How can frameworks facilitate component reuse?

Rikard Land

Department of Computer Science
Mälardalen University
Västerås, Sweden

rikard.land@mdh.se

Abstract: Reuse of software has the potential to decrease costs and development time at the same time as software quality is increased – i.e. solve the infamous “software crisis”. In this paper, we investigate the notion of “frameworks” and discuss, in a general manner, the problems of reuse within frameworks. The term “framework” is in the paper used in a very broad sense: our argumentation is applicable on component frameworks (as in chapters 17 and 18 of the book), but also on object-oriented frameworks, software architectures; indeed, any software environment or abstraction can be regarded a framework (such as operating systems and programming languages).

Frameworks have proven to evolve, and we argue about the reasons for this and find that they will continue to do so. Based on a literature survey and our discussion on framework evolution, we outline answers to how a framework should be designed to facilitate reuse and how a component, developed within a framework, should be designed to become as reusable and long-lasting as possible.

We apply our discussion to chapters 17 and 18 of the book, evaluating their prospects of facilitating reuse.

1 Introduction

This extended report covers both chapters 17 “Architectural Support for Reuse: A Case Study in Industrial Automation” and 18 “A Framework for Integrating Business Applications”. The two chapters present similar frameworks for building applications and thus deal with similar problems. In chapter 17, the framework is called “Aspect Integrator Platform” and in the one in chapter 18 “Information Organizer”.

The frameworks presented in the two chapters are said to facilitate reuse. The idea of reusing software has been around in one form or another for a long time, but has largely failed to fulfill its expectations [11]. It has proved to be very difficult to reuse software in a systematic manner. “Although reuse is valuable, it is not free” [10].

This paper discusses reuse in the context of frameworks. We start in section 2 by examining the notion of “framework”, and continue in section 3 surveying the literature and discussing reuse in a more general manner. In section 4 we introduce a model to describe abstractions in general, apply it on frameworks in general and the frameworks in the chapters in particular. In section 5 we argue that there are problems with frameworks that will never be fully overcome: there are
properties inherent in every framework making them unstable. In section 6 we summarize the discussion.

2 What Is a Framework?

“Framework” is one of those very general words (as “system” and “object”) whose meaning depends heavily on the context. It is therefore important that we clarify what we mean with the term “framework” in this report. We can discern many other uses of the term:

- We note that in the context of component-based software engineering, “framework” is not a precisely defined term (see chapter 1, “Basic Concepts in Component-Based Software Engineering”; see also Bachman et al [1]). In the two chapters of the book, the term “framework” is never defined, but is used in the sense “environment for developing and executing software”\(^1\), and the components are binary executables.

- The term is used in the object-oriented community for customizable source code [5,6,9,10], and Johnson gives two definitions: “a framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact” and “a framework is the skeleton of an application that can be customized by an application developer” [9,10]. Codenie et al says that a “framework is usually defined as a skeleton program defining a reusable software architecture in terms of collaboration contracts between (abstract) classes and a set of variation points, or hot spots” [5].

- A system’s architecture [2,3,14] can be said to be its framework.

- In fact, every “abstraction” can be said to be a framework such as an operating system or a high-level language – such abstractions make it easier to build homogeneous applications by abstracting away certain elements from the level “below” and introduce new concepts.

For an object-oriented framework, the reusable entity is thus the framework itself. It is used as a skeleton to build upon; it is not an environment to build reusable entities in. For a component framework on the other hand, such as the ones in the two chapters, the reusable entities are components. In an application resulting from an object-oriented framework, it is not possible to find the clear line between the framework and the customized program code; in a component framework, there is a clear difference since the components reside in separate files. We therefore believe that a component framework has a larger chance to facilitate component reuse.

Using a broad definition of the term, it is implicit in subsequent discussions that all software is actually developed within some sort of framework (often several simultaneously, but this would be a separate discussion). The discussion in this paper is valid for any of these uses of the term, but we will return to the chapters’ use of the term every now and then.

\(^1\) Words like “platform”, “infrastructure”, and “environment” are also used almost interchangeably for “framework” in the two chapters. We will not elaborate on any differences between the terms, but will consequently stick to “framework” in this paper.
3 Software Reuse in the Past

The NATO Software Engineering conference in 1968 is considered the birthplace of the Software Engineering field [11]. The term “software crisis” meaning “the problem of building large, reliable software systems in a controll, cost-effective way” [11] was coined at this conference [11]. As a solution McIlroy suggested a future where software components are reused [12]. His notion of components is “routines”, while today’s notion of component-based development are about binary components [15]. Nonetheless, McIlroy anticipated several aspects of the development of the field of component-based software engineering – for example, he discusses how other features than mere functionality will affect components, such as performance and memory consumption, but also business issues such as time of deliverance, and how easy it is to assess that it will suffice for the needs at hand. These are “hot” research areas of practical interest – see e.g. chapters 2 and 6 of the book about semantic specification, or part 4 about component evaluation and system evaluation.

Krueger presented a thorough overview of software reuse in 1992 [11]. He states that “[f]or a software technique to be effective, it must reduce the cognitive distance between the initial concept of a system and its final executable implementation” (cognitive distance is the difficulty to develop a system one step). In other words, if it does not become easier to build a system with reuse than without, one can as well do without reuse; reusing software must not only be possible, but also “better” than rewriting it. Quite obviously, “better” is actually a sort of trade-off; it might e.g. be cheaper and faster to use an existing component, but some desired functionality will not be available.

Garlan et al discuss a case when a project estimated to cost six months and one person-year, turned out to cost two years and five person-years until the first prototype was running [8]. In one sense, the different components were designed for the same framework (operating system), but each component included parts that one would consider part of a framework. For example, each component implemented an event loop, based other assumptions on this, and required single-threading – it proved to be a major task making these components work together! The authors named this problem “architectural mismatch”. Johnson writes that if “components make different assumptions [about its environment] then it is hard to use them together” [10]. One reason for the difficulties to reuse software is therefore strongly connected to the view the developers of the components have about the environment – or framework – of the components. We will in section 5 discuss another aspect of how the developer’s view of the framework may make it difficult to reuse software. One of the ways forward is to “[m]ake architectural assumptions explicit” [8].

The reusable entities of a framework must be well defined to be meaningful. As we said, in an object-oriented framework, it is the framework that is the reusable asset; it does not support construction of reusable objects within the framework (at least not to the same extent as a component framework has the potential to do). In the two chapters the reusable entities are well defined. Aspect Integrator Platform has libraries of “Object Types” (with a versioning facility) and in Information Organizer it is possible to create and reuse applications and databases. In addition, to make the entities reusable in practice, the framework must be stable enough to be compatible with older components. This is because for a successful framework one expects new applications to be created, and this happens while time passes (and new versions of the framework are released).
4 Krueger’s Model

In Krueger’s article about software reuse, a generic model is used where every abstraction is divided into a variable part, a fixed part and a hidden part [11] (see Figure 1). The hidden part is invisible for the user, the fixed part is visible but not modifiable by the user, and the variable part makes it possible to create “realizations” of the abstraction. When the variable part has been instantiated, the resulting “abstraction realization” can be used as a component (in a wide sense). This model conforms very well to the description we gave of a framework above: it abstracts away certain elements (the hidden part) and provides support for new concepts (that is what makes it an abstraction).

![Krueger’s model for software abstractions](image)

Krueger uses the example of the “stack” abstract data type [11]. The fixed part of the stack abstraction contains the last-in-first-out semantics of a stack, and the “push” and “pop” operations. The variable part contains e.g. the type of elements stored in the stack and the maximum stack depth, and the hidden part contains implementation details.

4.1 Frameworks Revisited

We will now give a rough description of a “framework” in this model. It should hide implementation details of the underlying system, to make it abstract enough – only properties interesting in the application domain should be left visible in the fixed part and the variable part. The variable part must be variable enough to allow for useful instantiations, but the variability should be quite limited, to make sure that components built for the framework will work in all instantiations of the framework. The decisions whether a certain feature of a framework should be hidden, fixed, or variable is crucial for a framework’s success (or any type of abstraction, for that matter): if the hidden part is too small, the abstraction is not much of an abstraction; if the variable part is too small, it might not be very useful or reusable; if the fixed or hidden part contain elements that should be variable, it is not as usable or reusable as it could have been; etc.
Fayad et al emphasize “Enduring Business Themes” (EBTs) as the elements that should be built into a framework to make it stable [6] – this corresponds to the fixed part of Krueger’s model. With this they mean business specific invariants of the framework, properties that survive over time: “in the transportation industry, the EBT is concerned with moving material from one location to another”.

As a result of this discussion, it is clear that any framework must have a clearly defined application domain\(^2\). Otherwise, the variable part must be very large, causing the framework to be hard to create, understand, and test. It also becomes more likely that the framework will change over time, since not all future changes was anticipated in the initial design. An application programmer has some sort of notion of the application domain (business domain, technical domain, etc), and wants to choose a framework that will support him the most – i.e., which is specific for the domain.

4.2 The Chapters Revisited

Let us return to the frameworks presented in the two chapters, Aspect Integrator Platform and Information Organizer. How does these fit the general description of frameworks we have given here? Do they have the features we have outlined for a framework to be successful?

- Both have well defined application domains – the Aspect Integrator Platform is a “component-based, industrial automation” platform, and Information Organizer “is mainly intended to be used in office automation domain”. Another way to put it – both frameworks are built upon the “Enduring Business Themes” of each domain (industrial automation and information systems respectively). Therefore, both are likely to become stable.

- Both have a certain amount of variability, but not more than what can be made transparent to the users. For Aspect Integrator Platform, the number of servers and services is variable. Information Organizer uses Microsoft technologies such as security and Active Directory to achieve some variability; the types of user interfaces and data sources are other variables chosen at installation time (or later).

- It is not very clear which parts of the frameworks are hidden, but it is clear that parts are hidden to make up a useful abstraction. For example, it is possible to implement a certain COM interface in Information Organizer and rely on that the framework will invoke certain methods when appropriate (but how the framework achieves this is hidden).

5 The Inevitable Evolution of Frameworks

There are a lot of abstractions upon the logical foundation of software making software construction easier and faster by providing a framework easier to understand – there are high-

\(^2\) Although the domain should be clearly defined, the notion of domain should here be interpreted in a broad sense: a domain can be e.g. a technical domain (which is the case e.g. when designing a high-level programming language, intended for a certain type of programs) or a business domain (such as broadcast planning software for television stations [5]). Our point here is that the domain the framework is constructed for must be well defined.
level languages, operating systems, abstract data types, databases, component technologies, etc. In a sense, such a framework can also be said to make up the “universe” in which software can be written. This view is not uncommon among developers, since it is appealing: one can work in an environment consisting of understandable concepts. However, it is a dangerous point of view, since the environment in which software is built is in fact not stable. Is this an evil we might eventually get rid of, or is it an inherent property of frameworks? We will argue that it is indeed an inherent property of the nature of frameworks, and that therefore the view of a framework being an unchangeable universe is misleading for developers and will unmistakably lead to problems, since frameworks actually change. Similarly, with this view the software will probably need to be redesigned if it is to be integrated into a competing framework (such as porting software to other operating systems).

Developers are drowned in new tools, new middleware, new operating systems, released in new versions with new features and bug fixes every too often\(^3\). The question to answer is whether we can hope for a future where a framework has become totally stable? We do not think so: society evolves, technology evolves, businesses evolves; contrary to what some software developers might believe, software is only a tool for humans to achieve something in these (and other) contexts. “As the business evolves, so must the framework. It is simply not possible to conceive a framework that anticipates all future evolutions. A framework is never finished” [5] (italics added). There is no way to hinder this instability in the world of software – there is no such thing as a “best” framework we may eventually arrive at, since the notion of “best” depends on societies and businesses (which evolve and vary). We are thus aiming at a moving target, and believing it is immovable will lead to more problems than if we face the problem.

Relating this discussion to Krueger’s model, this corresponds to changes of the fixed or variable parts of the framework; changes to the hidden part should of course not affect the functionality of applications built on top of a framework (applications should only experience higher performance or security). Speaking with Fayad et al [6] this is the “Enduring Business Themes”. Our point in this section is that competition and evolution will make these parts/themes change as time passes.

To address this problem, the responsibility lies as we see it both on the (developers of a) framework and the (developers of) components. A component developer must understand that frameworks will change, and that there will probably be other, competing frameworks the component must support (especially if the component is successful in one framework). But “[w]ithout carefully architecting for reuse, reuse will not succeed” [4]. On the other hand, a framework designer should try to make the framework as stable as possible, by anticipating future requirements of the framework.

---

\(^3\) Imagine a physical universe released in new versions every now and then – “Cosmos release 4, build 1381: minor changes to laws of electromagnetism and gravity”. This is the environment software developers accept (although grumpingly) if they adopt the view that the framework is the universe in which they create software (observe the “if”).

---

**Rikard Land: How can frameworks facilitate component reuse?**

5.1 Slowing Down Framework Evolution

The problem of software reuse lies not only within any single framework, but the whole software development environment with many frameworks evolving rapidly. This must be addressed by any reuse attempt. To prolong the lifetime of components, can something be done to slow down framework changes? “Only where an overall architecture is defined and maintained do evolution and maintenance of components and systems find the firm foundation they require” [15]. Fayad et al emphasize “support for software stability” as being important for a framework [6]. What can framework designers do to decrease the speed with which frameworks evolve, thus extending the lifetime of components and applications?

- Of course, future changes should be anticipated to as large extent as possible. The ideal case, that all future requirements were known in advance, is as we have argued quite out of reach.
  - However, it is to our experience not uncommon that this fact is just ignored. Just taking the issue under serious consideration is one major leap forward.
  - After building several systems in the same domain, an organization gathers experience of the Enduring Business Themes [6] that may be used to build a successful framework. Codenie et al describes exactly this scenario [5].

- In practice, the mere existence of applications built upon a framework provides some motivation to keep the framework stable.

5.2 The Chapters Revisited

Chapter 17 reflects our discussion, since Aspect Integrator Platform is “ABB’s next generation of automation system architecture”: there were presumably other, earlier frameworks that evolved for a period of time but eventually were not sufficient. “Early in the life cycle, a framework is less likely to reflect the level of completeness required for the application domain” [6] (this goes for earlier generations of similar frameworks too). At some point of time, it was probably decided that the framework evolved too far from the original specifications. Most likely, Aspect Integrator Platform will last longer and facilitate a higher degree of reuse than its predecessors, since ABB presumably has gathered a lot of experience through previous products and have identified “Enduring Business Themes”, as Fayad et al puts it [6]. It should be noted that we have no detailed knowledge of the history of ABB’s automation products, so this discussion is somewhat speculative.

Information Organizer is intended for reuse of components created in frameworks that are becoming out of date. But instead of modifying components to fit in a new framework, the framework is designed to handle components “as is”. This gives an unexpected opportunity to reuse components not originally intended for reuse (mainly applications and databases, as we see it). There is also the possibility to modify components to be more tightly integrated into the framework, as we discussed above.
6 Summary

Thus, software components are hard to reuse because of several reasons, reasons inherent in the frameworks in which it is developed. How then can frameworks facilitate component reuse? The question of the title of this extended abstract is too complex to give a simple answer. We believe though that we have outlined parts of an answer in this paper, and summarize these below.

- Our discussion and the literature strongly support the conclusion that frameworks inevitably evolve: “A framework is never finished” [5].
  - There is no such thing as “the ultimate framework”. Society, technology, businesses etc. evolves, and the requirements on software (including frameworks) changes.
  - It is of course possible to freeze the framework forever, and never allow any updates other than simple bug fixes. This of course works, but at the prize of not having bugs fixed, and the software becoming out of date [13]. In our era of system integration, this is for many companies not an option.

- We still believe in software reuse as a solution to the “software crisis”. We have to find means to make it possible even though frameworks are not static.
  - For a component developer, having a realistic view of frameworks solves part of the problem. A framework should not be regarded as an absolute universe to create software in, but rather as an evolving environment. Szyperski talks about an inevitable “world of change” [15]. Garlan et al states that to allow for realistic reuse of components, these should “[m]ake architectural assumptions explicit” [8].
  - For a framework designer, it is important to try to make the framework stable by anticipating future changes. Fayad et al describes this as using Enduring Business Themes as the fundamentals of a framework [6]. Using Krueger’s model, the task is to decide whether a certain feature of a framework should be hidden, fixed, or variable [11].

6.1 The Chapters Revisited

What have we said about the particular frameworks of the two chapters? We said that both have well defined application domains, and seem to be built upon the “Enduring Business Themes” [6] of their respective domain, and have potential of proving stable. Relating them to Krueger’s model, both seem to have made good decisions about what to put in the variable, fixed, and hidden part.

The approach described in chapter 18 is the more advanced from a reuse perspective. With “Information Organizer”, it is possible to reuse data and applications not designed for the framework. While the framework in chapter 17, “Aspect Integrator Platform”, follows the pattern we have described and forms a new foundation for reuse but requires new reusable components and applications to be written, Information Organizer addresses the problem of integrating legacy systems; the framework thus turns the discussion above upside-down – it is
possible to create a new framework with the goal of reusing components designed for other frameworks.

On the other hand, Information Organizer is more dependent on other frameworks (Microsoft technologies), which might make it less stable.

6.2 Recent Trends and Future Work

In 1968, the only possible reusable entities for McIlroy were “routines” [12]; in 1992 Krueger discussed eight ways of reusing software, of which reusing source code components was one, but binary components was not discussed [11]. This reflects the development that has taken place during the years, from the more specific to the more all-embracing – from routines to classes, from language-specific to language-independent, from platform-specific to platform-independent. Technologies and frameworks have emerged that facilitate reuse (for example, language independence is achieved through binary standards like COM, CORBA and .NET). Software Architecture is a field on the highest abstraction level with a high reuse potential [2,3,14]. Reuse of expertise through the use of design patterns is also a rapidly emerging field [7].

An interesting issue, briefly touched upon in this paper, is the possibility of cross-framework reuse: to what extent is it possible to create components that will work in similar, competing frameworks, such as databases or graphical packages? Szyperski talks about virtual machines and Apple’s “fat binaries” (files containing binary code for several platforms) as ways to reuse components on several platforms [15]. The use of virtual machines has become more and more popular: Java’s virtual machine is well known, and the .NET platform specifies one to facilitate both cross-operating system and cross-language reuse.

7 References


Rikard Land: How can frameworks facilitate component reuse?

Chapter 17: Architectural Support for Reuse: A Case Study in Industrial Automation, O. Preiss, M. Naedele
Chapter 18: A Framework for Integrating Business Applications, E. Gyllenswärd, M. Kap