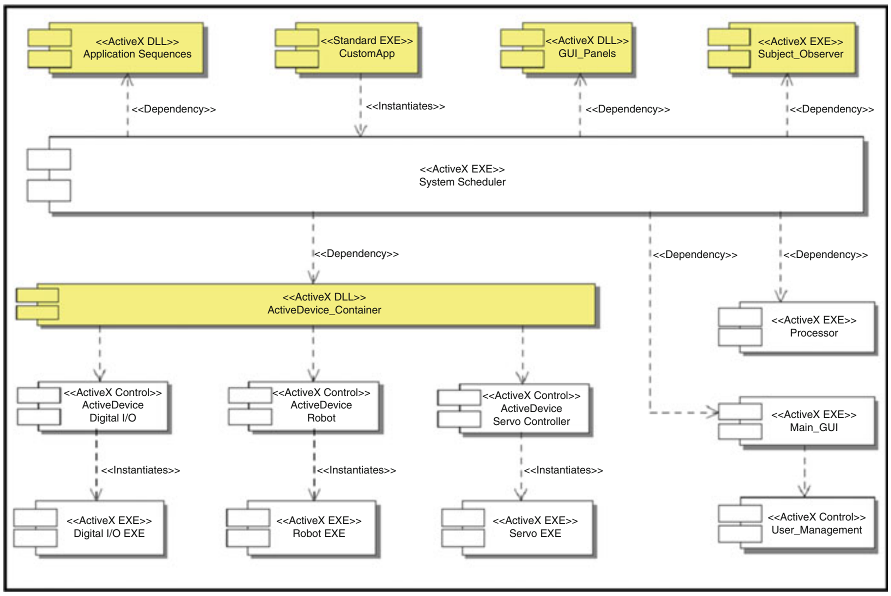

Fig. 26.14 System deployment diagram

In order to simplify the use of ActiveDevices, these components were implemented in two parts: an ActiveX control and an ActiveX EXE executable. The ActiveX control can be dragged and dropped onto an object container, which in turn instantiates an out-of-process COM server in a way that becomes transparent to the application programmer. Figure 26.14 shows a UML deployment diagram with the code components required to implement System A, described above. Similar structures were used to implement Systems B and C.

Software components represented with white symbols are reusable in binary form across multiple applications, and shaded symbols (top row) represent components that must be modified with application-specific code. Therefore, the set of software components that comprise an instance of the system architecture described herein is divided in two subsets according to their relation to the custom application: They are either dependent or independent.

26.6.2 Application-Independent Components

Components represented in the UML deployment diagram of Fig. 26.9 are directly mapped from high-level classes in the system architecture. Notice that virtually every software components in the Product Family Infrastructure layer are reusable in executable form. The only exception is ActiveDevice_Container, which is discussed below.

All of the components in the Family Platform Framework layer are also reusable in binary form across multiple applications, with the sole exception of the Subject_Observer component, which is also discussed below.

From an economical point of view, components that are reusable in binary form are very valuable assets for an organization, since they are designed, developed, and tested once and can then be used in many future applications using the same devices abstracted by these software components.

26.6.3 Application-Dependent Components

As naturally expected, most of the components in the top layer (product-specific features) must be customized to fit the target application. Nevertheless, these components are developed based on code templates, where more than 60 % of the source code is reused, with the exception of App_Sequence, which only reuses a code skeleton or, in other words, just the interface.

Going back to ActiveDevice_Container, there is virtually no code required in this component, since its only purpose is to package the selected ActiveDevice components into a single compiled component.

As its name hints, Subject_Observer serves objects of two types: Subject and Observer. There is only one class for Subject, which requires no customization whatever. It is reused at the source code level as is. On the other hand, an Observer class is supplied as an example and code template, which then has to be copied and modified to fit the application at hand. For customization, a typical Observer class requires modifying or writing less than 100 lines of code.

The case for GUI_Panels is very similar, although most of the modifications are related to user interface objects, like command buttons, data grids, data-bound controls, and labels.

Finally, we get to the executable program CustomApp, which in fact is only a generic name for this component. This program has the sole task of instantiating the System Scheduler with the main settings for the user interface. This is a template project that requires customizing <10 lines of code. In practice, this program is usually compiled with the application name, for instance, RobotCell.exe. Once instantiated, the System Scheduler takes over, and from that point on, CustomApp is actually relegated to the background.

26.6.4 Evaluation

We now set to evaluate the impact of this approach to software architecture in terms of code reuse and programmer productivity. At the time this project was completed, a survey of the literature on software reuse metrics was made (Chidamber and Kemerer 1994; Price and Demurjian 1997; Price et al. 2001; Washizaki et al. 2003;

Table 26.1 Summary of product family optimization methods evaluated in study

Projects with similar applications	Total engineering hours
Project A	3,030
Project B	2,290
Project C	2,225
Project D	1,530
Project E (using PATF)	384

Devanbu et al. 1996; Chen et al. 1995; Ferri et al. 1997; Cardino et al. 1997). The objective was to find an appropriate technique that would reflect the benefits derived from reusing enterprise frameworks, where reuse includes both design and executable code. Most proposed metrics addressed class structure, complexity, and static relations that use source code files and other fine-grain elements as inputs. The coarse-grain modularity and functionality of an enterprise framework lay beyond the scope of such metrics. Our conclusion was that new metrics were needed, such that the influence of reusing design, architecture, and other system features located at higher levels of abstraction are also taken into account.

In the case of application frameworks, we think that reuse of architectural design is a major factor in the success of deployed applications using it. When architectural design is reused, the design phase of the software development cycle is greatly simplified, reducing it to a mapping of the application's specific requirements to the different components offered by the standardized architecture, which results in a well-known system organization that streamlines the implementation phase and enhances the quality of the final product. This effect includes important engineering labor savings, which should be quantified in order to take this important benefit into consideration.

26.6.5 Methodology

Given this situation, we opted for an informal and simple pragmatic approach to measuring the impact of framework reuse and carry a soft evaluation by using historical data from comparable projects previously completed by the same software development team.

Table 26.1 shows a list of projects previously developed and their respective software development cost in man-hours. All listed applications are very similar robotic cells for automated manufacturing. They all integrate the same robot, the same machine vision system, the same servomotors for linear motion, and other industrial automation tools. Their main difference is that each machine builds a different product. Projects A, B, C, and D were developed using the traditional approach to software development. Project E was implemented using the PATF product family platform. With this information as our starting point, we took the number of engineering hours that were required to design, implement, test, and debug the previous projects and then compared it to the effort required to

Table 26.2 PATF black-box reuse profile (Project E)

Component	SSI	RSI	Reuse (%)
Scheduler	112,963	112,963	100
Processor	15,961	15,961	100
Main GUI	18,046	18,046	100
User management	14,584	14,584	100
ActiveDevice robot	110,864	110,864	100
ActiveDevice DIO	12,449	12,449	100
ActiveDevice DMM	12,170	12,170	100
ActiveDevice vision	14,762	14,762	100
ActiveDeviceDeviceNet	16,842	16,842	100
ActiveDevice servo	15,308	15,308	100
Total	343,949	343,949	100

Key: *SSI* shipped source instructions, *RSI* reused source instructions

implement a brand new application using the PATF platform. Please note that this table does not include the effort required to design, develop, and implement the platform itself (3,800 h). However, this overhead cost is included in the final evaluation shown in Table 26.5.

26.6.6 *Results*

Let us now characterize the PATF platform and its reusable code base, as fully implemented in the completed product family platform or enterprise framework.

26.6.6.1 Code Reusability

Table 26.2 shows the PATF black-box reuse profile, which lists software components that are reused in binary (executable) form. The total number of shipped source instruction lines (SSI) is given as an indicator of the component's size and cost. This is later used as a reference point when quantifying labor savings. Since these components are reused without any changes, the total black-box reuse ratio is 100 %.

Table 26.3 shows the PATF white-box reuse profile. It presents the source code reuse ratio for components that are reused as project and code templates. These templates have to be modified to suit the new application at hand. Naturally, all application-specific code components show low source code reuse ratios. Nevertheless, the overall white-box reuse ratio reaches 20 %, which accounts mostly for abstract classes, type libraries, and other PATF interface elements.

Table 26.4 summarizes the PATF Compound Reuse Profile, which comprises both black-box and white-box reuse ratios, yielding a net shipped code reuse of more than 90 %.

Table 26.3 PATF white-box reuse profile (Project E)

Component	SSI	RSI	Reuse (%)
Subject	1,597	1,597	100
CustomApp	1,354	620	46
ActiveDevice container	1,288	425	33
GUI_Panels	2,761	863	31
AppSequence	17,620	1,621	21
Observer	2,678	366	14
Total	27,298	5,492	20

Key: *SSI* shipped source instructions, *RSI* reused source instructions

Table 26.4 PATF compound reuse profile (Project E)

Component	SSI	RSI	Reuse (%)
Black-box reuse	343,949	343,949	100
White-box reuse	27,298	5,492	20
Total	371,247	349,441	94

Key: *SSI* shipped source instructions, *RSI* reused source instructions

Table 26.5 Comparing PATF vs. traditional approach

Software implementation technique	Total engineering hours
Total project cost using traditional approach (median)	2,258
Total project cost using PATF as the platform (includes 10 % of the development cost of the PATF framework itself)	766
Development cost reduction:	66 %
Productivity improvement:	295 %

26.6.6.2 Cost Savings When Using the Platform

It is evident that a reduction in the total implementation effort translates directly into cost savings. Looking at the previous projects, we find that the median development effort for Projects A, B, C, and D is 2,258 man-hours, and the average is 2,269 man-hours.

On the other hand, the total cost for the design, implementation, and test of the PATF framework was 3,815 man-hours. Once ten applications were implemented based on the PATF framework, the investment resulted in a unit cost of approximately 382 man-hours per system. If we add this PATF unit cost to the implementation effort for Project D and take the median as a representative cost value for projects implemented using the traditional approach, we get the results shown in Table 26.5.

The numbers show that projects implemented with the PATF platform cost approximately 1/3 of what it would cost if it were developed from scratch. Likewise, since our cost units are man-hours, the same numbers show that the software team gets the job done three times faster than with the traditional approach.

Cost savings in engineering effort are an important piece of information to economically justify the greater investment needed to develop reusable software architectures and code within an organization.

Keep in mind, however, that software product platforms, or enterprise application frameworks, are hard to design and the challenge should not be taken lightly. It requires an expert software architect with deep knowledge of the domain, supported by a team of software professionals, and the full commitment of the organization to make the significant investment that requires such project. Nevertheless, the payoff of a successful software product platform is significant and rewarding in the long term.

26.6.6.3 Other Observations

Since PATF implies that a complete set of design decisions have been made for software developers of future projects that use it, they did not have to spend any time at all designing their new applications.

Likewise, developers had to spend no time thinking of how to identify, create, or abstract new modules or interfaces, since everything is fixed a priori in the framework. Functional policies that are typical to most applications within the particular domain are also implemented in the underlying design and platform. Therefore, the only task left for them to do was to focus on the particular functional details that made the target application unique, i.e., the specific algorithms of their new assignment.

Due to the phenomenon of inversion of control exhibited by an application framework, programmers are forced to write application-specific code strictly following the guidelines and interfaces prescribed in the framework design. The resulting code was more homogeneous and standardized across the team, as compared to code produced for previous projects. This effect is an additional benefit, since it also helps to improve the software product maintainability.

It is important to mention, however, that at the beginning, it was a difficult task for software developers to grasp the system model and the new inverted-control programming style where the family platform framework was in charge, not them. Inflexibility of the interface design and the multitude of stand-alone external components required by the system as mandatory also demanded additional adjustments to the typical programmer mindset. Nevertheless, after the learning period was over, programmers acknowledged the benefits of the new structured development based on the new product platform.

26.6.6.4 Code Reuse Conclusion

In agreement with previous reports (Fayad et al. 2000; Fayad and Schmidt 1997; Schmidt and Fayad 1997; Yassin and Fayad 1999; Poulin et al. 1993), the effect of using a software product platform for the industrial automation and test domain was positive, having achieved an estimated level of executable code reusability above 90 % and cost savings of about 60 %. Productivity of software development teams was improved, although the learning curve could be steep for some programmers.

Reusing both design and code yields multiple benefits in terms of cost, new product development lead time, and robust quality of the production code due to extensive testing at each new product iteration. Homogeneity and consistency among different members of the product family makes maintenance and evolution feasible, efficient, and cost-effective.

References

Alur D, Crupi J, Malks D (2003) Core J2EE patterns, 2nd edn. Prentice Hall, Upper Saddle River, NJ

Booch G, Rumbaugh J, Jacobson I (1998) The unified modeling language user guide. Addison-Wesley, Reading, MA

Brooks FP (1975) The mythical man month. Addison-Wesley, Reading, MA

Buschmann F et al (1996) Pattern-oriented software architecture: a system of patterns. Wiley, Chichester

CACM (1997) Special issue on object-oriented application frameworks. Commun ACM 40 (10):32–87

Cardino G, Baruchelli F, Valerio A (1997) The evaluation of framework reusability. ACM SIGAPP Appl Comput Rev 5(2):21–27

Chen YF, Krishnamurti B, Vo KP (1995) An objective reuse metric: model and methodology. In: Schäfer W, Botella P (eds) ESEC '95: 5th European software engineering conference, Sitges, Spain, Sept 1995

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Soft Eng 20(6):476–493

Clemens P, Bachmann F, Bass L, Garlan D, Ivers J, Little R, Merson P, Nord R, Stafford J (2011) Documenting software architectures, 2nd edn. Addison-Wesley, Reading, MA

Cockburn A (2000) Writing effective use cases. Addison-Wesley Professional, Reading, MA

Daigneau R (2011) Service design patterns: fundamental design solutions for SOAP/WSDL and RESTful web services. Addison-Wesley Professional, Reading, MA

Devanbu P, Kartsu S, Melo W, Thomas W (1996) Analytical and empirical evaluation of software reuse metrics. In: Dieter Rombach H et al (eds) ICSE '96: 18th international conference on software engineering, Berlin, Germany, 25–29 Mar 1996. Proceedings IEEE Computer Society 1996, pp 189–199

Douglass BP (2002) Real-time design patterns: robust scalable architecture for real-time systems. Addison-Wesley Professional, Reading, MA

Douglass BP (2011) Design patterns for embedded systems in C. Newnes, Boston, MA

Fayad ME, Schmidt D (1997) Object-oriented application frameworks. Commun ACM 40 (10):32–38

Fayad ME, Schmidt D, Johnson R (1999) Building application frameworks: object-oriented foundations of framework design. Wiley, New York, NY

Fayad ME and Johnson R, (1999) Domain-Specific Application Frameworks: framework Experience by Industry. Wiley, New York

Fayad ME, Hamu DS, Brugali D (2000) Enterprise frameworks: characteristics, criteria and challenges. Commun ACM 43(10):39–46

Ferri RN, Pratiwadi RN, Rivera LM, Shakir M, Snyder JJ, Thomas DW, Chen YF, Fowler GS, Krishnamurti B, Vo KP (1997) Software reuse metrics for an industrial project. In: Bieman et al (eds) METRICS '97: Proceedings of the 4th international software metrics symposium, Albuquerque, NM, Nov 1997, pp 165–173

Fowler M (2002) Patterns of enterprise application architecture. Addison-Wesley Professional, Reading, MA

Fowler M (2003) UML distilled, 3rd edn. Addison-Wesley, Reading, MA

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading, MA

IEEE (2011) ISO/IEC/IEEE Std 42010-2011 systems and software engineering: architecture description, international standard

ISO (2011) The ISO architecture group maintains a survey of all known architecture frameworks as reference material in their ISO/IEC/IEEE Std 42010. http://www.iso-architecture.org/ieee-1471/afs/frameworks-table.html. Accessed Jul 2012

Jacobson I, Christerson M, Jonsson P, Overgaard G (1992) Object-oriented software engineering: a use case driven approach. Addison-Wesley, Reading, MA

Johnson RE (1992) Documenting frameworks using patterns. In: Pugh J (ed) OOPSLA '92: ACM conference on object oriented programming systems, languages and applications, conference proceedings, Vancouver, BC, Canada, Oct 1992

Johnson RE (1997) Frameworks=(components+patterns). Commun ACM 40(10):39–42

Microsoft (1994) COM and COM+Technology reference papers. http://msdn.microsoft.com/

Morales CO (2013) Chapter 21: Design principles for reusable software platforms. In: Simpson T, Jiao R, Siddique Z, Holtta-Otto K (eds) Advances in Product Family and Product Platform Design: Methods and Applications, Springer, New York, NY

Object Management Group (2011) The unified modeling language UML, ver. 2.4.1, Object Management Group

Poulin J, Caruso J, Hancock D (1993) The business case for software reuse. IBM Syst J 32 (4):567–594

Price MW, Demurjian SA (1997) Analyzing and measuring reusability in object-oriented designs. In: Berman AM (ed) OOPSLA '97: ACM conference on object oriented programming systems languages and applications, conference proceedings, Atlanta, GA, Oct 1997, pp 22–33

Price MW,Needham DM, Demurjian SA (2001) Producing reusable object-oriented components: a domain-and-organization-specific perspective. In: Proceedings of ACM symposium on software reusability SSR '01, Toronto, ON, Canada, May 2001, pp 41–50

Rumbaugh J, Jacobson I, Booch G (1998) The unified modeling language reference manual. Addison-Wesley, Reading, MA

Russell S, Norvig P (2009) Artificial Intelligence: a modern approach, 3rd edn. Prentice Hall, Upper Saddle River, NJ

Schmidt D, Fayad ME (1997) Lessons learned building reusable OO frameworks for distributed software. Commun ACM 40(10):85–87

Schmidt D, Stal M, Rohnert H, Buschmann F (2000) Pattern-oriented software architecture, vol 2: patterns for concurrent and networked objects. Wiley, Chichester

Washizaki H, Yamamoto H, Fukuzawa Y (2003) A metrics suite for measuring reusability of software components. In: Proceedings of the 9th IEEE international software metrics symposium (METRICS 2003), 3–5 Sept 2003, Sydney, Australia, pp 211–223

Yassin AF, Fayad ME (1999) Chapter 29: A survey of object-oriented application frameworks. In: Fayad ME, Johnson R (eds) Domain-specific application frameworks. Wiley, New York, NY, pp 615–632

Zachman JA (1987) A framework for information systems architecture. IBM Syst J 26(3):276–292