Architectural Frameworks

Hafedh Mili[1], Ali Mili[2]
[1]Department of C. S., Univ. du Québec à Montréal, Canada, hafedh.mili@uqam.ca
[2]Department of C.S. & E.E., West Virginia University, USA, amili@csee.wvu.edu

Abstract

The *architecture* of a software system *is* the *inherent* organization of the software components that make up the system. Software architecture is *key* to software reuse. A *good* software architecture enables the maintainability of the system and the reusability and interchangeability of its components, and architecture is a key factor in the success of application frameworks of all kinds and sizes [Fayad et al., 1999]. Developing the architecture of a system, for long an *ad-hoc* activity intermingled with the development of the *components* of the system themselves, has only recently received the attention it deserves as a stand-alone development activity with its own deliverables [Garlan et al., 1996], [Shlaer & Mellor, 1997],[Bass et al.,1998]. As a software artefact, software architecture is worth documenting and reusing. However, because the realization of an architecture is often inseparable from the realization of the components, we can only reuse architectures at a fairly high level, and in a fragmented way. Viewing the architecture of a system as the binding of a domain-specific functional organization to a domain-independent computational model, we propose the concept of an *architectural framework* as an application framework whose purpose is to implement a computational model in a domain-independent way, enabling a more leveraged reuse of architectures, and thanks to the later binding of components to architectures, greater reuse of components as well. We first review the concepts of architecture, architectural style, and architectural qualities [Garlan & Shaw, 1996], [Bass et al.,1998]. Next, we present a lifecycle for developing the architecture of a system, and identify the concept of architectural frameworks. Section 4 discusses a set of techniques needed to implement them. We conclude in section 5.

1 Introduction

The *architecture* of a software system *is* the *inherent* organization of the software components that make it up. The reusability and maintainability of a software may be measured by the extent to which its components may be used in other systems, and the extent to which components from other systems may replace existing ones. Large-scale organized efforts at reuse have established that the greatest technical obstacle to reuse is lack of *usability* and not lack of *usefulness*. In particular, there is a lot of *useful* functionality out there in legacy systems, but most of it does not use the right control paradigm, or doesn't abide by the right interaction protocols, or simply, isn't implemented in the right programming language (see e.g. [Garlan et al., 1995], [Bass et al., 1998]). These properties of software components are *architectural* in nature. To ensure inter-operability of useful components, we need to enforce some sort of "architectural cohesion" *within* and *across* systems. Researchers in Pittsburgh (Carnegie Mellon and Software Engineering Institute) have given such cohesion a name, and it is called *architectural style*. An *architectural style* is a set of *principles* for organizing software modules into a complete software system [Garlan & Shaw, 1996], [Bass et al., 1998]. In addition to the intrinsic value of adopting *one* architectural style throughout an entire system, different styles enable us to attain different levels for such architectural properties as modifiability, reusability, performance, etc. [Bass et al., 1998].

A *good* architecture—with respect to a set of architectural properties—is hard to find because of the often contradictory requirements [Bass et al., 1998]. A good architecture is also hard to reuse, for several reasons. First, it is hard to define what is there to reuse. The architecture of a system is a *pervasive* property of the system that cuts through several components [Coplien, 1997], [Kerth & Cunningham, 1997]. We distinguish between the *functional architecture* of a system, which is a description of the main functions of the domain and of their interrelationships, and the *computational architecture*, which is the application-independent software organization of these components in terms of inter-component interfaces and interaction mechanisms [Shlaer & Mellor, 1997]. When we talk about reusing the architecture of a system, we could be talking about reusing the functional architecture, the computational architecture or both. If we remain within the same application domain, we can reuse both the functional architecture and the computational architecture without having to worry about separating the two. This is generally the case for product-line engineering or enterprise frameworks where we reuse the underlying computational infrastructure and the domain components. However, we might run into trouble if we try to reuse functional components in a different application that uses a different computational architecture because the code that implements the computational architecture is typically embedded in the component itself (see e.g. [Garlan et al., 1995]). For the same reason, we cannot reuse the implementation of the computational architecture across functional domains, and yet, that implementation generally represents a significant portion of the development effort and complexity.

We define *architectural frameworks* as a special kind of application frameworks that embody computational architectures in a domain independent way. As such, they may be considered as domain independent *implementations of architectural styles*. Some *middleware* products (e.g. EJB platforms) are example architectural frameworks. Architectural frameworks are hard to build, but can dramatically enhance development productivity and decrease product complexity. They also lead to much greater reuse of the software components that are deployed using them.

In the next section, we provide an overview of the notion of architecture, architectural styles, and architectural frameworks. Much of the contents of this first section are based on the book by Mary Shaw and David Garlan of Carnegie Mellon University [Garlan & Shaw, 1996], and the book by Bass, Clements, and Kazman, of the Software Engineering Institute [Bass et al., 1998]. Section 3 deals with the "engineering" of development of architectural frameworks. First, we propose an architecture of architectural frameworks. Next, we discuss a set of techniques required to build them. Finally, we provide a brief overview of some architectural frameworks. Section 4 discusses issues related to the development *with* architectural frameworks. First, we discuss the effect of architectural frameworks on development lifecycles, and then, discuss ways in which component implementations may be generated. We discuss directions for future research in section 5.

2 Architectures, with style

2.1. Architecture

The very definition of architecture has been a subject of debate for some time in the software engineering community, with some authors questioning the appropriateness of the term to software, in the same way that "engineering" was deemed an inappropriate characterization of what we do [Coplien, 1999]. Bass et al. define architecture as follows [Bass et al., 1998]:

> "*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*".

In this definition, software architecture is seen as the high-level design of a software system in terms of software components (modules, subsystems, processes), their external properties (API, run-time behaviour), and their inter-relationships [Garlan & Perry, 1995]. There is some confusion in the literature as to whether the "architecture" of a software system refers to the specific components of the system and the relationships between them—we refer to this as the *extensional interpretation*—or to the *emergent patterns* from the descriptions of the individual components and their inter-relationships—we refer to this as the *intensional interpretation*. One consequence of this confusion is disagreement as to whether the architecture of a software system embodies domain-specific information or not (see e.g. [Shlaer & Mellor, 1997] and [Bass et al., 1998]). We liken the difference to that between a *class diagram* where associations represent possible associations between instances of the classes (*intensional interpretation*), and an *instance diagram* that represents actual objects and the corresponding relationships between them (*extensional interpretation*). For our purposes, the term "software architecture" refers to the *extensional interpretation*; architectural *styles*, discussed in section 2.3, correspond to the *intensional interpretation*.

Because software tends to be very complex, we need several views of the software that reflect different properties of the software components and different relationships between them, or different decompositions of the software altogether. Depending on the kind of components and the kind of relationships between them, we have different structures to describe the software, including *module structure* where the nodes represent software modules (packages, classes), and links represent dependency and containment links, the *conceptual structure*, where the components represent (business) functional units of the system, and the relationships between them represent data flow between them, the *process structure* in which the components represent processes or threads, and the relationships between them represent relationships such as "synchronizes with", "can't run with", "pre-empts", "rendez-vous with", etc. Each one of these structures reflects one aspect of the system, and serves different purposes. The module structure may be used to identify dependencies within the parts of the system, and break down system development into as many independent parallel tracks as possible [Coplien, 1997], [Bass et al., 1998], [Herbsleb & Grinter, 1999]. The various structures that describe a software system reflect different concerns but ultimately overlap as the same software entities crosscut different structures. 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 [Bass et al., 1998].

2.2. Quality attributes of architectures

Each software system *has an architecture*, which is the actual organization of the software system as it stands, regardless of how carelessly thought out it was. However, we are able to intuitively recognize *good architectures* from obviously *bad architectures*. A number of authors have identified architecture-related attributes that help assess the quality of the process (see e.g. [Herbsleb & Grinter, 1999]) as well as the product ([Garlan & Perry, 1995], [Bass et al., 1998]). Bass et al. draw a number of distinctions between quality attributes, including the following important one:

- Quality attributes of the *architecture* as a *deliverable* in its own right, and
- Quality attributes of the *software product* as a whole, but which are influenced by architectural properties.

The quality attributes of a software architecture include:

- *Conceptual integrity*: an important property of the architecture of a system is its consistency. An architecture should not be crafted piece by piece, but should embody general organizing principles that are consistently applied throughout the system. Section 2.3 discusses *architectural styles*, which are embodiments of sorts of the conceptual integrity property,
- *Completeness and correctness*: an architecture is complete if it addresses all the non-functional requirements of the software[1] effectively and efficiently. For example, one such requirement could be platform independence. This can be attained using a layered architecture (thus isolating platform specificities) with a well-defined small API (isolation costs little),
- *Feasibility*: using the existing technology, the existing resources (human and technical), and within the required timeframe.

The three quality attributes are somewhat contradictory. Feasibility often requires shortcuts, which violate conceptual integrity. Further, completeness and conceptual integrity may clash. For example, a perfectly layered architecture where the different layers communicate with only the immediately adjacent layers, have performance penalties, and thus, may not address performance requirements (see e.g. [Van Der Linden & Muller, 1995]).

Qualities of the software system that are influenced by the architecture may be divided into observable properties of the system during run-time, and properties of the system as the product (deliverable) of development and continued maintenance. Run-time properties include performance (transactions per second), availability (mean-time to failure), security, usability, functionality (the extent to which the product addresses user needs), etc. Not all of these properties are architectural in nature, although a good architecture can only help make things easier.

Qualities of the software system as a product of development and maintenance include things such as *testability, integrability*, *modifiability*, *portability*, and *reusability*. Testability refers to the ease with which certain system properties may be verified. For example, an event-based architecture with a fixed pool of event handlers cannot guarantee that a given system will be able to handle some hard real-time constraints. In safety critical software, testability may become the overriding factor in choosing an architecture [Garlan & Perry, 1995],[Bass et al., 1998]. The

[1] Paul Clements disputes the distinction between functional requirements and non-functional requirements—which underlies the assumption that they are separable—especially when "non-functional" is construed to mean secondary.

remaining four properties are related to the architecture's accommodation of present-time differences and future evolution, and are tightly related to reuse.

2.3. Architectural styles and connectors

Given a set of quality attributes to seek (or optimise) in an architecture, there are two ways of architecting a system in a way that attains those attributes:

1. Devising an architecture development procedure that takes as input, a prioritised list of architectural quality attributes, and that produces an architecture that satisfies those properties by construction, or
2. Studying existing software system architectures with the hope of, a) identifying recurrent architectural styles, and b) finding a reasonably good correlation between these architectural styles and quality attributes.

The first alternative is fairly difficult, as software engineering in general remains an a-posteriori quantitative science, and thus, architectural design has gone the way detailed design has: *codifying best practices into recognizable abstractions*, i.e. *architectural styles*. An architectural style may be defined *a class of architectures* characterized by:

- *Component types*: these are component classes characterized by either software packaging properties (e.g. "COM component") or by functional (e.g. "transaction monitor") or computational ("persistence manager") roles within an application,
- *Communication patterns between the components*: indicating the kinds of communications between the component types,
- *Semantic constraints*, indicating the behavioural properties of the components individually, and in the context of their interactions, and
- *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 and practitioners have catalogued a dozen or so styles grouped in five families, the *independent components* family, the *data flow* family, the *data-centered* family, the *virtual machines* family, and the *call and return* family [Shaw & Garlan, 1996], [Bass et al., 1999]. Describing those styles in detail is beyond the scope of this paper. However, for our purposes, it is worthwhile to think of why these styles emerged. The short answer is that those 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 attributes, which suit different *classes of applications*.

3 Architecture development lifecycle

3.1. The traditional lifecycle

Devising the architecture of a software system is a fairly complex task, and one that has the greatest impact on the development, deployment, and maintenance of the software. It may be likened to a multi-criteria optimisation problem, where the criteria include the quality attributes discussed above, and possibly others. Viewing the architecture of a software as a software artefact in its own right, we can break down its development into the typical stages of

requirements, specifications, design, and implementation. This top-down view of architectural development may be a necessary abstraction for the purpose of documenting and managing an architecture (see e.g. [Bass et al., 1998]), even if we believe or feel that the creative steps of architectural development were not top-down [Coplien, 1999], [Perrochon & Mann, 1999]. However, 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 would normally have to be intertwined with the lifecycles of the other artefacts. We start by discussing the various stages of the lifecycle, and then explore ways of separating them.

We distinguish between the *functional architecture* of a system, which is a description of the main functions of the domain and of their interrelationships, and the *computational architecture*, which is the application-independent software organization of these components in terms of inter-component interfaces and interaction mechanisms [Shlaer & Mellor, 1997]. The two are not completely independent. For example, a given functional architecture might require lots of inter-module exchange of complex data, precluding data poor styles like pipe-and-filter and event-based, whereas another functional architecture might involve simply transfer of control, and would favour something like an event-based style. In this paper, we are interested only in the *computational architecture* and for the time being, we will assume that the functional architecture is a given[2].

For the case of the computational architecture, the ***requirements*** stage consists of identifying the business and technical constraints of the target system, as well as the (prioritised list of) desired quality attributes [Bass et al., 1998]. The ***specifications*** stage consists of going from quality attributes to *specifying* the *computational properties* of the architecture. An example such property may be "module A needs to send data to module B, and B is independently (re)usable, module C needs to have module D running, but it should be able to run in a degraded mode in D wouldn't start", etc., where modules A, B, C, and D have been identified as part of the functional architecture (called *conceptual model* in [Bass et al., 1998]). An alternative to specifying the (computational) nature of the pair-wise relationships between functional modules, we could specify the *type* of association *intensionally* as in "for all modules X and Y such that X sends data to Y, then Y is independently (re)usable", and specify the interaction mechanisms and protocols (*connectors*) needed to enforce the constraint. In other words, the ***specifications*** phase consists of ***selecting an architectural style*** that will guarantee all or most of the desired architecture quality requirements.

With our definition of computational architecture, designing and implementing the architecture consists of designing and implementing:

- The packaging of the functional components of the system,
- The interaction mechanisms between the components, as prescribed by the architectural style.

Traditionally, these two aspects have been inseparable from the functional aspects of system components, and consequently, the design and implementation of the computational architecture was a by-product of detailed design and implementation of the functional components, and the

[2] In fact, depending on the development methodology, the development of the function al architecture is anywhere from the beginning of analysis (e.g. [Shlaer & Mellor, 1992], Fusion [Coleman et al., 1994]) to the beginning of design (e.g. OMT [Rumbaugh et al., 1991]).

software architecture, as a stand-alone software artefact, ceases to exist when we move from the "architectural design of a system" (i.e., the specification of its computational architecture). Figure 1 illustrates this development lifecycle.

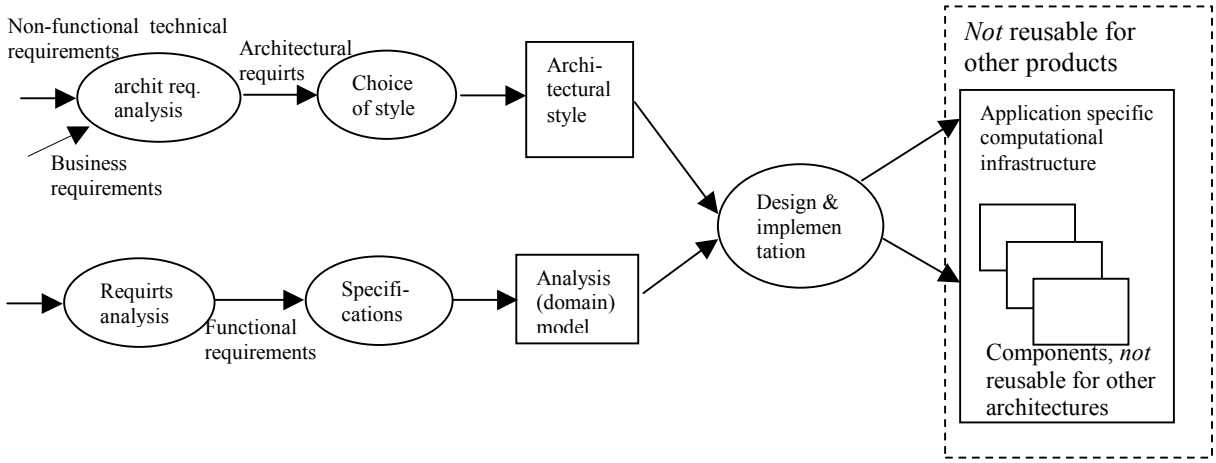

Figure 1. The design of the container (computational architecture) is intertwined with that of the contents (functional components).

Because of the lack of separation of domain aspects from architectural aspects (component interaction mechanism), the parts of the code that implement the computational architecture are embedded in domain components, and are thus not reusable for other domains. Similarly, because domain components carry with them architecture specific interaction mechanisms, *they* cannot be reused in different architectures (see e.g. [Garlan et al., 1995]). Domain components are *bound* to a particular computational architecture at *component design* time.

This problem is alleviated if we all choose the same computational architecture: all the components built into that architecture would rely on the same interaction mechanisms, and in principle, should inter-operate. In practice, however, inter-operation of the components will depend in part on the architectural style (the extent to which its connectors allow late bound compositions) and on the intrinsic quality of the components. For example, if we all use the *main program and subroutine* style, the various components will have the same invocation and control mechanism. However, to inter-operate, two components will usually need to know each other's interfaces, and while we may be able to inter-change implementations, the interaction style (procedure call) means that *functional dependencies* are hard-coded in the components themselves.

Generally speaking, depending on the architectural style, the component interaction mechanisms may be more or less complex, and more or less separable from the body of the components. If the interaction mechanism is complex, it is worth reusing. If it is separable, we get the ideal reuse situation, because of the reuse leverage. If it is not separable, we get the worst of situations: there is lots of complex interaction code that needs to be recoded for every component. If the interaction mechanism is simple, then the issue of separability becomes less important because there isn't much to reuse anyway. For example, if the connector used is procedure call, this is part

of high-level programming languages, and inter-procedure communication is implemented by writing a call to a procedure inside another. Thus, there isn't much to reuse, in terms of computational infrastructure, and as far as the computational infrastructure is concerned, it doesn't matter that the interconnection code is embedded in the components themselves—it does matter because the components are less reusable, but that is a different issue.

3.2. A separable architecture development lifecycle

By separating the development lifecycle of the computational architecture of a system from that of its components, we are able to:

- Reuse the computational architecture at all stages of development (style, design, and actual code), and
- Reuse some late incarnation (design or code level) of the components of the system across architectural styles and computational infrastructures.

In order to do this, we need two things, 1) a view of software components that separates the functional aspects from the operational aspects, and 2) an architectural style that supports flexible (late bound) compositions between software components. We discuss the first issue below.

We consider software components as points in a multi-dimensional space, with three important dimensions [Mili & Pachet, 2000], [Mili et al., 2000]:

- *The functional dimension*: this is the expression of the functionality of the software component in domain terms, regardless of any packaging or computational model,
- *The packaging dimension*: this refers the to embodiment of the functional dimension in a specific software artefact. For example, the same functional module, described by an IDL interface, may be represented as a Java class or an Ada package,
- *The computational dimension*: this is the dimension that considers the software component as a computation agent that is manipulated by a *virtual machine*.

The packaging dimension is more paradigmatic in nature, and the computational dimension is architectural. Figure 2 illustrates this view of software components.

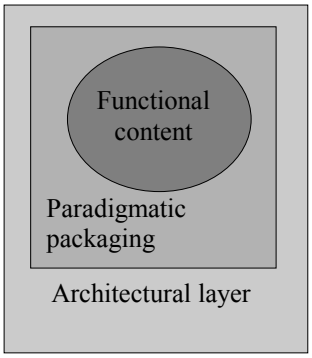

Figure 2. Dimensions of a software component as concentric layers.

Ideally, we should be able to develop the three dimensions *fully* and *separately* and bind them when there are no more manual and/or creative steps left to be performed to obtain a fully concrete component. The Shlaer & Mellor method is based on this premise [Shlaer & Mellor, 1997]. The analysis phase of their methodology develops a *complete executable specification* of the system to be developed[3]. The early phases of the design develop an *application-independent* architecture (i.e., a computational architecture) using the so-called *recursive design*, which is then mapped to the analysis model using code generators [Shlaer & Mellor, 1997] (see Figure 3).

[3] They argue, convincingly, that design is not "adding detail to analysis", but an analysis model must be *executable*, albeit on an inefficient, non-distributable, brittle (no fault-tolerance) virtual machine.

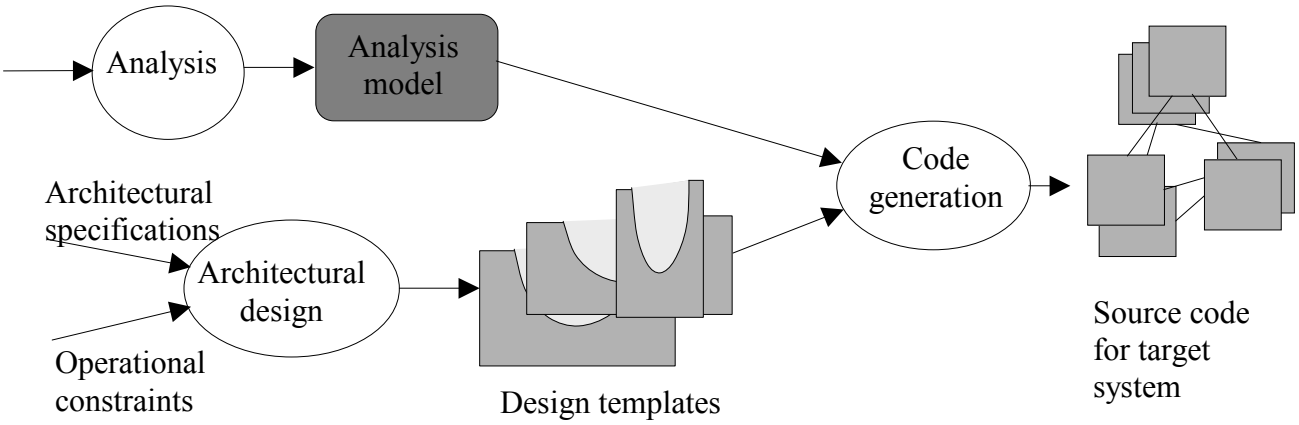

Figure 3. A Simplified version of the Shlaer and Mellor methodology

In the Shlaer & Mellor method, the output of the design stage is a set of *concrete* software artifacts consisting of design templates, which are macro-like structures that embody both the packaging of the components (as classes or modules or packages), and their architectural properties (their way of interacting with other components and with the environment). In particular, Shlaer & Mellor argue that, while the analysis model is object-oriented, the design model need not be. The last step is to generate the code of the target system by mapping the various analysis entities to the handful of design templates [Shlaer & Mellor, 1997].

The Shlaer & Mellor methodology and the accompanying toolset proved successful in real-time and reactive applications. Such applications are typically control intensive and a good portion of their processing logic can be captured in finite state machines, rendering code generation worthwhile; one is typically left with a only a few functional stubs to fill in manually [Shlaer & Mellor, 1994]. For our purposes, the importance of this method is that it stands as a real-life proof that this separation between the domain aspects of software components, and their architectural aspects is possible and practical[4].

Note that in this particular case, software components are still bound to their architectural style at coding (pre-compilation) time. However, the use of templates and code generation ensures that:

- The computational architecture (embodied in the templates) is reusable *without (re)coding*, and
- The domain components, which are fully specified at the analysis level, may be adapted to different architectural styles *without (re) coding*.

Note, however, that depending on the architectural style implemented by the design templates, it may or may not be possible to incorporate new components into a running system without having to regenerate the other components. Nowadays, the ability of an architecture to evolve during run-time is an increasingly important requirement of mission critical systems. Further, the advantages of the Shlaer & Mellor approach will only materialize in control-intensive applications, and reuse of components across architectures and of architectures across

[4] Practical enough that somebody (Project Technology Inc.) is making money with it.

applications will only occur if all are specified and built using the Shlaer & Mellor method (and toolset).

3.3. Towards a separable and late bound architecture development cycle

In order to attain the desired level of reuse, for both architectures and components, we need to further delay the binding the components to their host architecture. Figure 4 illustrates this schema.

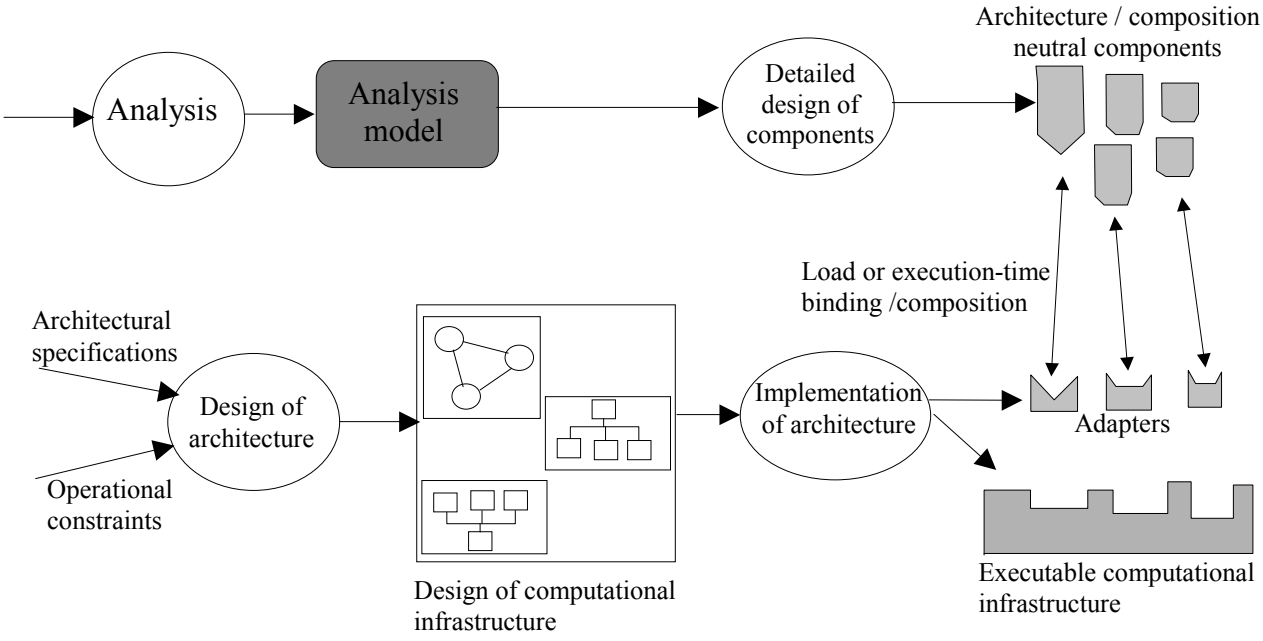

Figure 4. A separable development lifecycle for late-bound architectures

In this idealized lifecycle, the computational infrastructure is developed independently of the domain components. The architecture analysis (which includes the choice of the style) is not shown in the figure; we just show its output, the architecture specifications, which feed into the design of the computational architecture stage. If we think of the computational architecture as some sort of a *virtual machine* on which will run the domain components, then the design of the computational architecture consists of designing that machine. The output, labelled "design of computational infrastructure", is shown as a set of complementary diagrams representing different facets of that virtual machine (i.e., the architecture of the computational architecture). The implementation of that virtual machine is shown as a common layer of services with plug-in points for domain components.

The design of the domain components themselves is shown here to encompass two fairly distinct activities, *detailed design*, which deals with the internals of the component, and *architectural adaptation*, which prepares the components to interact through the computational infrastructure.

In this lifecycle model, detailed design and architectural adaptation are so distinct and independent that they can be executed in the reverse of the conventional order. The architectural adaptation is shown here using some sort of a wrapping technique whereby domain components are wrapped into "pluggable adaptors" that bridge the component interfaces with those expected by the computational architecture. Going back to Figure 2, the conceptual separation of a software component into dimensions translates into actual separate software artefacts, a *architecture-neutral component*, and an *architectural adapter*. How realistic is this lifecycle? We illustrate it using an example.

Assume that our architecture analysis stage led us to choose the *publish-and-subscribe* style. To corresponding computational architecture, we have to:

- Design the system's event manager and dispatcher that manages the "subscriptions" of the components, and handles (dispatches) the events they generate.
- Design the API of the components so that they can plug into the publish-and-subscribe infrastructure. 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,

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.

The following shows one potential implementation. We define **Component** as an abstract class that implements the publish-and-subscribe functionality, but that has an abstract definition of the event handling function. The **EventHandler** class is concrete and fully implemented, and so is the **Event** class.

```
abstract class Component {
    public void postEvent(Event e) {
          EventHandler.handleEventFrom(e,this);}

    public void publishEventClass(Class eventClass) {
          EventHandler.addPublisher(this,eventClass);}

    public void subscribe(Class eventClass) {
          EventHandler.addSubscription(this,eventClass);}

    abstract public void handleEvent(Event e);
}

public class EventHandler {
    static Hashtable publishers;
    static Hashtable subscriptions;
    …
    public static void addSubscription(Component comp,Class evtCls){
          Vector subs = (Vector)subscriptions.get(evtCls);
          if (subs == null) {subs = new Vector();
                subscriptions.add(evtClas,subs);};
```

```
            subs.addElement(comp);
        }

        public static void handleEventFrom(Event e, Component from) {
            Vector v = (Vector)subscriptions.get(e.getClass());
            if (v == null) return;
            Enumeration subscribers = v.elements();
            while (subscribers.hasMoreElements())
                ((Component)subscribers.nextElement()).handleEvent(e);
            return;
        }
        …
    }

    public class Event {
        private String type;
        private Object source;
        private Object data;
        …
    }
```

In this case, a single method (`Component.handleEvent(Event e)`) will take care of dispatching messages to component specific methods, depending on event type and data.

The reader will note that this design/implementation does not yield as clean a separation as that illustrated in Figure 4. For instance, while we are able to provide a domain- (application-) independent implementation of the computational infrastructure (**EventHandler** and **Event**), we have created a dependency between the components and the host computational architecture: the components have to inherit from the architecture-specific abstract class **Component** to be able to plug into this architecture. Section 4.2 will illustrate a set of techniques that can be used to ensure this separation.

For the purposes of this section, we have just illustrated the concept of an *architectural framework*. The classes **EventHandler**, **Event**, and **Component** constitute a framework whose purpose is to implement an application-independent *computational architecture* as an instantiation of the *publish-and-subscribe* style. Frameworks are well suited for implementing application-independent architectures for several reasons:

- Frameworks generally mediate interactions between components,
- Frameworks embody good design principles for separating the variable parts of a family of applications from the fixed part in such a way that the fixed parts can be fully implemented. In our case, the fixed part is the interaction mechanism between the components, while the variable part is the domain semantics of the components,
- Frameworks embody the principle of inversion of control, which enables us to instantiate a framework with components that were not developed with the framework in mind,
- Frameworks embody good design principles for supporting late-bound compositions between software components.

The next section discussed issues related to the construction of architectural frameworks.

4 Architectural frameworks

4.1. Definition

An application framework may be described by the combination of a *blueprint* for an application, combined with a set of *realizations* of components that play a particular role in that blueprint. Most authors will use the term ***design*** instead of ***blueprint***, which is perhaps a reflection of the fact that historically, the first application frameworks were *computing domain* (or *infrastructural*) *frameworks*, and more specifically, GUI frameworks. However, when we talk about *business frameworks*, the framework identifies, first and foremost, *domain classes*, their *interrelationships*, and their *interactions* (analysis level description); they may also include the design and partial realization of such classes, interrelationships, and interactions. In this case, the ***blueprint***, can describe either the analysis or the design, or both.

Architectural frameworks are application frameworks that implement architectural styles in an application (or domain) -independent way. In section 2.2, we talked about quality attributes of architectures, and talked about how architectural styles, viewed as classes of architectures, are geared towards optimising some of these attributes. Because architectural frameworks implement architectural styles, we can characterize the services offered by these frameworks in terms of the properties they help optimise. Bass et al. have broken down those properties into run-time observable properties, and static properties of the software product as an artefact. The run-time properties include *performance*, *scalability*, and *security*, and the static properties include things such as *modifiability*, *reusability*, *portability*, *composability*, and *integrability with legacy systems*. *Performance* and *scalability* usually imply both *persistence* and *distribution*. *Security* means supporting things such as *security policies*, *users*, *groups*, *encryption*, *authentication*, and the like. To some extent, thanks to the separation between the functional and the computational aspects of architectures, the things that are considered by Bass et al. as *properties of architectures* become actually *functionalities of architectural frameworks*. Among the static properties, *modifiability*, *reusability*, *composability*, and *integrability with legacy systems* imply a flexible component composition mechanism. There are two interesting variants on the component composition problem. We mentioned earlier the issue of *binding* regarding the binding of domain components to the architecture of the target system, and the issue of *component composition*. With the Shlaer & Mellor methodology, both are generated automatically from an architecture neutral (and composition-neutral) description of the components, which is valuable. However, because the binding takes place pre-compilation time, it precludes the possibility of adding third-party components in general, and more so at run-time.

Architectural frameworks are particularly challenging to build. First, their application domain is fairly wide: any business domain that needs the architectural properties “implemented” by the framework. As such, architectural frameworks have stringent requirements in terms of abstraction and openness. First, most of the computational infrastructure should be independent of the target application domain. Second, we should strive for the ability to connect components to the computational infrastructure as late in the process as possible. Finally, we should hide the complexity of the underlying infrastructure away, so that developers only deal with the underlying conceptual model, and not with the implementation details.

4.2. Technical issues

The building of application frameworks raises a number of technical challenges, which are discussed below. The challenges may be summarized into three design problems:

- *Late bound composition*: in order for the components that plug into the architecture to cooperate (via the architecture) and to be reused in *other* contexts, component composition should be late-bound, i.e. either at load time (we compose pre-compiled components) or at run-time,
- *Dynamic addition and removal*: architectures for mission critical systems require the possibility of dynamically adding or removing components while the system is running. This requires two things, a) the availability of meta-level information about the components, and b) dynamic linking,
- *Separation of concerns*: there are two issues here. First, we need to find a way of removing architectural or infrastructural code from the domain components, and leave it (or most of it) in the framework's code. This means that domain components should only have to embody domain logic. Second, we should be able to separate the various architectural services within the framework so that different combinations of services (or policies, for the same service) may be configured at will.

We discuss a set of common techniques for addressing these problems.

4.2.1. Composition techniques

Two components are composable if they are able to communicate (i.e. talk the same language) and inter-operate (i.e. have a way of invoking each other's services). Early approaches to composition focussed on using the same programming language, the same behaviour invocation protocol (e.g. procedure call), and building the composition logic in the components themselves (e.g. a procedure calls another procedure within its body). Composition through explicit invocation creates *lexical dependencies* between components. Because the dependency is at coding time, it precludes their composability with other components. Researchers and practitioners have been struggling with this issue and have striven to find ways of composing independently developed components, as late in the lifecycle of the system as possible. Notwithstanding composition techniques that are built into specific programming languages, there are a number of techniques for supporting late-bound (post-coding) compositions between independently developed components. Most of these techniques are design-based, and some will be discussed here. Others propose novel packaging techniques for composable program fragments, and will be discussed in section 4.2.3.

A number of the design patterns in the GOF book deal with composability issues [Gamma et al., 1994]. One such pattern, the *observer/observable* design pattern is actually a simplified version of the *publish-and-subscribe* framework described in section 3.3 where the objects that publish events are *observable*'s, and the ones that subscribe to events are *observers*. This patterns enables us to compose the behaviour of an object *A* with that of an object *B* without *A* having to know about, or communicate with *B*. This enables us to implement *A* in a context-independent fashion. One of the advantages of this pattern is that new *observers* can be added to an existing *observable* during *run-time*, enabling us to add services to a running application. Another pattern that is

useful for architectural frameworks is the *adapter pattern* [Gamma et al., 1994]. The adapter pattern enables us to adapt the interface of an object *A* (call it *server*) to the expectations of an object *B* (call it *client*) in those case where *A* does implement the required services, but does so under a different nomenclature. This problem is a common one in architectural frameworks, which are application-independent by *construction* but where the framework participants are necessarily application-specific components with their own way of performing some general computations. Figure 5 shows the aggregation-based version of this pattern.

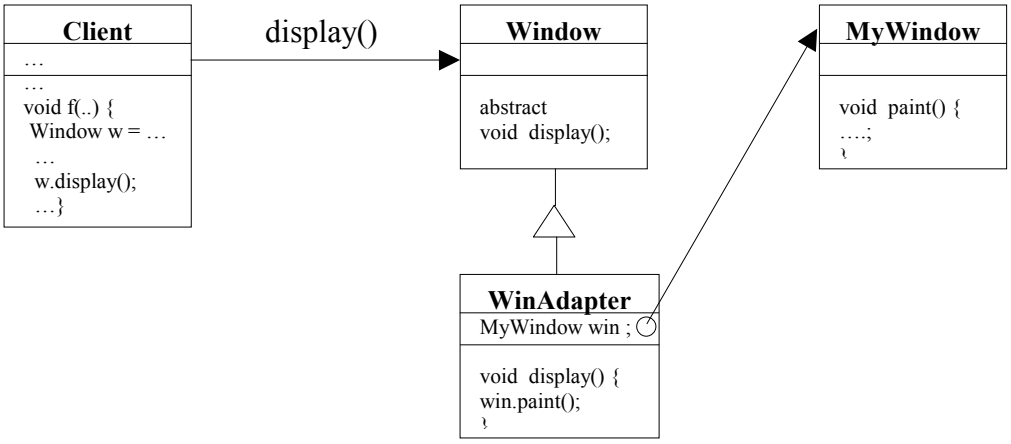

Figure 5. The adapter pattern. An aggregation-based implementation

Going back to our publish-and-subscribe framework of section 3.3, we could use a variation on this pattern to further separate the implementation of domain components from the requirements of the publish-and-subscribe framework. We reproduce the earlier version of component for convenience:

```
abstract class Component {
    public void postEvent(Event e) {
          EventHandler.handleEventFrom(e,this);}

    public void publishEventClass(Class eventClass) {
          EventHandler.addPublisher(this,eventClass);}

    public void subscribe(Class eventClass) {
          EventHandler.addSubscription(this,eventClass);}

    abstract public void handleEvent(Event e);
}
```

The idea was that domain components would have to inherit from **Component** and implement the function 'handleEvent(Event e)', i.e. define their way of responding to specific events. The problem with this implementation is that it is specific to this architecture and this architectural framework, and thus, our goal of developing the domain components and the architecture independently was not satisfied. A variation of the architectural framework would have the class **Component** be a superclass of a *delegator to the domain component*, rather than a *superclass of the domain component*. For example, let **Account** be a business component with the following signature:

```
public class Account extends WhateverIwant {
```

```
    …
    public float getBalance() {…}
    public void makeDeposit(float deposit) {…}
    …
    public void chargeTransactionFee(String transType) {…}
    …
    public void computeAccruedInterestSinceLastTransaction() {…}
    …
}
```

We may want to charge a transaction fee to the account each time there is a transaction, and the fee depends on the transaction. We may also want to compute the accrued interest since the last transaction, whenever there is a transaction that modifies the balance. In order to plug this component into the publish-and-subscribe architecture, we need to implement a class called **AccountComponent** defined as follows:

```
class AccountComponent extends Component {
    private Account account;

    public AccountComponent(Account acct) {
        account = acct;
    }

    public void handleEvent(Event e) {
        account.chargeTransactionFee(e.type);
        if (e.type.equals("Deposit")) {
            account.makeDeposit(e.data)[5];
            account.computeAccruedInterestSinceLastTransaction();
        }
        …
    }
}
```

In this case, the class **Account** embodies the business functions, regardless of the way these functions are invoked. The class **AccountComponent** is architectural in nature, and adapts the functional interface of the business component **Account** to the expectations of the publish-and-subscribe framework. Compared to the previous implementation, we have now decoupled the implementation of business components from the architectural. However, with this implementation, we can't add a new business component to a running application at will because to do so requires a new class whose definition depends on that of **Account**. We will see in the next section how to handle that problem as well.

4.2.2. Computational reflection

By definition, domain-independent architectural frameworks are supposed to work with components from different domains, and thus, cannot possibly anticipate the types (interfaces) of the components that will plug into them. So how do they contend with different domains? The previous section showed one way of doing just that: assuming that domain components exist

[5] The attentive Java programmer may notice an incompatibility between the type of e.data (**Object**) and the type expected by makeDeposit(…), which is a **float**. The call should be makeDeposit(((**Float**)e.data).floatValue()).

independently of the architecture, we have to create adapter code that bridges the expectations of the architectural framework with the semantics of the domain component. Good frameworks generally come with tools that generate the glue code automatically. For example, in the CORBA standard, *IDL compilers* usually generate the (server-side) code that the object request broker (ORB) needs (skeleton) in order to 'interface' with the actual object implementation. However, as mentioned earlier, this does not make it possible to add new components at run-time because the only way we know how to invoke their services is by calling them (i.e. methods) by name.

The solution to this problem depends on us being able to do three things:

1. Being able to introduce (link and load) new components into running applications,
2. Being able to ask those components (or some shared repository) about the functionalities they offer, and
3. Being able to invoke those services using a generic protocol.

Generally speaking, the solution rests on a combination of run-time environment functionalities (e.g. dynamic linking) and programming language features. The distribution frameworks (Java RMI, CORBA, EJB) offer these three features. We illustrate them for the case of the Java language. The idea here is to replace the class **AccountComponent** by a class that performs the same functionality, that does not mention the class **Account** by name, but that nonetheless, invokes the right methods for the right events. We start with a special version that assumes that each event requires the invocation of *one* method *with no arguments* on the domain object.

```
class DomainComponent extends Component {
(1)private Hashtable dispatchTable;
(2)private Object domainObject;

   public AccountComponent(Object obj) {
        domainObject = obj;
   }

(3)public void addDispatchEntry(String eventType, String methName){
        dispatchTable.add(eventType,methName);
   }

   public void handleEvent(Event event) {
(4)     String name = (String)dispatchTable.get(event.type);
(5)     Class[] types = new Class[0];
(6)     Method meth = domainObject.getClass().getMethod(name,types);
(7)     Object[] parms = new Object[0];
(8)     meth.invoke(domainObject,parms);
   }
}
```

Line (1) shows a variable that contains a dispatch table that associates event types (a string) to method names. This table can be constructed at run-time (when we initialise the domain component) by calling the method 'addDispatchEntry(…)' (line (3)) repeatedly. The method 'handleEvent(Event event)' first gets the method name associated with the event type (line (4)), then builds an array of types (classes) that correspond to the input parameters of the method we are looking for (line (5)); in this case, we assumed that all the methods have no arguments, hence the empty array. We retrieve a **Method** object in line (6) by 'asking' the domain object for its

class (an instance of the Java class **Class**), then asking that class to return the method with name 'name', and with signature 'types' (i.e. no arguments). Line (7) constructs an array of parameters for this method (again, an empty array of **Object**), and line (8) invokes the method.

The following shows an example Java code for loading an account object, and creating the corresponding domain component object, for illustration purposes:

```
…
(1) String newClassName = ….;
(2) Class domainObjectClass = Class.forName(newClassName);
(3) Object domainObject = domainObjectClass.newInstance();
    // initialisation code
    DomainComponent domComp = new DomainComponent(domainObject) ;
    // initialize dispatch table
    while (….) {
          String eventName = someFunction(…);
          String methodName = anotherFunction(…);
          domComp.addDispatchEntry(eventName,methodName);
    }
```

Line (1) gets the name of the new class somehow. In line (2), the method 'forName(…)' returns the **Class** object that represents the class named 'newClassName'. If the class is not yet loaded into the virtual machine, it will be. Line (3) creates an instance of this class (assuming that it has a public no-argument constructor), and the rest of the code is self-explanatory.

This definition of **DomainComponent** is independent of the application domain. It works under the assumptions that we specified, i.e. each event requires one method call, and all the methods have no arguments. The reader can imagine how we can extend this example to make it more general. For example, we may want to invoke a sequence of methods, for each event type, in which case the dispatch table will contain entries such as <event type, (method name 1, method name 2,…, method name j)>. If we want to have methods with arguments, then things get more complicated because we have to figure out which objects fill which arguments. This is not a syntactic issue; it is a semantic one. Note however that in the publish-and-subscribe paradigm, we don't expect the components to exchange complex data, and hence, the event handling functions will most likely have a few parameters (e.g., the data field of the event).

The example does raise, though, the issue of the limitations of reflection, be it language supported (as in this case) or simulated, such as the dynamic invocation interface (DII) used in CORBA: how to identify the objects that play particular roles in a particular computation. This will limit the 'semantic bandwidth' of architectural frameworks.

4.2.3. Separation of concerns

As applications grow in complexity and size, so does the underlying computational infrastructure. Within a distributed application, for example, we need to worry about issues of location transparency, persistence, component lifecycle management, security, error-recovery, distributed transactions, load balancing, and the like. A (very) naïve implementation of these functionalities would implement these functionalities on an object/component basis. For example, persistence,

which consists of saving objects' states on persistent storage, could be programmed on a per class basis, where each class would have a 'save(…)' method that establishes a connection to a relational database, constructs a query for saving the object, and executes it. The code for opening a connection to a database will be the same for all objects, modulo differences between database vendors and differences in table names. The code for constructing a query will be class specific, since different classes will have different attributes to store. The code for executing the query will be the same, again modulo differences between database vendors and differences in table names. This seems to be wasteful. The industry has solved the problem of differences between vendors a long time ago by creating an additional vendor independent layer (ODBC or JDBC), but this does not address the fact that each class needs to handcraft its own version of the persistence functionality.

A better solution would consider persistence (and other functionalities) as a *service* that application components may invoke with a simple protocol, and the 'service' will know what to do for each object. For example, with persistence, an object asks the persistence service to "save it", and the persistence service will (know how to) construct the query and execute it. This is made possible in part thanks computational reflection and to the availability of the meta-data during run-time. Alternatively, a persistence framework would include tools that parse class definitions and that generate default persistence procedures that developers can override if they wish[6]. What happens if we have several services to invoke? The following shows a simplistic implementation of the method 'makeDeposit(float deposit)' to illustrate the problem:

```
public class Account extends … {
   …
(1)synchronized public void makeDeposit(float deposit) {
(2)      balance = balance + deposit;
(3)      Log.log(this,"DEPOSIT",deposit);
(4)      PersistenceManager.synchronizeDB(this);
   }
   …
}
```

The signature of the method shows the qualifier 'synchronized' which means that the invocation of this method on an account will lock the object and prevent other threads from entering this function (or any other function that is also synchronized). Line (3) shows an example invocation of a logging service, and line (4) an example invocation to the persistence service, which synchronizes the contents of the database to make sure that the balance that is available in the database is the same one stored in the object. In the end, the function 'makeDeposit(**float**)' has one line that corresponds to the actual business functionality (line (2)), and a qualifier (line (1)) and two more lines to invoke architectural services. The end result is:

- Business functionalities are intertwined with architectural/computational functionalities,
- The code to implement a given service may cross cut several components (e.g. a multi-object transaction),
- The code is complex to build and maintain,

[6] This is the approach taken by a number of commercial object-to-relational persistence frameworks, including those that are part of implementations of the Enterprise Java Beans architecture.

- We can't activate or deactivate services on demand.

Interestingly, if the architectural functionalities are implemented as independent services, and if we use the publish-and-subscribe style, we can activate and deactivate these services at will. In this case, architectural services will simply subscribe to selected business events. This solution will work *if* we are already using a subscribe-and-publish style, *and* it is possible to invoke these services *at the end* of business method calls (a question of granularity of business methods) *and* will only work for those services that won't require coordination between several objects/components. In all other cases, we need another solution.

A number of researchers have been working on software packaging techniques that help the separation of concerns. The major challenge for these approaches is the fact that the code that addresses one particular concern does not follow traditional encapsulation boundaries, and may, in some cases (often) be smaller than a class or even a method. Aspect-oriented programming deals with "concerns" that crosscut several classes, which are packaged as *aspects* [Kiczales et al., 1997]. Aspects are "weaved" into traditional OO programs to inject a concern across a set of classes. Subject-oriented programming considers object-oriented applications as "fusions" of compositions of class hierarchies, each embodying a point of view on the same domain entities/objects. View programming considers classes as time-varying aggregates of *views*, where each view is an instantiation of a generic functional template for the domain object at hand [Mili et al., 1999]. Subject-oriented programming, the oldest of the three approaches [Ossher & Harrison, 1992], [Harrison & Ossher, 1993] has made it into a product, IBM's C++ toolset, and prototype support is available for Java (*HyperJ*). Aspect-oriented programming [Kiczales et al., 1997] has been implemented in Java (*AspectJ*) and is available free of charge. View programming is still a laboratory prototype. We discuss aspect-oriented programming briefly, in part because we feel that it best addresses the kind of problems we face with architectural frameworks, but also because it has a wider following than SOP, and because it is easier to explain.

Aspect-oriented programming recognizes that the programming languages that we use do not support all of the abstraction boundaries in our domain models and design processes. Underlying AOP is the observation that what starts out as fairly distinct concerns at the requirements level, or at the design requirements level (non-functional requirements) end up tangled in the final program code because of the lack of support, both at the design process level, and at the programming language level, for keeping these concerns separate. With aspect-oriented programming, these concerns can be packaged as *aspects*, which may be woven into "any" application that has those concerns. Kiczales et al. have proposed different forms of aspects. The simplest form of aspects, *advisories*, add some piece of code to specific methods identified by more or less complex <method,class> expressions[7], and may be used to instrument code or to handle some fairly generic functionality (logging, error handling, etc.). The following example shows an aspect used for tracing. In this case, whenever a **Task** is started, suspended, stopped, or resumed, we would like to log a message to that effect. An aspect called *Logger* is written to that

[7]. For example, the class and the method may be specified nominally, or as by properties that they satisfy (e.g., method "f(...)" of all the classes that implement some interface *I*).

effect. The aspect states what needs to be added to the core application classes, and where. An *aspect weaver* will "inject" the corresponding pieces of code appropriately.

```
class Task {
   public static int Priority = 0;
   ...
   public void run(int priority) {
      ...
   }
   public void suspend() {
      ...
   }
   public void resume(int priorty) {
      ...
   }
   public void stop() {
      ...
   }

...
}
aspect Logger {
   advise
      Task.run(int),
      Task.resume(int),
      Task.suspend(),Task.stop() {
   static before {
      Log.println(
         "Task number "+ taskId+
         " running at "+
               Calendar.time());
         }
      }
}
```

```
class Task {
   public static int Priority = 0;
   ...
   public void run(int priority) {
      Log.println(
          "Task number "+ taskId +
         " running at "+
         Calendar.time());
      ...
   }
   public void suspend() {
      Log.println(
         "Task number "+ taskId +
         " running at "+
         Calendar.time());
      ...
   }
   public void resume(int priorty){
      Log.println(
         "Task number "+ taskId +
         " running at "+
         Calendar.time());
      ...
   }
   public void stop() {
      Log.println(
         "Task number "+ taskId +
         " running at "+
         Calendar.time());
      ...
   }
...
}
```

In this example, the *Logger* aspect specifies explicitly the class whose methods we wish to instrument. More advanced forms enable us to use "wild cards" on class names and/or method names. The ability to define aspects *intensionally* is one of the major advantages of aspect-oriented programming, and make it more appropriate for architectural frameworks where the aspects are to be injected in domain entities that are not known before hand[8]. The same is true with view-oriented programming [Mili et al., 1999] where views are created by instantiating domain-independent templates (called *viewpoints*); with subject-oriented programming, the subjects (functional slices of class hierarchies) are supposed to be composed with other subjects or class hierarchies that mirror, more or less their structure and nomenclature. A second major advantage of aspect-oriented programming is the fact that the aspects are smaller-grain than methods, as the example above illustrates. Aspect code may be injected at different places in the

[8] See the www.aspectj.org web site

code, besides entering or exiting methods. However, both subject-oriented programming and aspect-oriented programming require compile-time "injection" (composition) of aspects (subjects), whereas view programming enables us to activate and deactivate views (services) on demand [Mili et al., 1999].

Along with the growth of the user communities of these technologies (AOP and SOP), researchers have started conducting empirical studies to assess the effectiveness of these techniques, and to characterize, if at all possible, the kinds of problems for which each one of these techniques is best suited (see e.g. […] and [Murphy et al.,1999]). Aspect-oriented programming may prove to be a key ingredient of architectural frameworks, because, not only does it offer novel composition technologies to build more flexible and configurable computational infrastructure, it may also be used to alleviate the need for removing the architecture specific code from domain components. Because that code can be "woven" into complete stand-alone Java (or others) applications, the development cycles for domain components, on one hand, and architectural frameworks, on the other, are kept separate, and unlike the Shlaer & Mellor approach, the aspect weaver takes standard Java as input, and not some proprietary format like Shlaer & Mellor's templates.

4.3. Building architectural frameworks

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. [Aksit et al., 1999]),

(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. [Shmid, 1999]).

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.

4.3.1. Frameworks as products of domain engineering

Domain engineering is a set of activities aiming at developing reusable artefacts within the real 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. [Kang, 1990], [Jacobson et al.,1998]). 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.

The literature abounds with recent experiences with framework development in both industry and R&D labs (see [Fayad, Schmidt & Johnson, 1999]), 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 [Aksit et al., 1999], [Jacobson et al., 1998], and several approaches described in [Fayad & Johnson, 1999].

4.3.2. 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. But instead of building it from the top down, we start building applications from the domain, and then start introducing variations in the those parts of the applications that concern the domain of interest.

A good number of frameworks originated from R&D labs, and fall into this category, for many reasons. First, the work in R&D labs 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 labs are interested in investigating the process of building frameworks *as a subject of study*[9], and thus, studies in 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.

[9] 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.

A representative example of these approaches is *hot-spot driven development* as advocated by Pree [Pree, 1999] and Schmid [Shmid, 1999]. A framework "hotspot" 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 hotspot. The idea here is that we can identify those hotspots early on in the process, but we don't necessarily develop them in the first iteration: the first iteration will solve a specific problem; later iterations will design variability into the identified hotspots. Figure 4 illustrates this idea.

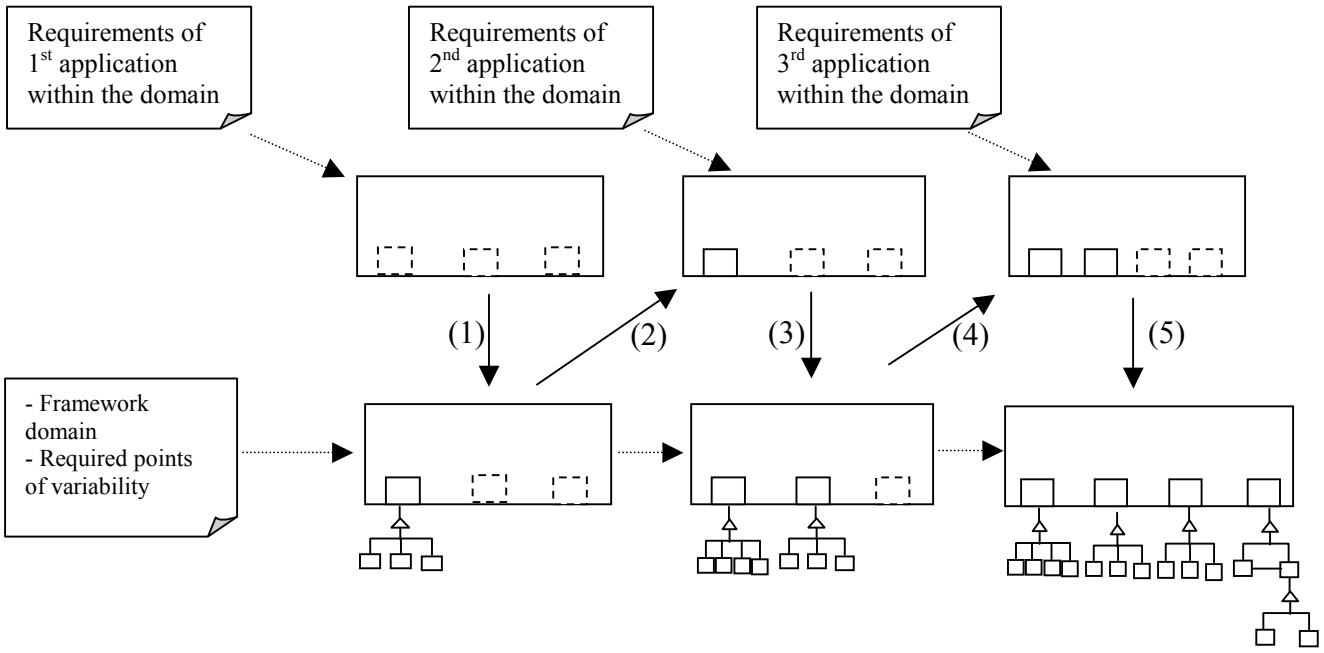

Figure 4. Incremental framework development by hotspot design.

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 (hotspot) is fully developed; probably that aspect that must often changes from one application to another (step 1). Designing a hotspot means changing a specific binding of a point of variation (e.g. a terminal logger) into [Schmid, 1999]:

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, 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 *hotspot*. 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.

4.4. 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 (*hotspot 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 lots of effort go into developing software artefacts whose usefulness and cost-effectiveness is 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 (hotspot driven), and the consistency of the resulting overall architecture may suffer, as different hotspots 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. [Mili & Pachet, 2000]). The functional aspects of the architecture will be developed in an incremental and iterative fashion, as shown in Figure 4 above. The idea here is that the points of variation will be functional in nature. Figure 5 illustrates this lifecycle.

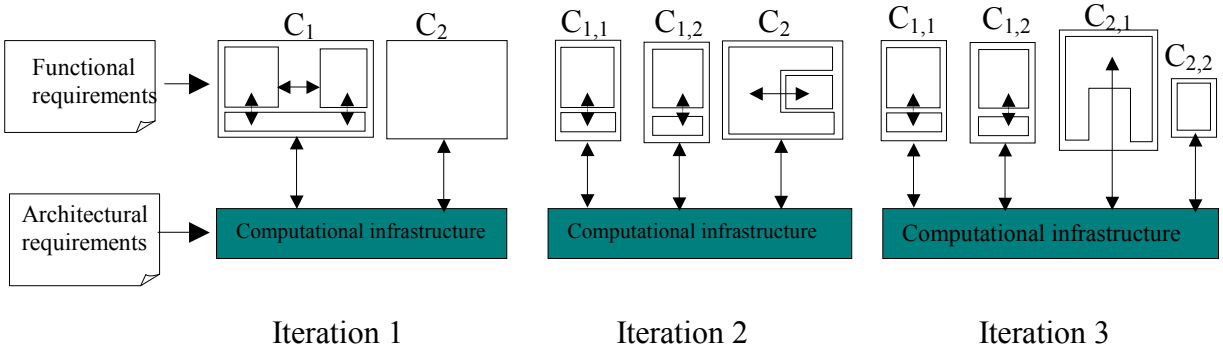

Figure 5. A lifecycle for enterprise frameworks

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 computational infrastructure, we also develop some application components which may coarse, and 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 "externalised", 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 externalise the subcomponents of that legacy system, providing greater degrees of freedom, and supporting the interchangeability of components.

5 Summary and discussion

The reusability of a software artefact may be seen as the combination of two properties, *usability* and *usefulness*. Usefulness is related to the "number of times" that a given artefact is needed within a (or its) particular application domain. Usability refers to the ease with which the artefact may be used in those cases where it is useful. Large-scale organized efforts at reuse have established that:

- Fine-grained software components may be more usefulness (higher degrees of freedom), but are not cost-effective, and
- The greatest technical obstacle to the reuse of large-grained software artefacts is lack of *usability* and not lack of *usefulness*.

In particular, there is a lot of useful functionality out there in legacy systems, but most of it does not use the right control paradigm, or doesn't abide by the right interaction protocols, or simply, isn't implemented in the right programming language (see e.g. [Garlan et al., 1995], [Bass et al., 1998]). These are all architectural issues. Enterprise frameworks are large-scale architecture-centric application frameworks that cover a significant portion of the functional requirements of *enterprise applications* within a particular business domain. Because of their size and complexity, care must be taken to develop them to achieve the desired goals, and to minimize the financial risks inherent in developing them. As such, they require carefully designed development lifecycles, ones that ensure continual improvement through conservative extensions. In this paper, we discussed the lifecycle issues involved, and proposed a lifecycle that combines the "directiveness" of architecture-centric development, with the flexibility and incrementality required of creative continual improvement. More work is needed to flesh out such a process in a way that corporations may use for planning purposes.

6 References

- [Aksit et al., 1999] Mehmet Aksit, Bedir Tekinerdogan, and Francesco Marcelloni, "Deriving Frameworks from Domain Knowledge," in *Building Application Frameworks – Object-Oriented Foundations of Framework Design*, eds Fayad, Schmidt & Johnson, John Wiley & Sons, pp. 169-198, 1999.
- [Allen & Garlan, 1994] Robert Allen and David Garlan, ''Formalizing architectural connection,'' Proceedings of ICSE'16, May 16-21, Sorrento - Italy, pp. 71-80.

- [Bass et al.,1998] Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice*, Addison-Wesley, ISBN 0-201-19930-0, 1998.
- [Fayad & Johnson,1999] Mohammed Fayad and Ralph Johnson, *Domain Specific Application Frameworks*, John Wiley & Sons, 1999, ISBN 0-471-33280-1.
- [Garlan et al., 1995] David Garlan, Robert Allen, and John Ockerbloom, ‘‘Architectural mismatch: Why reuse is so hard,’’ IEEE Software, vo. 12, no. 6, November 1995, pp. 17-26.
- [Kang *et.al.* 1990] Kang, K., S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study" CMU/SEI-90-TR-021. www.sei.cmu.edu/publications/documents/90.reports/ 90.tr.021.html
- [Schmid, 1999] Hans Albrecht Schmid, “Framework Design by Systematic Generalization,” in *Building Application Frameworks – Object-Oriented Foundations of Framework Design*, eds Fayad, Schmidt & Johnson, John Wiley & Sons, pp. 353-378, 1999.
- [Szyperski, 1999] Clemens Szyperski, *Component Software : Beyond Object-Oriented Programming*, Addison Wesley, 1999, ISBN 0-201-17888-5
- [Van Der Linden & Müller, 1995] Frank J. Van Der Linden and Jürgen K. Müller, “Creating architectures with Building Blocks,” *IEEE Software*, November 1995, pp. 51-60.