SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2002; **32**:801–831 (DOI: 10.1002/spe.460)

Enterprise frameworks: issues and research directions

Hafedh Mili[1], Mohamed Fayad[2,*,†], Davide Brugali[3], David Hamu[4] and Dov Dori[5,6]

[1]*Department of Computer Science, University of Quebec at Montréal, Canada*
[2]*Department of Computer Science and Engineering University of Nebraska at Lincoln, U.S.A.*
[3]*Facoltà di Ingegneria—Università di Bergamo, Dalmine, Italy*
[4]*Liberty Consulting, Mesa, Arizona, U.S.A.*
[5]*Massachusetts Institute of Technology, Cambridge, MA, U.S.A.*
[6]*Technion, Israel Institute of Technology, Haifa, Israel*

SUMMARY

Enterprise frameworks are a special class of application frameworks. They are distinguished from other application frameworks in terms of scale and focus. In terms of focus, application frameworks typically cover one particular aspect of an application, either a domain-dependent aspect (e.g., billing in a web-based customer-to-business ordering system), or a computational infrastructure aspect such as distribution, man-machine interface, or persistence, etc. Generally, an application framework alone delivers no useful end-user function. With infrastructure frameworks, we still have to plug in domain functionalities, while with domain frameworks, we need to set-up the infrastructure. In contrast, enterprise frameworks embody a reference architecture for an entire application, covering both the infrastructure aspects of the application, and much of the domain-specific functionality. Instantiating an enterprise framework is nothing short of application engineering, where the architecture and many of the components are reusable. While creativity and continual improvement may be the major ingredients for building a good application framework, anything related to enterprise frameworks, be it building, documenting, or instantiating them, is complex and requires careful design and planning. In this paper, we identify the issues involved in building, using, and maintaining enterprise frameworks, both from research and practical perspective. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: application frameworks; enterprise information systems; software architecture; separation of concerns

1. ORIGIN OF FRAMEWORKS

1.1. A first definition

Different definitions of application frameworks stress different aspects of frameworks, including their objective, structure, and behavior. We start with a simple definition, and refine it as we proceed.

---

[*] Correspondence to: Mohammed Fayad, Department of Computer Science and Engineering, University of Nebraska at Lincoln, P.O. Box 880115, NE 68588-0115, U.S.A.
[†] E-mail: fayad@cse.unl.edu

An application framework may be roughly defined as a set of interacting objects that, together, realize a set of functions. The set of functions defines the area of 'expertise' or 'competencies' of the framework; we refer to it as the domain of the framework. A *domain* may be either a subset of the *business domain* (the problem space), or a subset of the *computing domain* (the solution space). A banking application framework, for example, implements functions with the business domain of banking. The Model View Controller (MVC) framework, developed for the Smalltalk language, covers a subset of the *computing domain*, and more specifically, the design domain. It does so by addressing the problem of connecting business logic with graphic user interface (GUI) logic in a way that minimizes the dependencies between the two. Other computing domain frameworks address architectural issues, and include artifacts such as middleware frameworks (e.g., CORBA, EJB, and COM).

An application framework may be described by the equation:

$$\text{Application framework} = \text{a blueprint} + \text{component realizations}$$

Most authors use the term *design* instead of *blueprint*. This is perhaps a reflection of the fact that the first application frameworks known to the computing profession were computing domain frameworks, and more specifically, GUI frameworks. However, when we talk about business frameworks, the framework identifies domain classes, their interrelationships, and their interactions (analysis level description), and possibly the design and partial realization of such classes, interrelationships, and interactions. In this case, the *blueprint*, can describe the analysis, the design, or both. Clearly, the earlier in the development lifecycle reuse takes place, the greater the benefits. An analysis-only framework (sometimes referred to as a modeling framework) typically focuses on analysis level constructs, without making any commitment, design-wise, to the way those classes will be implemented. Analysis-only frameworks are typically the product of *domain analysis*. The IBM SanFrancisco initiative aims at developing modeling frameworks for a variety of business domains.

1.2. The structure of frameworks

A framework views an *application fragment* as a set of objects that interact to accomplish a set of domain functions. An enterprise framework includes three sets:

- a set of framework participants;
- a set of relationships between the framework participants; and
- a set of interaction scenarios between the framework participants.

The participants are described in terms of obligations: each framework participant fulfills a particular *role* within the framework. That role is often described in terms of an interface that the participant must support. The interface consists of a set of attributes and method signatures that a component fulfilling that role must implement.

The description of the behavior of the participants is inevitably incomplete. This is not only true because the participants may have other behavior that is not relevant to the framework, but because some of the framework-related behavior cannot be specified generically, in which case it is specified 'intentionally' and often, informally, or worse, implicitly. For example, in the MVC framework, it is important that all the operations that modify the state of the model broadcast that change (using the dependents mechanism) to propagate to the various views. However, there is no easy way of expressing this constraint.

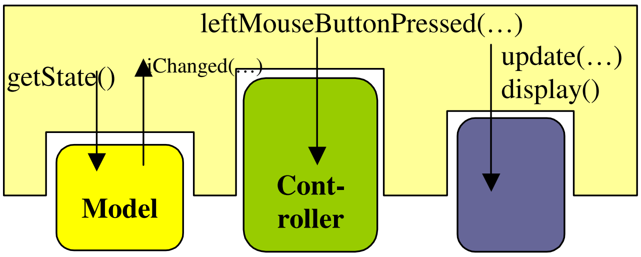

Figure 1. Framework documentation only exposes the part of the framework that needs customizing.

The structure of frameworks also typically reflects the idea of inversion of control[‡]. Instead of the application-specific code calling reusable code, we have reusable code calling application-specific code, and knowing when that application-specific code will be called, we can put in it whatever it is that we want done in certain situations.

A framework does not expose all of its internals, but only those parts that need to be customized by the framework user to adapt it to their needs; for a given set of functionalities, the fewer and simpler the parts, the better the framework.

Figure 1 illustrates this. We have no knowledge of what happens inside, but we know when control gets out of the 'framework body', and which methods are called. Knowledge of what happens inside is required for evolving or emulating the design of the framework, and is usually reserved for advanced users [1].

1.3. A framework metamodel

Figure 2 shows a general metamodel of a framework. It shows the three components of a framework (participants, relations, and interaction sequences), and their relationships. Within a framework, each participant plays a certain role in the context of static relationships ('Relation') or in the context of interactions, which are the basic elements of an interaction sequence. We may think of an interaction sequence as the closure of more elementary interactions. For example, an interaction I_1 may result in an event e_1, which in turn invokes another interaction, and so forth. This representation is independent of the interaction style, and can also accommodate more direct interactions such as method calls [2]. We use this representation as the basis for expanding on the model of a framework.

This metamodel represents what is common about frameworks, regardless of focus or scale. We first revisit the notion of object-oriented (OO) frameworks in the context of this metamodel, and then point to potential areas of differences with enterprise frameworks.

[‡] Also referred to as 'the Hollywood principle: don't call us, we will call you'.

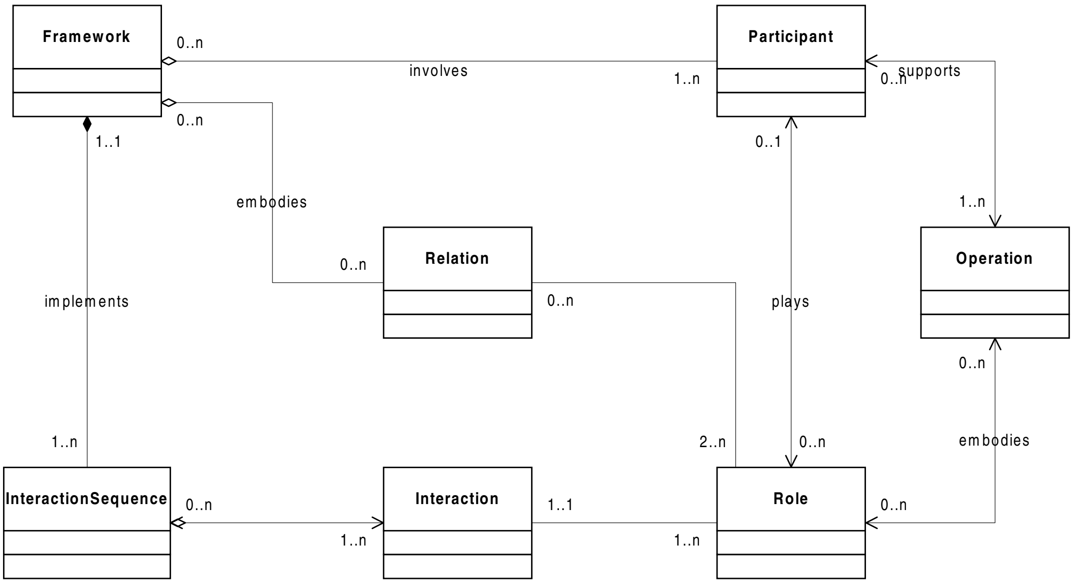

Figure 2. An initial metamodel of a framework's static aspect.

First, with OO application frameworks, the participants are typically objects, and the operations are methods attached to those objects. Because the code of a framework does not run in isolation, it may have to interface with the underlying software environment for some runtime services. If those services represent one of the points of variability (or *hot spots*, as they are often called) of the framework, then they may be wrapped by object-like service providers which will, in turn, delegate to the actual non-OO services. For example, a Java persistence framework that uses relational databases for storage would manipulate the database through an object interface, such as JDBC.

The second distinguishing characteristic of OO application frameworks is the interaction medium between the participants. With typical small-scale OO frameworks, the interaction mechanisms are typically at the language level, such as procedure calls. Remote procedural calls (as implemented by distribution frameworks such as CORBA or Java RMI) may be used in application frameworks that offer distributed services, but for the most part we stay close to language mechanisms. Assembling the components often requires writing glue code in the programming language with which the framework was implemented.

The third characteristic of OO application frameworks, which is a consequence of the first two, is that OO application frameworks are typically based on the same paradigm, and developed in the same language. This is mostly a matter of convenience. When given a problem to solve, we rarely choose a multi-paradigm, multi-language solution, unless the problem domain calls for one (e.g., the need to integrate with a legacy system).

With enterprise frameworks, the picture is markedly different, in part because of the focus of enterprise frameworks, and in part because of their scale. The characteristics of enterprise frameworks are discussed in Section 2.3.

2. STYLES, ARCHITECTURES, AND ENTERPRISE FRAMEWORKS

2.1. Architecture

Bass *et al.* [3, p. 23] defined software architecture as the high-level design of a software system in terms of software components (modules, subsystems, processes), their external properties (application programming interface (API), runtime behavior), and their inter-relationships.

> 'The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them.'

Since software tends to be very complex, several views are needed to reflect different properties of the software components, relationships between them, and different decompositions of the software.

Depending on the kind of components and the kind of relationships between them, the different structures that describe the software include:

- the *module structure,* where the components are software modules (packages, classes), and links represent dependency and containment relations;
- the *conceptual structure*, where the components are functional business units of the system, and the relationships denote data flow between them;
- the *process structure*, in which the components represent processes or threads, and the relationships between them represent constraints, such as 'synchronizes with', 'can't run with', 'pre-empts', and 'rendezvous with'.

Each structure reflects one aspect of the system, and serves a different purpose. The module structure may be used to identify dependencies among parts of the system, and break down system development into independent parallel tracks. While these structures reflect different concerns, they ultimately overlap, since the same software entities cut across them. The number and nature of architectural structures required to describe a software system ultimately depends on the application domain, and on the complexity of the application within that domain [3].

2.2. Architectural styles and connectors

Architectural styles are classes of similarly patterned software architectures, characterized by:

- *component types*: component classes characterized by either software packaging properties (e.g., 'COM component') or by functional (e.g., 'transaction monitor') or computational (e.g., 'persistence manager') roles within an application;
- *communication patterns between the components*: indicating the kinds of communications between the component types;

- *semantic constraints*, indicating the behavioral properties of the components individually, and in the context of their interactions;
- *a set of connectors*, which are software artefacts that enable us to implement the communication between the components in a way that satisfies the semantic constraints.

Researchers at Carnegie Mellon University and the Software Engineering Institute have catalogued a dozen or so styles grouped into the following five families [3,4]: (1) the *independent components* family; (2) the *data flow* family; (3) the *data-centered* family; (4) the *virtual machines* family; and (5) the *call and return* family.

Describing these styles in detail is beyond the scope of this paper. However, for our purposes, it is worth thinking of why these styles emerged. The short answer is that these styles had some intrinsic qualities, such as uniformity (e.g., applying the same interconnection pattern among components across the board), plus some design qualities, in terms of optimizing some design quality attributes (e.g., modifiability, performance, etc.). Different styles optimize different combinations of design quality attributes, which suit different *classes of applications*. Figure 3 shows an object-process diagram (OPD)[§] [5–8] of this structure. As the legend of the OPD shows, three structural relations are used: Aggregation–Participation (▲); Exhibition–Characterization (◬); and Generalization–Specialization (△).

2.3. Reference architectures and enterprise frameworks

We define enterprise frameworks as the embodiment of the semantics of an application domain, along with a computational architecture (or infrastructure) for applications within the domain. Clemens *et al.* [3] calls the semantics of the application domain *reference models*, which are typically the product of a domain analysis process. *Reference models* are distinguished from *reference architectures*, which are the result of mapping a reference model (domain model) into an architectural style (see Figure 4).

Using our definition of frameworks, we argue that reference models, in the sense of Bass *et al.* [3] are nothing but *analysis-only enterprise frameworks*. Reference architectures, in the sense defined by Bass *et al.*, constitute the blueprint for *architectural-level enterprise frameworks*. Concrete realizations of the architectural-level components may include the specification of components of the architecture in terms of an interface definition language and some form of documentation of the functionalities of the components, and their variants.

We are interested in enterprise frameworks that go beyond the architectural level. More specifically, we are interested in those that go down to the implementation level. Typically, well-designed frameworks include an implementation of the computational infrastructure of the reference architecture, and concrete realizations of some of the functional components. Such realizations may take the form of code templates, or as *components* in the sense defined by Szyperski [9], i.e. binary components ready to deploy such as COM or Enterprise Java beans. We argue that such enterprise

---

[§] The OPD was drawn using OPCAT—Object-Process CASE Tool. This computer-aided software engineering (CASE) tool supports OPM{xe “OPM” \t “See Object-Process Methodology”}{xe “Object-Process Methodology”} by drawing OPDs and checking the legality of the various links. It is downloadable from http://iew3.technion.ac.il/%7Edori/opcat/index-continue.html.

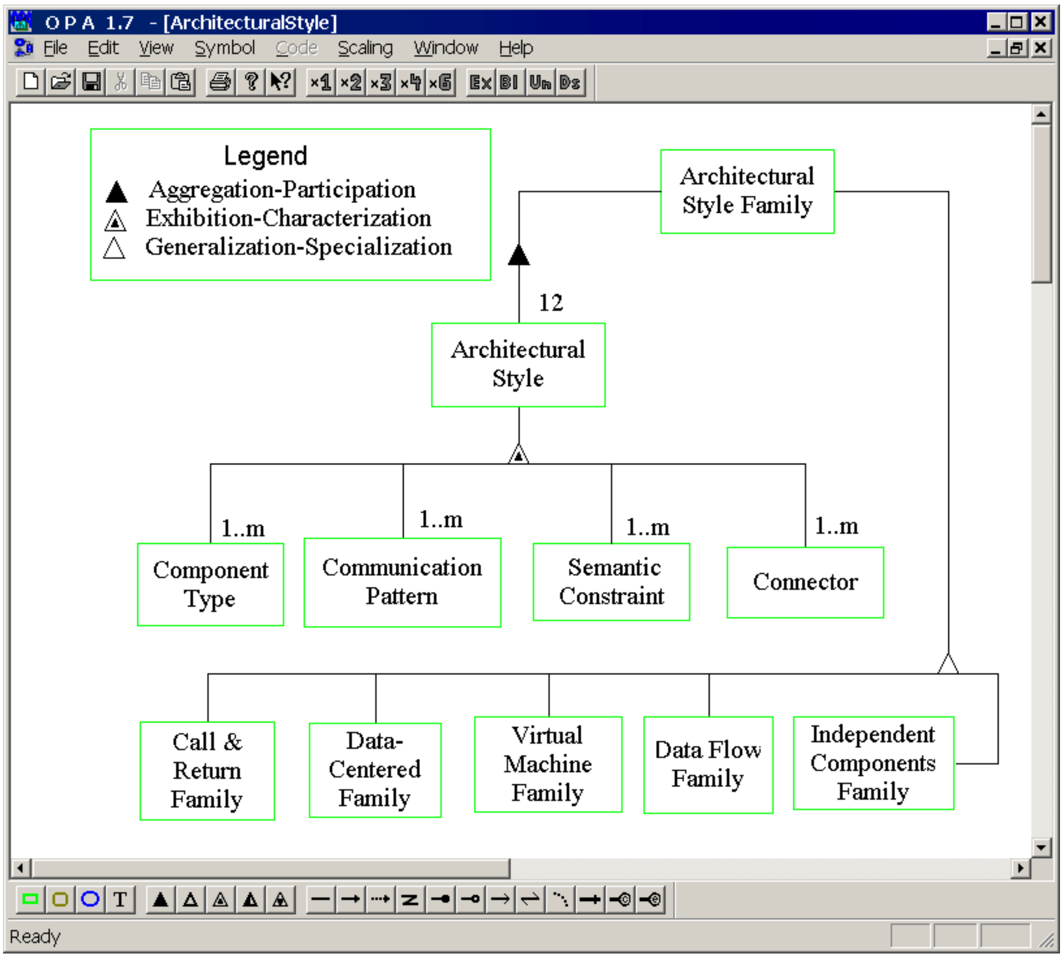

Figure 3. An OPD of the architectural style families, their specializations, and the attributes of each family.

frameworks are much more beneficial than architectural-level enterprise frameworks, in part because they allow us to reuse detailed designs and implementations, but more importantly, because they enforce, *de facto*, adherence to the architectural style by actually implementing it!

Figure 5 shows an enhanced metamodel for frameworks. We show here the possible packaging of framework participants, with three specializations, namely, *objects* (for the case of OO frameworks), binary components (e.g. COM, EJB beans, etc.), and *frameworks*! The idea that frameworks may be part of other frameworks comes from the fact that, from the outside, the instantiation of a framework

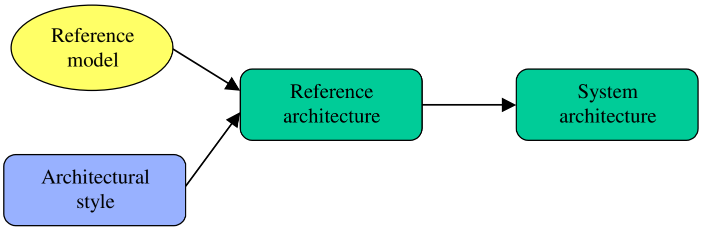

Figure 4. Architectural styles in reuse-centered development.

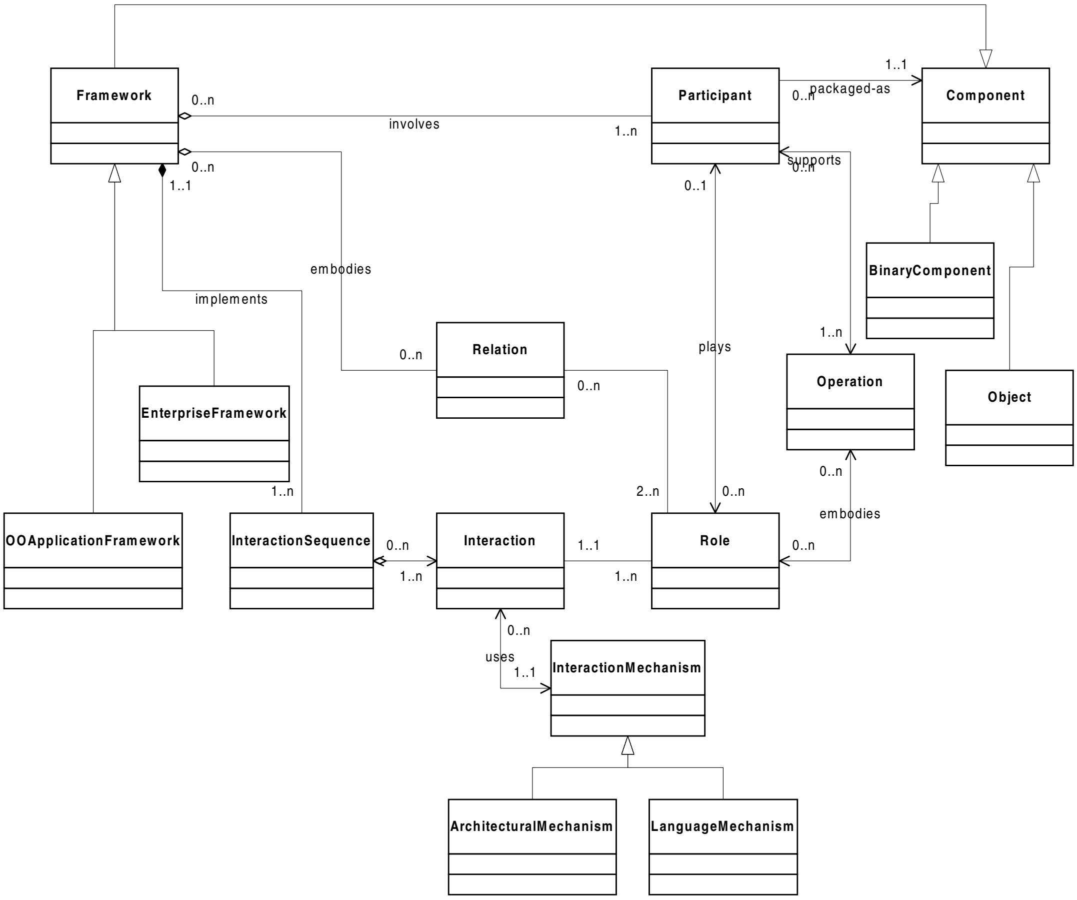

Figure 5. A metamodel of frameworks showing interaction mechanisms and participant packages.

may simply look like an aggregate object whose interface consists of the closure of the various interaction sequences (see, e.g., [2]). For instance, from a structural point of view, if we view a framework as a set of contracts between m components $C_1, C_2, \ldots, C_m$, if we add component C_{m+1} and contracts involving C_{m+1} and one of the components C_i for $1 \leqslant i \leqslant m$, we may consider the new contracts as contracts between C_{m+1} and an aggregate object having $C_1, \ldots,$ and C_m, as components [2]. In this particular case, the initial framework may be seen as an implementation of the behavior desired of the aggregate.

We also showed two kinds of interaction mechanisms that may be found in frameworks, namely *language mechanisms* and what we called *architectural mechanisms*, which are mechanisms that are inherent in architectural styles, and involve design level artifacts.

Because of a question of scale, at the first level view of enterprise frameworks, the participants are seldom objects; they are typically subsystems (or subframeworks) that implement distinct domain processes. These subsystems are seldom fully realized subsystems that do not require any customizing ('plug-and-play'); typically, the subsystems need to be instantiated in an application-specific way using a variety of instantiation and composition techniques. Because these subsystems need to interact, they have to be instantiated in a mutually consistent way.

In terms of component/participant interaction, unlike OO application frameworks where the integration mechanisms are typically language level mechanisms, with enterprise frameworks, the integration of components happens at the architectural level. Depending on the architectural style chosen, integration may require more or less customization effort.

Finally, while OO application frameworks tend to be single paradigm (OO) and single language, enterprise frameworks tend to be multi-paradigm and multi-language where, more often than not, existing independent components are integrated into a coherent whole. There are a few notable exceptions where the enterprise framework is built from the top down, with a cohesion in design, and tight integration between the components: a number of Enterprise Resource Planning (ERP) systems today are built this way (e.g. SAP).

3. ENTERPRISE FRAMEWORK CHARACTERISTICS

In selecting or building an enterprise framework, it is helpful to have a context by which to measure or grade the overall quality and goodness of fit of an enterprise framework. In practice, we have found that enterprise frameworks can be evaluated based on a variety of characteristics. We classify these characteristics in three broad categories: stability, adequacy, and economy.

3.1. Stability

The bulk of the engineering performed during a software project should be focused on those areas that will remain stable over time. Such an approach ensures a stable core design and, thus, a stable software product [10,11]. Those changes that are introduced to the software project will then be in the periphery, since the core was based on something that has remained and will remain stable. Therefore, only these small external modules need to be engineered. Through a stability-centric approach, we avoid the endless circle of reengineering entire software projects for minor changes.

Software stability is achieved through a variety of measures applied through a disciplined process applied throughout the software's lifecycle. However, software stability is much more than that. Where software stability is found, we often find related characteristics including:

1. mature runtime functionality;
2. support for extensibility, tailoring and customization;
3. enduring business themes;
4. enduring business processes;
5. support for separation of concerns;
6. separation of concerns.

3.1.1. Mature run-time functionality

An important characteristic of a good enterprise framework is that it provides mature runtime functionality within the specific domain in which it is to be applied. Although the functionality may be less mature in the early stages of the framework's lifespan, the framework is designed with many considerations that ensure that software stability and functionality remain high throughout the software lifespan.

A suitable framework requires relatively little code to meet the user requirements for new enterprise applications—this is the *challenge* for the framework team. Framework developers or providers must have a clear understanding of the application domain in which the framework will be applied, and the design and implementation of the framework must reflect that understanding.

3.1.2. Support for extensibility, tailoring, and customization

Software is frequently extended in ways its developers did not anticipate. A framework is not used like an application. Instead, a framework is the basis for constructing and delivering applications that are highly tailored and configured to a specific domain. Support for extensibility, tailoring, and customization ensure that the framework may adapt new constructs and more accurately model a specific domain.

A framework enhances extensibility and customization by providing explicit hook methods and hot-spots and other architectural elements that allow applications to extend its stable interface. Framework extensibility is essential to ensure timely customization of new application services and features.

3.1.3. Enduring business themes

Whereas an enduring business theme (EBT) is the structure for the sentences that are found in the semantics of an organization, *business objects* are instantiations of EBTs [10,11]. They fill in the blanks, attributes or semantics in these sentence structures. The EBT is concerned with a business issue; namely, 'what is being done', not 'how it is being done'. For example, in the transportation industry, the EBT is concerned with moving material from one location to another. The best way to identify EBTs is to look at the organization from its customer's point of view. The customer cares most about what is being accomplished, and only to a lesser extent, how it is being done.

Enterprise framework developers must be challenged to understand the purpose of the business as well as the current practices of the business. Therefore, the quality of an enterprise framework is measured in part on its ability to model both the enduring business themes and the current business practices.

3.1.4. Enduring business processes

Workflow management and enduring business processes (EPBs) are the key to an OO enterprise framework. Workflow management streamlines the complex interactions between objects that are found in large-scale OO applications. Proponents of frameworks go so far as to suggest that workflow mechanisms should eliminate the need for most application programming in the workplace.

Modern workflow management tools provide a graphical design palate for workflow definition. Nested state diagrams are well suited for the task of dynamic modeling of application workflows, although other process flow representations are equally applicable.

A workflow management metaphor provides the necessary modeling capabilities for constructing business processes. Well-designed frameworks should capture the EBPs or workflows that are fundamental to the target application domain. These enduring processes capture workflows that do not change over time. They are concerned with the high-level sequence of what is done (the top-level state diagram). The same workflow metaphor also provides a means for representing detailed and dynamic business processes (nested state diagrams), which are concerned with the step-by-step sequence of activities used to complete the task today.

Workflow management is commonly found in a variety of frameworks and traditional software products. Increasingly, workflow mechanisms have been incorporated into enterprise frameworks. Workflow provides a means to model both the enduring processes and the transient policies that govern business operations over time.

3.1.5. Support for separation of concerns

Separation of concerns is at the heart of framework development. The OO paradigm (OOP) works well only if the problem at hand can be described with relatively simple interfaces among objects. The core complexity is that concurrent/distributed systems have more than one dimension. Features such as scheduling, synchronization, fault tolerance, security, testing, and verification and validation are all expressed in such a way that they crosscut different objects. Hence, simple object interfaces are not violated and traditional OO benefits no longer hold.

A current attempt to resolve the issue is the aspect-oriented architecture. Aspect-oriented architectures address the multi-dimensional structure of any framework. Therefore, we distinguish between components and aspects.

Aspects are defined as properties of a system (framework) that do not necessarily align with the system's functional components. These properties include communication, performance, load balancing, synchronization, and scheduling. Aspect properties tend to cut across functional components, increasing their interdependencies, and resulting in a code-tangling problem. The aspect-oriented paradigm helps to retain the advantages of OOP and helps to avoid the tyranny of the dominant decomposition [26]. The goal is to achieve an improved separation of concerns in both design and implementation. The application of an aspect-oriented architecture is an important characteristic of OO enterprise frameworks and it helps to overcome the challenge of separation of concerns.

3.1.6. Ease of use

An enterprise framework must be user friendly. Enterprise frameworks are adapted to different client's needs through transparently attached role objects, each one representing a role that the framework objects play in that client's context. The object manages its role set dynamically. By representing roles as individual objects, different contexts are kept separate and system configuration is simplified. Thus, the role object pattern must be utilized in the enterprise framework.

3.2. Adequacy

Framework adequacy relates to the goodness of fit of the framework. A framework must not be so general that it is overkill for the intended application. A one-size-fits-all approach introduces unnecessary risk. For example, a framework for industrial packaging may be well suited for applications in packaging of food products or children's toys, but may not be well suited for packaging of pharmaceuticals. This is due to the numerous differences in data collection and modeling between one industry and another. Furthermore, the level of effort to extend or tailor a framework is much greater if the framework is very general.

On the other hand, it is important that a framework is not too narrowly focused. A narrowly focused framework may not be sufficient to satisfy the functional requirements of the target application. The most suitable frameworks for a particular application domain reflect a mature, yet narrow focus on a particular problem or group of problems. We now identify a variety of measures that assist an organization in assessing adequacy when developing or selecting an enterprise framework.

3.2.1. Descriptive adequacy

Descriptive adequacy refers to the ability to visualize and monitor objects in the framework. Every defined object should be browseable, allowing the user to view the structure of an object and its state at a particular point in time. This requires understanding and extracting metadata about objects that will be used to build a visual model of objects and their configurations. This visual model is domain dependent—that is, it is based on domain data and objects' meta-data. Descriptive adequacy requires that all of the knowledge representation is visual.

3.2.2. Logical adequacy

Logical adequacy refers to the representation tools that describe the framework components' behavior, roles, and responsibilities.

3.2.3. Synthesis adequacy

Synthesis adequacy refers to an integrated problem resolution methodology, or built-in trouble-shooting tools. Built-in trouble-shooting tools are very important in managing complex distributed systems, because there are typically many potential points of failure.

3.2.4. Analysis adequacy

Analysis adequacy refers to integrated validation and verification tools. With built-in validation and verification, the process of maintenance and regression testing can be streamlined and the cost of validation and verification is minimized.

3.2.5. Blueprint adequacy

Blueprint adequacy refers to the modeling features that provide for integrated system specifications. Integrated system specifications are important because they facilitate the extensibility of the system. An integrated blueprint for an enterprise framework should clearly identify the hot-spots and frozen-spots in the framework.

Blueprint adequacy also refers to the system documentation. Documentation is the key to the success of the framework. It is the single most distinguishing characteristic of a high-quality software product. Many mature frameworks are lacking in one or more of the qualifications described above. Mature documentation ensures that a framework gets used. In addition, high-quality documentation ensures that design and implementation standards are reflected in all of the application content built with the framework. Therefore, mature framework documentation ensures reuse and maintainability.

Framework providers are challenged to deliver mature documentation with their product. Framework documentation must describe the purpose of the framework, how to use the framework, and the detailed design of the framework. In addition, Enterprise frameworks include concise documentation of the framework architecture, configuration and development tools, and object and API references. These documents are best deployed online and employ a standard hypertext presentation. Furthermore, mature framework documentation provides references and links to numerous examples including sample source code, and templates to be used in developing specific application components.

In general, the application developer need not understand all aspects of the framework, but should be able to search the framework documentation for patterns that reveal techniques for solving a specific class of problems using the framework.

Framework documentation must be extensible, allowing the framework team to insert details about framework extensions directly into the framework document library. In addition, the framework documentation should be closely tied to the modeling tools provided within the framework, so that documented analysis and design elements are easily translated into functional models.

3.2.6. Epistemological adequacy

Epistemological adequacy refers to tools for representing objects in the real world. There are two ways to view the world based on simplicity: (1) a perfect but simple view—the world is represented in this view as an ideal environment; and (2) a faithful, but complex and detailed view—the world is represented as an ultimate reality. There are also two ways to view an organization: (1) with a flat and single view and (2) with layered and multiple views. It is very obvious that most modeling techniques, such as the unified modeling language (UML) and the object-modeling technique (OMT), model the world as an ideal environment with a flat or single view of itself. Nevertheless, successful enterprise frameworks have made great leaps in representing objects in the real world and in providing the necessary tools to alter these objects as required by the business.

3.2.7. Notational adequacy

Notational adequacy refers to the presentation constructs and the impact these presentation constructs have on the operation of the system as well as on the ease of modification.

Among other concerns, today, notational adequacy increasingly implies that the framework is Web and e-business ready. As the Internet is used increasingly for commerce, enterprises will find ways to work more effectively through electronic data transfer and through online transactions over either identical or disparate frameworks. The result is a faster response to customers' needs, and a faster time to market.

3.2.8. Procedural adequacy

Procedural adequacy refers to recognition and search capabilities.

3.2.9. Contractual adequacy

Contractual adequacy refers to the client tools for representing the system behavior.

3.2.10. Scalable adequacy

Scalable adequacy relates to the constructs and tools supporting partitioning, composition, security, and access control. Today, scalability also implies a mature model for object distribution. Therefore, an important characteristic of enterprise frameworks is that they provide an integrated model for distributed objects, such as:

- object request brokering;
- distributed message passing;
- remote procedure calls/remote method invocation.

The distribution model must support transparent communication between objects over a distributed computing environment. In this way the distribution model supports scalability; as more users (clients) are added to the system, additional server instances can be added to improve throughput.

Support integration of multiple application frameworks and legacy components. An enterprise framework is, by definition, the cornerstone of a system architecture. Therefore, it must provide the structure and tools for easy integration of multiple application frameworks and legacy components. Several problems [12] are encountered when integrating multiple frameworks. It is important that framework developers consider each of these issues and provide tools to reduce the impact of these problems. The delivered framework should provide mechanisms for dealing with the more common and more challenging integration problems. An open API provides a ready means of integrating best-of-breed applications to the framework. The API allows these integrated applications to participate within the framework and provides a means of exposing the implied or actual object models of integrated applications to the framework. However, an open API is only a part of the solution to the problem of integrating multiple frameworks.

Mature enterprise frameworks reflect an understanding of integration problems. In addition, they will provide recommended solutions to these integration problems. Framework providers must possess a working knowledge of the business issues that fall outside of the framework domain and design frameworks with an awareness of the types of applications that will be integrated in the overall architecture.

Platform independence or portability. Platform independence and portability ensure that the framework supports all of the platforms in use by an organization. In addition, platform independence is important, because it ensures that other applications can interact with applications built from the framework.

Platform independence is closely related to the concern for open APIs and support for distributed objects. However, it is not sufficient to specify that frameworks should be open or should provide support for distributed computing. Platform independence is essential for framework developers working in large organizations supporting multiple platforms. Generally, the framework team may be required to provide support for the major platforms operating in an organization. Thus failing to provide platform independence may diminish the scope of use and acceptance of the framework. Platform independence is important for framework vendors; if a framework is not open, it may result in lost sales opportunities when customers are not willing to be limited to the platforms supported by the framework.

3.2.11. Administrative adequacy

Administrative adequacy refers to the tools for modeling the deployed system's performance, reliability, and administrative characteristics, and to the actual tools for administering the system. Administrative adequacy also considers the availability of install set builders, start and stop procedures or scripts, integrated database management capabilities, archiving, fail-over mechanisms, etc.

Modeling is the key to the success of frameworks. If the business models are as simple as possible, and not tied to any particular problem, they can be used to solve problems that were not envisioned when the framework was built.

4. ISSUES IN BUILDING ENTERPRISE FRAMEWORKS

As we mentioned earlier, enterprise frameworks are a special kind of application framework that differ from 'traditional frameworks' both in terms of scale and focus. In terms of focus, we stress the equal importance afforded to architecture and the computational infrastructure, on the one hand, and domain functionality on the other. We start by discussing software packaging issues (Section 4.1). Process and lifecycle issues are discussed in Section 4.2.

4.1. Packaging issues

Enterprise frameworks are information system templates that address the needs of various organizations within a particular domain. As such, they have to be customizable along the organizational, business, and technical dimensions of the host organization. Accommodating variations along any of the dimensions poses a number of technical challenges. This section discusses those challenges, which we grouped as follows:

1. instantiate domain semantics for the specifics of an organization (Section 4.1.1);
2. separate the computational aspects from domain aspects (Section 4.1.2); and
3. accommodate heterogeneous architectures or enterprise frameworks that include legacy components (Section 4.1.3).

4.1.1. Instantiating domain semantics for an organization

Traditionally, in business applications, when we think of the *application domain* we think of industries, such as banking, financial services, health care, telecommunications, and so forth. In order to support business to business applications, the information systems implemented by enterprises in each industry need to have some common ground to be able to exchange information. A number of earlier initiatives have thus adopted an 'industry' interpretation of domains, including IBM's SanFrancisco project and Insurance Institute, OMG's CORBA Domains, etc. However, if we go one level down in each industry, we realize that there are a few similarities beyond a common nomenclature.

When we think of health care, we think of health providers, outpatient care, drug information systems, patient monitoring applications, etc. How about medical equipment manufacturing, hospital facilities management, laundry services, catering, etc.? Catering services in hospitals have more in common with catering for prisons than with drug supply to hospitals[¶]. If we were to build an information system for both, we cannot use the entity **Patient** or **Inmate**. We might use the term **Beneficiary** or **Customer** of the catering service, i.e. move towards a more *business process view* of the functionality, rather than a *market domain* view of the domain. 'Renting TVs' has more in common with 'renting computers' than with 'manufacturing TVs'. For rental, both TVs have a purchase price to the leaser, a monthly cost to the renter, an amortization period, a deductible amount for repairs, etc. For manufacturing, the two TV entities will probably simply share the name; and using things such as subclassing for customization will do very little.

Consequently, to build effective enterprise frameworks, the components need to be *business process components*, and the customization for a given organization should involve two orthogonal steps:

1. specializing the *business process view* to the specific process of the organization (e.g. a general lease versus a lease to own); and
2. mapping the custom process view to application (enterprise) specific entities.

There is an increasing move towards this view of things [13,14] which may be seen as the natural extension of analysis patterns [15], for the optimists, or a resurgence of some of the earlier valuable—but non-fashionable—work on information engineering such as [16,17].

Practically, there are a number of ways of implementing this two-step process. We will be content to mention two variants. The reader can explore others (see also [18]). The first variant relies on generation techniques: the business process components are implemented as templates, which can be instantiated

---

[¶] The senior author of this paper has provided consultancy for a company that caters to hospitals and prisons, where dietary concerns and delivery modes are fairly similar.

by replacing the process roles by domain entities. For example, we can represent the object of a rental service as follows:

```
ROLE $RENT_OBJECT$ {
   ATTRIBUTE Hashable $BUSINESS_KEY$;
   ATTRIBUTE float $ACQUISITION_PRICE$;
   ATTRIBUTE Date $ACQUISITION_DATE$;
   ATTRIBUTE float amortizationRate;

   FUNCTION float getResidualValueAsOf(Date aDate) {
        int numYears = aDate.year() - $ACQUISITION_DATE$.year();
        float residualValue = $ACQUISITION_PRICE$* (1 - ...);
        return residualValue;
   }

   ALLOW OTHER_ATTRIBUTES;

   ALLOW_VALIDATE OTHER_FUNCTIONS;
}
```

In this imaginary Java-like macro language, things (attributes, functions) that are supposed to be replaced/substituted in the customization process (‘macrovariables’) are of the form $<identifier>$; the others are to be reproduced verbatim. This particular template supports the addition of other attributes and functions, but validates the user functions against utility functions generated by the processor. This seems to be the general approach used by some ERP systems, which map a template conceptual model on the entities of the organization, through a mapping of the attributes and relationships needed by the ERP systems, to attributes and relationships embodied in the specific domain.

The second variant may be schematized as follows. Here, **Rent_Object** is an actual class, which delegates to a domain-specific entity for things such as the identifier:

```
class Rent_object {
   private Rentable businessObject;

   public Hashable getBusinessKey() {
        return businessObject.getId();
   }

   public float getAcquisitionPrice() {
        return businessObject.getAcquisitionPrice ();
   }

   public Date getAcquisitionDate() {
        return businessObject.getAcquisitionDate();
   }
```

```
   private float amortizationRate;

   public float getResidualValueAsOf(Date aDate) {
        int numYears = aDate.year() - getAcquisitionDate().year();
        float residualValue = getAcquisitionPrice()* (1 - ...);
        return residualValue;
   }
}

interface Rentable {
   public Hashable getId();
   public float getAcquisitionPrice();
   public Date getAcquisitionDate();
}
```

In this case, an actual concrete class (**Rent_Object**) represents the rental object. The actual business object (a TV or a computer or an RV or a paddleboat at the local park) is represented by the interface **Rentable**. If a class **PaddleBoat** existed before bringing in this framework (legacy code), we could edit it or subclass it to support **Rentable**.

This second approach is similar to the approach proposed by Van Hilst and Notkin for implementing role models [19][‖]. Among other things, it allows us to implement several business roles that refer to the same business entity, without (or little) interference with the other roles, or with the legacy code[**].

These are but two points in the solution space for the general problem of instantiating business process components for domain entities. Most solutions combine subclassing and generation, even for moderately sized OO application frameworks.

4.1.2. *Separation of domain aspects from computational aspects*

Typically in software, the development lifecycle of software artefacts comprises stages similar to requirements, specifications, design, and implementation. Software architecture is a peculiar artefact in the sense that it cuts across several other software artefacts—namely, all the components of the system—and, as such, its lifecycle will have to be intertwined with the lifecycles of the other artefacts. We would benefit from separating the *artefact* and its *development lifecycle* from the other artefacts and their development lifecycles. Broadly speaking, the architecture of a particular software system is the juxtaposition of domain-specific aspects, in terms of domain functionalities, and domain-independent aspects, inherent in the architectural style that was adopted, and embodied in the interaction infrastructure of the architecture. We would benefit from keeping these two aspects independent as long as possible, and binding them together, as late as possible. However, separating

---

[‖] Van Hilst and Notkin's approach is actually a cross between the two variants shown here: they used C++ templates to implement role components.

[**] However, like most delegation-based approaches, it may suffer from the 'self problem' [18].

these two aspects is fairly difficult, and is the key to effective reuse of architectures. While it is generally possible to talk about the architecture in terms of requirements (what properties it must satisfy), and in terms of specifications (what style to use), it gets harder to talk about the 'design' or 'implementation' of an architecture without talking about the design and implementation of the domain-specific components it contains.

As it turns out, depending on the architectural style, *designing* and *implementing* the architecture can be more or less complex, and thus more or less worth separating from the design and implementation of the functional components. If we are choosing a main-program-and-subroutine-call style, then there is not much to do to 'realize' the architecture, beyond developing the components themselves: both the packaging of the components (subroutines, functions) and the communication infrastructure (subroutine call) are provided by the host programming language and the runtime system. If we are implementing a publish and subscribe architecture, then we have more things to do:

- designing the API of the components; for example, we could have a single event handling function that dispatches internally to various functions, or have several event handling functions, one per event type;
- designing the system's event manager and the dispatcher that manages the 'subscriptions' of the components, and handles the events they generate.

We could go beyond design and actually *implement* the computational infrastructure of the architecture, for example by:

- writing abstract classes that implement the event handling API of the components, and making sure that system components inherit from those classes; and
- coding the actual event manager.

As we can see from this example, depending on how we *design* the architecture, we may be able, in some cases, to *implement the computational infrastructure before any component is actually implemented*, thereby separating the domain-specific aspects from the computational aspects of the architecture. Figures 6 and 7 illustrate two possible lifecycle paths for implementing a given architecture, resulting in more or less reuse. In Figure 6, we implemented an application-independent (and thus reusable) computational infrastructure.

In Figure 7, we had to (or chose to) bind the analysis model to the architectural style before we did any architecture design/implementation. In this case, the resulting computational infrastructure is specific to the application at hand, or possibly, to applications within the same business domain.

In order to separate domain functionalities from the architecture, we need to do three things: (1) implement as much of the computational infrastructure as we can, without referring to any domain specifics; (2) implement as much of the domain functionalities as we can, without referring to a specific architecture; and (3) write as little glue code as we have to, to bind the two together. As we illustrated above, we can achieve the first goal through a combination of a good choice of the architectural style, and a judicious use of extension and specialization mechanisms. We now look at the second and third steps.

Selecting an architecture for a component embodies decisions such as selecting:

- a computational model (event-based system versus procedure call, sequential versus concurrent, synchronous versus asynchronous, etc.);

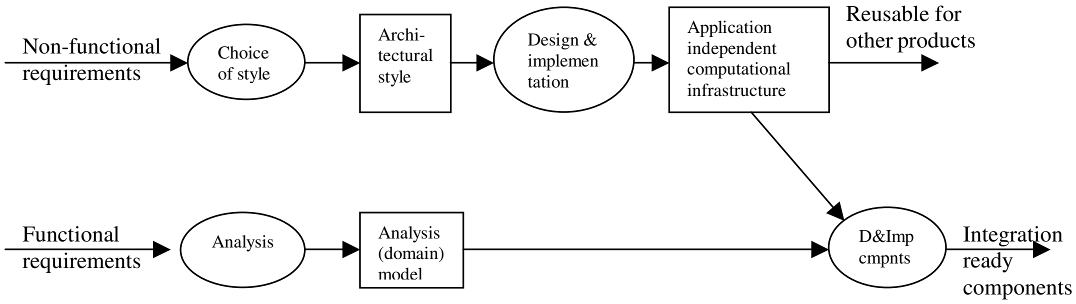

Figure 6. Implementing a (functional-) domain-independent computational infrastructure.

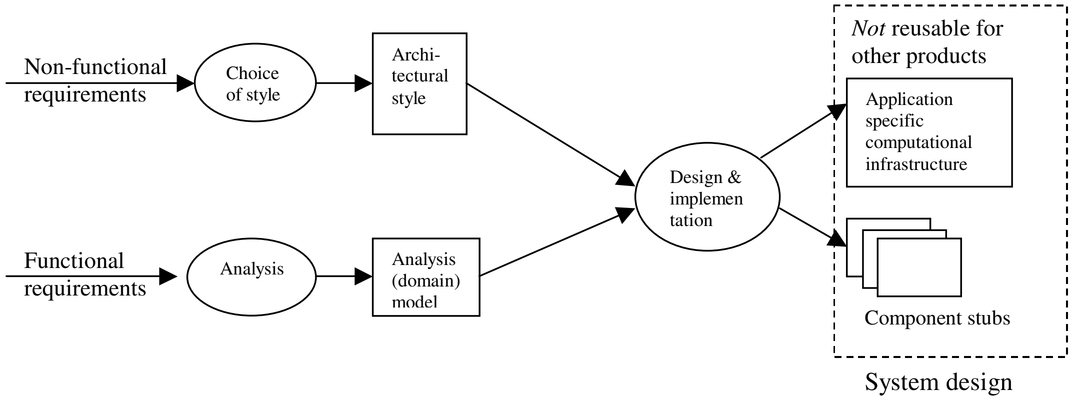

Figure 7. Binding the computational infrastructure to the analysis model early on.

- a set of system services to be supported (persistence, logging, error-recovery, etc.);
- the way in which those services are rendered.

Isolating each one of these decisions involves a set of design and packaging techniques that are more or less successful at achieving complete separation [20]. With regard to isolating the computational model, a first level of separation can be achieved using computational reflection techniques, either the ones built into languages (e.g. Java), or by using metamodeling patterns such as the ones discussed in [18] or [21]. Alternatively, we could use a virtual machine architectural style (see, e.g., [3]), which yields the greatest flexibility, but requires more development, and may suffer from performance problems.

Isolating the set of architectural services to be supported may be implemented using any of several techniques that aim at the separation of concerns by seeking modularization boundaries that are finer than the traditional class and method boundaries. One such approach is subject-oriented programming, which views OO applications as the composition of several partial views (*subjects*) of the same class hierarchies. Subject-oriented programming enables developers to develop applications incrementally by integrating new functions as they become available, without modifying the existing code [22,23,24]. Aspect-oriented programming enables us to separate non-functional aspects that cut-across several classes (e.g., logging, error handling, synchronization) from those functional aspects that are embodied in the OO core of applications [25]; an *aspect weaver* compiles *aspects* (some sort of high-level macros) into regular Java classes to produce aspect-enabled programs. We have developed our own brand of separation of concerns that we called *view-oriented programming* which embodies the functional aspects of a class in separate *views* that have their own state and behavior [26]. The novelty of our approach is the generation of views as instantiations of so-called *viewpoints* for domain-specific entities [27]. *Viewpoints* were meant as *generic entities that embody business processes*, and appear to be more suited to the problems discussed in Section 4.1.1, but may also be used to implement some stand-alone services and add them to objects as desired.

Isolating the way in which architectural services are implemented may be achieved using any one of a number of design patterns (e.g., Adapter or Bridge), or any one of the separation of concerns techniques, especially subject-oriented programming [20].

Notwithstanding the technical merits of the modularization methods discussed above, most of these methods are good at *integrating concerns that were developed separately* but can do little for applications into which the concerns have been hopelessly intertwined. For example, aspect-oriented programming enables us to integrate specific aspects into a code base without corrupting that code base, but cannot do much for code that was already written into a specific architecture. The same is true for subject-oriented programming and view-oriented programming. The methods based on design patterns and computational reflection have the advantage of being ‘backward compatible’ in that they allow us to connect APIs that are already written and that do not match. They can do little, however, for the control paradigm. In summary, most of these techniques hold the greatest promise for systems yet to be built.

4.1.3. Handling heterogeneous architectures

Issues. Enterprise frameworks typically encompass a broad area of functionalities that covers a set of departmental systems, including legacy systems that may use obsolete technologies. Figure 8 shows the result of integrating a departmental system into the overall enterprise architecture. For example, departmental systems 1 and 2 communicate via Java RMI (departmental computational infrastructure), and CORBA was chosen as the enterprise-wide computational infrastructure. In this case, a bridge is needed between Java RMI and CORBA (and, thankfully, standards exist for such a bridge). This shows that system 3 was built directly into the computational infrastructure of the enterprise, for example, it was built directly into CORBA. System 4 is stand-alone, uses a different architecture (e.g., COM), and needs a bridge to the enterprise-wide infrastructure.

The situation in this example, while far from idyllic, does not come close to some of the issues that arise when we try to integrate 20 and 30 year old legacy systems into a modern backbone (e.g., CORBA). Handling structural aspects such as data and API mismatch is relatively easy, and the techniques discussed in Section 4.1.2 can do much to alleviate the problems. The major problems reside in differences in the control paradigm [28]. If the components of the system need frequent two-way

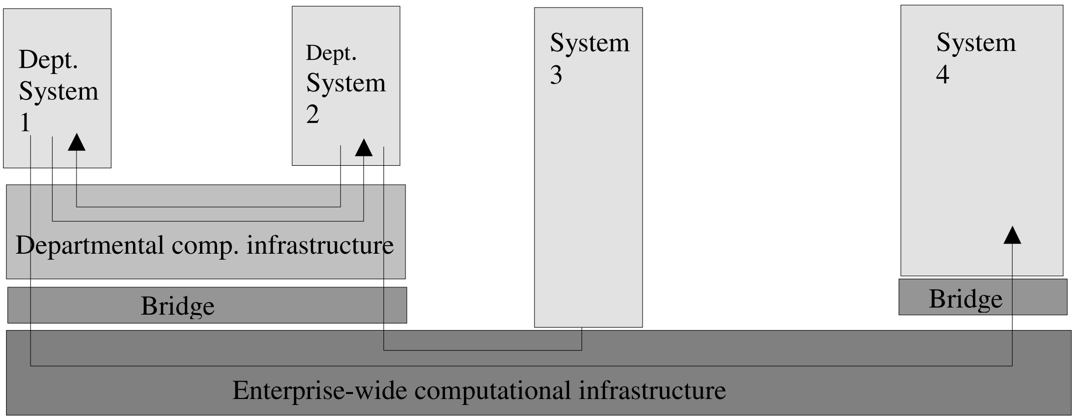

Figure 8. Integrating heterogeneous departmental systems into an enterprise framework.

interaction involving more than two components, then we can potentially have some serious integration problems. Practically, this means writing a significant amount of glue code and *ad hoc* modules that manage specific two-way interactions.

4.2. Process issues

Viewed as reusable software artefacts, frameworks have requirements, analysis level models, designs, and realizations; they address functional domains in a way that satisfies a number of design criteria (qualities), namely, flexibility and reuse. The first step in building frameworks is to identify the functional requirements. Most researchers and practitioners agree that this must be the first step in framework development. Regarding the remaining steps, there are two schools of thought, which we may caricature as follows.

(1) The top–down analytical school: we develop a framework by performing a domain engineering process, starting with domain analysis, then design, then realization (see, e.g., [29]),
(2) The bottom–up synthetic school: we start by developing an application within the domain of the framework, and then introduce variabilities into it (see, e.g., [30]).

The two authors cited will probably object to this characterization as too reductionist, which is probably true. We illustrate the two schools in this section.

4.2.1. Frameworks as products of domain engineering

Domain engineering is a set of activities aimed at developing reusable artefacts within the realm of a functional domain. Different domain engineering methods may have different deliverables or propose different heuristics, but they all rely on:

- performing some sort of commonality/variability analysis to identify those aspects that are common to all applications within the domain, and those aspects that distinguish one application from another;
- deriving increasingly concrete descriptions for these aspects/concerns/components, starting with analysis level descriptions down to code.

Existing domain engineering methods recognize that designing and implementing a 'domain' has to be incremental, for at least two reasons: (1) to spread out the investment in resources over a tolerable period of time; and (2) to road test the architecture of the domain (or framework) first, before developing a lot of components into that architecture. The question then becomes which parts to start with? And the answer from most experts is 'start with those aspects of the domain that have an influence on the architecture' (see, e.g., [31,32]). Our question might then be 'How do you identify those aspects that most influence the architecture?'. We find it useful to think of the architecture in terms of two layers, a computational layer and a functional layer. The computational layer underlies all of the functions. The functional layer may be more partitionable than the computational layer, and thus may lend itself better to incremental development. What the experts are saying is: start with those functions that are likely to require *most* or *all* of the computational infrastructure so that the major design trade-offs will be addressed with the first increment; new functions might be added later, but they should not require a new computational infrastructure, or the new infrastructure does not depend on the existing one.

For example, we want to develop a Web-enabled banking framework. In terms of computational infrastructure, mission-critical Web-enabled *business* (banking or otherwise) applications will require:

- distribution: the application logic should be location transparent, which means the use or development of a distribution infrastructure;
- security: communication across the network must be secure, which requires the handling of users, authorizations, certification, encryption, and the like;
- transaction services: which means the use of transaction monitors to ensure the integrity of distributed transactions; and
- recoverability: which means distributed logging, mirror sites, and the like.

Because all of these are related, if we want to develop a banking framework incrementally, the first increment should contain the smallest set of functions that will require all four computational services.

The literature abounds with recent experiences with framework development in both industry and research and development labs (see [33]), and a good portion of it falls into this school of thought, namely, viewing framework development as a planned domain engineering process where the framework starts taking shape during the domain analysis phase. Example approaches include [29,32], and several approaches described in [34,35].

4.2.2. Frameworks as planned by-products of application development

The idea here is that we *grow* frameworks out of subsets of applications within the targeted framework domain. In this approach, we still need to elicit the requirements of the framework. Instead of building it from the top down, we start building applications from the domain, and then start introducing variations in those parts of the applications that concern the domain of interest.

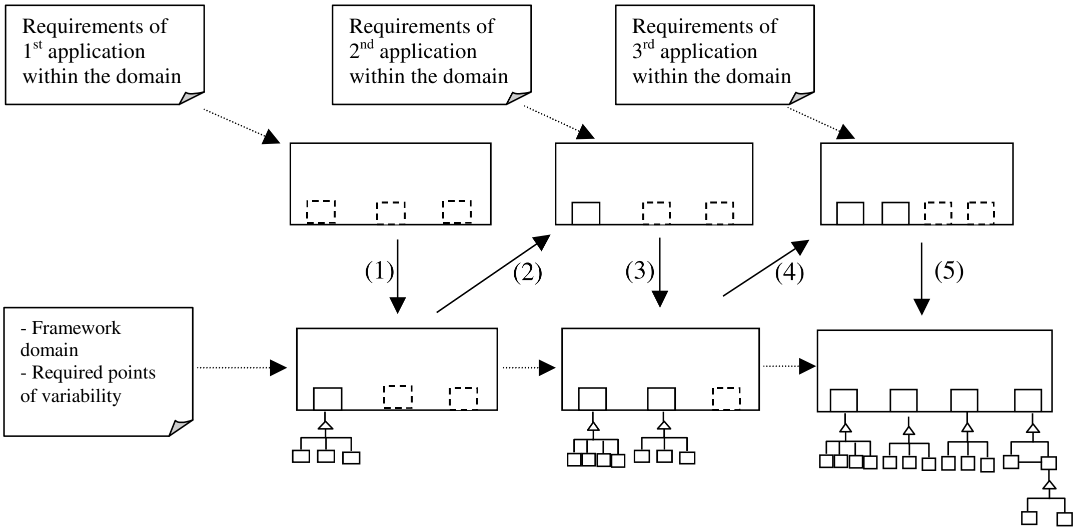

Figure 9. Incremental framework development by hot-spot design.

A good number of frameworks have originated from research and development (R&D) laboratories, and fall into this category, for many reasons. First, the work in R&D laboratories is driven by creativity, trial and error, and risk taking; these R&D 'values' are often in contradiction with notions of process, deliverables, documentation, and the like. Second, few R&D laboratories are interested in investigating the process of building frameworks *as a subject of study*[††], and thus studies into the process of building frameworks often come as a *post hoc* formalization of *ad hoc* development. These studies have the advantage of embodying true and proven design heuristics.

A representative example of these approaches is *hot-spot driven development* as advocated by Pree [36] and Schmid [30]. A framework 'hot-spot' is a point of variability within a framework. According to Pree, two applications within the application domain of a framework will differ by the binding of at least one hot-spot. The idea here is that we can identify those hot-spots early on in the process, but we do not necessarily develop them in the first iteration: the first iteration will solve a specific problem; later iterations will design variability into the identified hot-spots. Figure 9 illustrates this idea.

The process starts with an identification of the domain of the framework, and requirements for a first application. That implementation is later evolved into a first iteration of the framework where a first point of variability (hot-spot) is fully developed; probably that aspect that most often changes from one

---

[††] There are a few notable exceptions, of course, like the SEI and the Software Engineering Laboratory affiliated with the University of Maryland, but then these are the places that came up with top–down approaches such as FODA and OO variations thereof.

application to another (step 1). Designing a hot-spot means changing a specific binding of a point of variation (e.g., a terminal logger) into [30]:

(1) an abstract description (a generalization) of the aspect of interest; and
(2) a number of concrete realizations (including the existing one).

Of course, experience with the framework may identify unanticipated points of variations, which will be implemented along the way (steps (4) and (5)).

In a typical increment, a concrete class at iteration n is replaced at iteration $n+1$ by an abstract class or an interface (depending on the language) and concrete subclasses. The point of variation (concrete class) is called the *hot-spot*. Note that a concrete class that is used at a point of variation may be replaced by several collaborating classes, only one of which may be abstract. In other cases, variations may be handled by parameters.

4.2.3. A lifecycle for enterprise frameworks

As mentioned earlier, enterprise frameworks are special cases of application frameworks. As such, there are potentially two competing lifecycles for developing enterprise frameworks; (i) the top–down approach in which the framework is the intentional and immediate deliverable of a development process (domain engineering); and (ii) the bottom up approach in which the framework emerges out of incremental generalizations of points of variation (*hot-spot driven design*). Which is more appropriate for enterprise frameworks, if any?

There are two competing considerations here. On the one hand, because of the scale of the effort required to develop an enterprise framework, it seems risky to adopt an entirely top–down approach, where much effort goes into developing software artefacts whose usefulness and cost-effectiveness are not guaranteed; only actual projects using the framework will validate the architecture and the components. On the other hand, it seems wasteful to arrive at an architecture through local iterations (hot-spot driven), and the consistency of the resulting overall architecture may suffer, as different hot-spots may adopt different composition/interaction mechanisms. Further, having a catalogue of proven architectural styles with documented features and well-known profiles, there may not be a need for 'emerging' an architecture through iteration.

We argue for a hybrid development lifecycle for enterprise frameworks. The architectural aspects of the framework should be developed in a centralized top–down fashion, working from requirements, down to implementation. Care must then be taken to separate the computational infrastructure aspects from the functional aspects. Depending on the architectural style used, this can be more or less difficult, as illustrated by the example of publish-and-subscribe architectures in Section 2.1—in fact, this will often be one of the criteria for choosing an architectural style. If the style does not lend itself easily to this separation, additional design techniques and artefacts may be used to this end, such as reliance on reflection and the like (see, e.g., [21]). The functional aspects of the architecture will be developed in an incremental and iterative fashion, as shown in Figure 9 above. The idea here is that the points of variation will be functional in nature. Figure 10 illustrates this lifecycle.

The computational infrastructure is developed in the 'first iteration', and follows the regular top–down approach, going from requirements, to choosing an architectural style (analysis), to selecting the corresponding connectors (design), to actually implementing the infrastructure. In addition to the

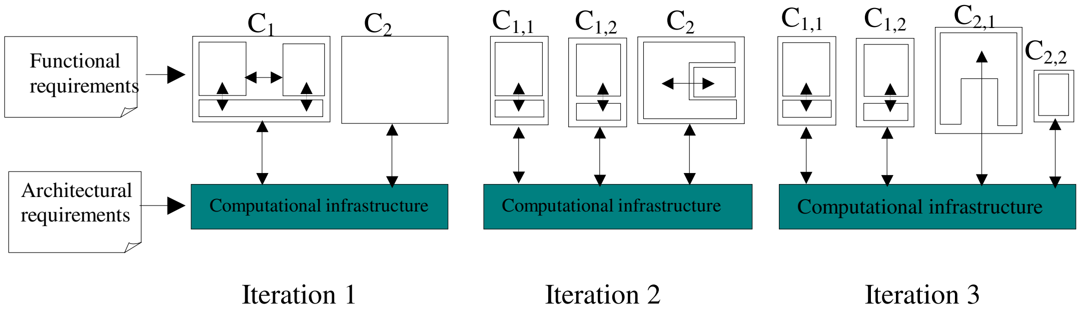

Figure 10. A lifecycle for enterprise frameworks.

computational infrastructure, we also develop some application components which may be coarse-grained, with a few degrees of freedom at first, which are refined and separated as we go through the successive iterations. We see here a component (C_1) that has its own internal structure and its own custom interaction mechanism. In the second iteration, the internals of that component are 'externalized', and its subcomponents ($C_{1,1}$ and $C_{1,2}$) now interact through the common computational infrastructure. For example, C_1 could represent a legacy system, with its own 'local architecture'. In the first iteration, a bridge API is written to connect it to the common infrastructure. Later iterations will externalize the subcomponents of that legacy system, providing greater degrees of freedom, and supporting the interchangeability of components.

5. ISSUES IN USING ENTERPRISE FRAMEWORKS

Enterprise frameworks are a powerful reuse and development tool with a high degree of complexity. Using an Enterprise framework raises some interesting questions. Which are the criteria to select the right framework. What is the process for transforming an enterprise framework into a concrete application? This section investigates these issues starting from an example in the manufacturing domain.

5.1. An example in the manufacturing domain

A manufacturing information system (MIS) consists of software tools and applications that process, handle, and control the business, logistic, and production activities of an enterprise. An information system can be considered as the nerve system of an enterprise. It facilitates the exchange of information among the constituent parties of the enterprise: customer offices, business units, departments, production cells, and distribution centres. It supports the co-operative work of managers, sellers, market analysts, administrative employees, engineers, machine operators, and workers. It allows the automation of a variety of functionalities that otherwise would require the direct intervention of human operators.

Developing an MIS requires deep domain knowledge in terms of enterprise business models and business processes. Business models and business processes describe an enterprise from the point of view of its mission. What is the enterprise's business? What does the enterprise produce and how? Which customers does it address? Which resources does it employ? How is it organized? Building accurate enterprise business and process models is extremely expensive both in terms of time and money. Nevertheless, it is an important step towards the construction of an effective and efficient manufacturing information system, which pays off in terms of resilience of the information system to changes in the business requirements.

The globalization of manufacturing organizations has led to the concept of a vital factory where different manufacturing operations are performed at geographically distributed locations. The manufacturing information system for such factories is usually a large distributed system composed of subsystems dynamically interacting with each other interconnected through a common communication medium such as the Internet. The typical tasks performed by these subsystems include: the management of the enterprise's knowledge base (product data, customers' records); the interaction with customers and providers (order management, product advertisement); the production process and product engineering (designing, prototyping); the control of the production system (planning, scheduling, monitoring); the supervision of the enterprise's resources (stores, machines, materials).

5.2. Issues in selecting an enterprise framework

At the early stages of an information system (e.g., MIS) development, the application developer has to answer the question of whether to buy an existing enterprise framework or build everything from scratch. Usually, the answer lies between the two extremes. In order to answer this question, the application developer should identify existing enterprise frameworks, and compare their specifications against the requirements of the application at hand.

If no existing framework matches exactly the application requirements, three solutions are possible.

(1) The application requirements are revised against the specification of available frameworks. According to Boehm [37], 'in the old process, system requirements drove capabilities. In the new process, capabilities will drive system requirements . . . it is not a requirement if you can't afford it'.
(2) The best existing framework is adapted to match the application requirements.
(3) The application is developed from scratch.

The second solution (the most common) implies the ability to identify the 'best' framework to adapt, that is, to measure the adequacy of a framework to the specific application's requirements [12].

Using an existing enterprise framework imposes a considerable up-front investment [38]. Therefore, while evaluating the adequacy of an enterprise framework, it is important to consider two goals: maximizing the initial reuse and minimizing the maintainability effort. The application developer has to consider not only application requirements but evolution requirements as well. Typical questions are:

- What kind of evolution does the application undergo?
- Which parts of the application are more stable?
- Which standards are available?

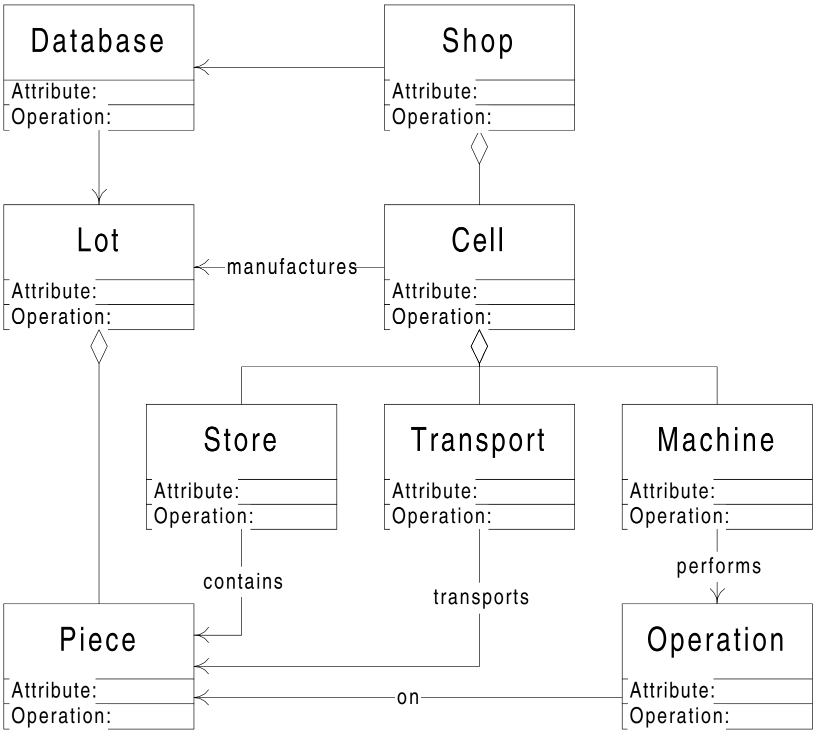

Figure 11. The hierarchical architecture of a FMS.

Let us consider again an example in the manufacturing domain. Flexible manufacturing systems (FMSs) follow an evolution, which is typically bottom–up. Very often, a new application builds on legacy systems. The architecture of this kind of systems consists of a hierarchy of component layers [39]. A higher-level component (e.g., a shop controller or a cell controller in Figure 11), acting as a client, integrates lower-level components, which act as servers (e.g., the cell controller, the machines, transports, and buffers). For example, a factory work cell is composed of a set of physical devices like machines, transports, and stores. The application that controls the work cell's activity integrates the software components that interface the physical devices.

Different vendors develop these components, without any prior knowledge of the environment in which they are to be embedded. A similar situation occurs when a set of work cells is integrated within a shop. In such systems, lower-level components are more stable than higher-level components, which undergo frequent reconfigurations according to rapidly evolving business requirements.

Major requirements of an enterprise framework for FMSs are the ability to support existing standards for stable components and the flexibility to adapt to rapidly changing business rules.

5.3. Issues in customizing an enterprise framework

Most MIS development is organized along the value-added principle. This divides the information system into a set of horizontal layers, each one built on top of another. This approach promotes separation of concerns and enforces modularity. Typically, today's MISs are subdivided into a system infrastructure layer, a business domain layer, and an application layer.

Enterprise frameworks offer a unified view to model and develop enterprise information systems at every level of the vertical decomposition from the system infrastructure to the final application through the enterprise's business model. The framework approach adopts the value-added principle as it supports the incremental transformation of an enterprise framework in a concrete application. Let us consider the customization of a MIS enterprise framework.

The *system infrastructure layer* (usually called middleware) consists of a collection of software components that offer basic services like communication between distributed applications, uniform access to heterogeneous resources, independence from processing platforms and distributed location, distributed component naming, service brokering and treading, and remote execution. Usually an enterprise framework extends a commercial middleware product, as it is more cost-effective for the framework developer to buy a system infrastructure rather than developing it from scratch. Examples of middleware systems are Microsoft DCOM, the implementations of OMG CORBA, and the Sun Enterprise Java Beans. An example of system infrastructure for real-time supervision and control systems that extends OMG CORBA can be found in [40]. The customization of the system infrastructure layer is usually *black-box*. It consists in the aggregation of the frameworks' elemental components (such as mechanisms for logical communication and concurrency) in which the whole application has to be written.

The *business domain layer* is the model of the enterprise's business and consists of software business objects, which directly map to real entities like product, order, customer, and provider. They are implemented on top of the system infrastructure layer. For example, customer order and product descriptors are recorded in a database; the customer and provider communicate through a common medium. The customization of the business domain layer is usually *white-box*. The basic components are intermediary classes, which by their very nature are fairly application independent, although they have been conceived bearing in mind a specific application domain. They are written using the elemental components of the framework, and they have to be specialized for each concrete application.

The *application layer* connects the business objects according to the information, control, and work flows of the enterprise's business processes. Examples of MIS business processes are production planning and scheduling, two critical processes that require a number of decision activities involved in managing flexible plants. These activities can be put into a hierarchy: from new product design and its links to manufacturing (level 1), one can proceed through the manufacturing/marketing interface (level 2) and medium term planning (level 3) down to detailed scheduling (level 4).

The customization of the application layer is usually *grey-box*. It consists of the interconnection of pluggable business components through the middleware system. These components are the result of the adoption of the framework for the development of more and more applications and in some cases they are commercial components-off-the-shelf (COTS). COTS products are designed to support a standard virtual interface, which consists of a set of rules for accessing the component's functionality in a homogeneous way, regardless of the execution platform, the programming language, and other specifics ([41]). The virtual interface is called the API. Typical COTS products for MISs are the

ERP, the Product Data Management (PDM), and the tools for computer aided manufacturing, process planning, and product design.

6. SUMMARY

Through our research, we have identified an emerging technology termed 'enterprise frameworks'. We believe that enterprise frameworks will be at the forefront of information systems technology for the next 10 years. We have described the roots of enterprise frameworks in application frameworks, and compared and contrasted enterprise frameworks with styles and architectures. In addition, we have raised a variety of issues where designers, developers, testers, and all other participants in enterprise computing should be informed. We have delineated the characteristics of enterprise frameworks that are critical measures of success in both the development and selection of enterprise frameworks. Finally, we have described the issues faced by the deployment and support staff as well as the end-users of enterprise frameworks.

REFERENCES

1. Butler G, Keller RK, Mili H. A framework for framework documentation. *ACM Computing Surveys* 2000; **32**(1), electronic symposium.
2. Mili H, Sahraoui H. Describing and using frameworks. *Building Application Frameworks—Object-Oriented Foundations of Framework Design*. Wiley: New York, 1999; 523–559.
3. Bass L, Clements P, Kazman R. *Software Architecture in Practice*. Addison-Wesley: Reading, MA, 1998.
4. Shaw M, Garlan D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
5. Dori D. Object-process analysis: Maintaining the balance between system structure and behavior. *Journal of Logic and Computation* 1995; **5**(2):227–249.
6. Dori D. Object-process analysis of computer integrated manufacturing documentation and inspection. *International Journal of Computer Integrated Manufacturing* 1996; **9**(5):339–353.
7. Dori D. Object-process methodology applied to modeling credit card transactions. *Journal of Database Management* 2001; **12**(1):2–12.
8. Dori D. *Object-Process Methodology—A Holistic Systems Paradigm*. Springer: Heidelberg, 2001.
9. Szyperski C. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley: Reading, MA, 1999.
10. Fayad M, Altman A. Introduction to software stability. *Communications of the ACM* 2001; **44**(9).
11. Fayad M. Accomplishing software stability. *Communications of the ACM* 2002; **45**(1).
12. Fayad M, Hamu D, Brugali D. Enterprise frameworks characteristics, criteria, and challenges. *Communications of the ACM* 2000; **43**(10).
13. Coad P, Lefebvre E. Enterprise component models in color. *Software Development* 1999; 39–48.
14. Coad P, Lefebvre E, De Luca J. *Java Modeling in Color with UML: Enterprise Components and Process*. Prentice-Hall: Englewood Cliffs, NJ, 1999.
15. Fowler M. *Analysis Patterns: Reusable Object Models*. Addison-Wesley: Reading, MA, 1997.
16. Carlson WM. Business Information Analysis and Integration Technique (BIAIT)—the new horizon. *Data Base* 1979; **10**(4):3–9.
17. Lefebvre É. How to improve MIS planning. *PhD Thesis*, University of Grenoble, 1996.
18. Mili H, Mili A, Yacoub S, Addy E. *Reuse-Based Software Engineering: Techniques, Organizations, and Controls*. Wiley: New York, 2001.
19. Van Hilst M, Notkin D. Using role components to implement collaboration-based designs. *Proceedings of OOPSLA'96*, San-Jose, CA, 6–10 October, 1996; 359–359.
20. Mili H, Mili A. Architectural frameworks. *Frameworks, Components, and Architectures*, Fayad ME, Garlan D, Pree W (eds.). To appear.
21. Mili H, Pachet F. Metamodeling for multidimensional reuse. *Proceedings of the Maghrebian Conference on Software Engineering and AI (MCSEAI'2000)*, Fès, Morocco, 1–3 November; 29–39.

22. Harrison W, Ossher H. Subject-oriented programming: A critique of pure objects. *Proceedings of OOPSLA'93*, Washington DC, October 1993; 411–428.
23. Ossher H, Kaplan M, Harrison W, Katz A, Kruskal V. Subject-oriented composition rules. *Proceedings of OOPSLA'95*, Austin, TX, 15–19 October, 1995; 235–250.
24. Tarr P, Ossher H, Harrison W, Sutton S Jr. N degrees of separation: Multi-dimensional separation of concerns. *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, 1999.
25. Kiczales G, Lamping J, Lopez C. Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland (*Lecture Notes in Computer Science*, vol. 1241). Springer: Berlin, 1997.
26. Mili H, Dargham J, Mili A, Cherkaoui O, Godin R. View programming for the decentralized development of OO programs. *Proceedings of TOOLS USA'99*, Santa-Barbara, CA, 1–5 August, 1999.
27. Mili H, Mcheick H, Dargham J, Sadou S. CorbaViews: Distributing objects with views. *Proceedings of AICCSA2001*, Beirut, Lebanon, 26–28 June 2001; 35–43.
28. Garlan D, Allen R, Ockerbloom J. Architectural mismatch: Why reuse is so hard. *IEEE Software* 1995; **12**(6):17–26.
29. Aksit M, Tekinerdogan B, Marcelloni F. Deriving frameworks from domain knowledge. *Building Application Frameworks—Object-Oriented Foundations of Framework Design*, Fayad, Schmidt, Johnson (eds.). Wiley: New York, 1999; 169–198.
30. Schmid HA. Framework design by systematic generalization. *Building Application Frameworks—Object-Oriented Foundations of Framework Design*, Fayad ME, Schmidt D, Johnson R (eds.). Wiley: New York, 2001; 353–378.
31. Kang K, Cohen S, Hess J, Novak W, Peterson A. Feature-oriented domain analysis (FODA) feasibility study. *Technical Report CMU/SEI-90-TR-021*. www.sei.cmu.edu/publications/documents/90.reports/90.tr.021.html.
32. Jacobson I, Griss M, Jonsson P. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley: Reading, MA, 1998.
33. Fayad M, Schmidt DC, Johnson R (eds.). *Implementing Application Frameworks: Object-Oriented Frameworks at Work*. Wiley, 1999.
34. Fayad M, Johnson R. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. Wiley: New York, 1999.
35. Fayad M, Johnson R. *Domain Specific Application Frameworks: Experience by Industry*. Wiley: New York, 1999.
36. Pree. 1999.
37. Boehm B *Software Engineering Economics*. Prentice-Hall: Englewood Cliffs, NJ, 1981.
38. Hamu D, Fayad ME. Achieving bottom line improvements with enterprise frameworks. *Communications of the ACM* 1998; **41**(8):110–113.
39. Brugali D, Menga G, Aarsten A. The framework lifespan. *Communications of the ACM* 1997; **40**(10):65–68.
40. Capobianchi R, Carcagno D, Coen-Porisini A, Mandrioli D, Morzenti A. A framework architecture for the development of new generation supervision and control systems. *Domain Specific Application Frameworks: Experience by Industry*, Fayad ME, Johnson R (eds.). Wiley: New York, 1999.
41. Hines JR. Software engineering. *IEEE Spectrum* 1996; January; 60–64.