

A “Framework” for Object Oriented Frameworks Design

David Parsons¹, Awais Rashid², Andreas Speck³, Alexandru Telea⁴

¹Systems Engineering Faculty, Southampton Institute, UK
dave.parsons@solent.ac.uk

²Cooperative Systems Engineering Group, Computing Department, Lancaster University, UK
marash@comp.lancs.ac.uk

³Wilhelm-Schickard-Institute for Computer Science, University of Tuebingen, Germany
speck@informatik.uni-tuebingen.de

⁴Department of Mathematics and Computing Science, Eindhoven University of Technology, The Netherlands
alex@win.tue.nl

This paper developed from collaborative work undertaken as a result of the eighth workshop for PhD Students in Object Oriented Systems (PhDOOS) held at ECOOP 98

Abstract. Object-oriented frameworks are an established tool for domain-specific reuse. Many framework design patterns and development processes have been documented, typically reverse engineering the common framework architecture from a number of conventionally built applications within a given domain. Frameworks vary in their construction from white box inheritance based systems to black box composition based systems with a full range of hybrids in between. The development cycle of a framework is generally assumed to follow a path from open framework to closed application. We describe a more flexible component-based approach to framework design that stresses a common interface for ‘plugging-in’ new components at different stages of the lifecycle. An analysis of roles in framework development indicates that the traditional boundaries between developers and end users is unnecessarily rigid. We can more helpfully regard the various stages of a framework’s development as an open continuum within which its various ‘actors’ can customise its behaviour. This allows an end user of a system built with a framework to be empowered to extend and tailor the system in a similar way to a component developer. This both increases the system’s flexibility and reduces its maintenance requirement.

Introduction

One of the central tenets of object technology adoption has been the promise of reuse, but this has proved difficult to deliver in practice. Only the most general of components, such as container classes and graphical libraries have found widespread use and acceptance. However, it is difficult to make arbitrarily extensible classes to provide the basis for reuse via inheritance and there exists a trade-off between reusability and tailorability [7] because the re-user’s requirements cannot be effectively anticipated. In practice, planning for reuse can only be achieved within constraints, where we to some extent anticipate the extensions that can be made to the given components. Because of this, the most effective route to reuse has been that of the framework, where the context of reuse is constrained within a particular domain. Object-oriented frameworks are a helpful technology to support the reuse of proven software architectures and implementations. The use of frameworks reduces the costs and improves the software quality [9]. Although domain specific, a framework differs from a *domain-specific architecture*, as the former is an object oriented design while the latter might not be [14].

[10] defines a framework as “a system that can be customised, specialised, or extended to provide more specific, more appropriate, or slightly different capabilities”. A framework provides a basic system model for a particular application domain within which specialised applications can be developed. It consists of already coded pieces of software which are reused, the so called *frozen spots* and the flexible elements, the *hot spots*, which allow the user to adjust the framework to the needs of the concrete application [18]. Because it inhabits a particular domain, it does not suffer from the excessive abstraction that would render

it too general for effective reuse. This is particularly true of ‘application specific frameworks’ such as those used in hardware control systems [21] where the application domain is narrow and focussed. Unlike class libraries, frameworks encapsulate control flows as well as object interfaces, providing a basis for the dynamic behaviour of a system as well as its structural characteristics.

If we accept that frameworks are an important aspect of object-oriented development, then we might ask ourselves what constitutes a framework architecture and how can we develop an effective framework that meets its users’ needs? In this paper we discuss the characteristics of frameworks, the roles of their developers and users and how they interact, and catalogue the issues and trade-offs that arise in framework development. From this perspective we develop a component-driven approach to framework design that stresses system flexibility at all phases of development, use and maintenance.

Performing an analysis of user requirements is seen as an essential aspect of the framework designer’s task, because meeting different requirements in different domains can often lead to radically different framework designs. We categorise the users of a framework into a number of roles, each of which needs to be provided with an appropriate component level interface. The union of these requirements is seen as the interface specification that the framework designer must provide.

Flexible Elements of Object Oriented Frameworks

Based on customisation characteristics object oriented frameworks fall into two main categories:

- White Box Frameworks
- Black Box Frameworks

In a white box framework the architecture of the framework is known to the component and application developers who build upon it. The complete design has to be documented because this knowledge is necessary to adapt the framework to a concrete application. The mechanism used to provide flexibility is usually limited to *inheritance*. The user, therefore, must have knowledge of the framework architecture in order to customise the framework to address the particular application.

Black box frameworks, in contrast, hide their internal structure. The user just knows the *hot spots* of the system and a general description of the framework’s usage rather than having a comprehensive knowledge of its architecture. Very often the mechanism used to provide flexibility in black box frameworks is *composition* (figure 1). Black box frameworks implement the *information hiding principle* of D.L. Parnas [17] best.

When the hot spots of a system are clear, the effort to build a white box framework (often by generalising from a number of complete applications) can be relatively small. The disadvantage of this type of framework is that the end user needs knowledge about the complete architecture in order to use it, implying a long learning curve and high risk of error. In contrast to these white box systems, the usage of a black box framework is easy because only the knowledge of flexible elements and the basic characteristics of the system is required. Black box frameworks, however, are harder to build than white box frameworks. It is more common to find the black box approach in application specific frameworks such as those encountered in systems for controlling machinery [21] [25] where software components tend to mirror hardware components that have predictable types and strong design constraints implied by the application domain. The approach is more problematic for more generic domain-specific frameworks where it is more difficult to anticipate the types of components that might be required.

In practice there are few pure white or black box frameworks, and in most cases some hot spots are developed using a white box approach while others use the black box approach. In the process of framework implementation white box elements can be refined to black box hot spots, and frameworks tend to mature by this process, beginning as white box frameworks and evolving to black box frameworks.

Our component driven approach to framework development supports both white and black box frameworks. The components of the frameworks may contain both elements in arbitrary combination.

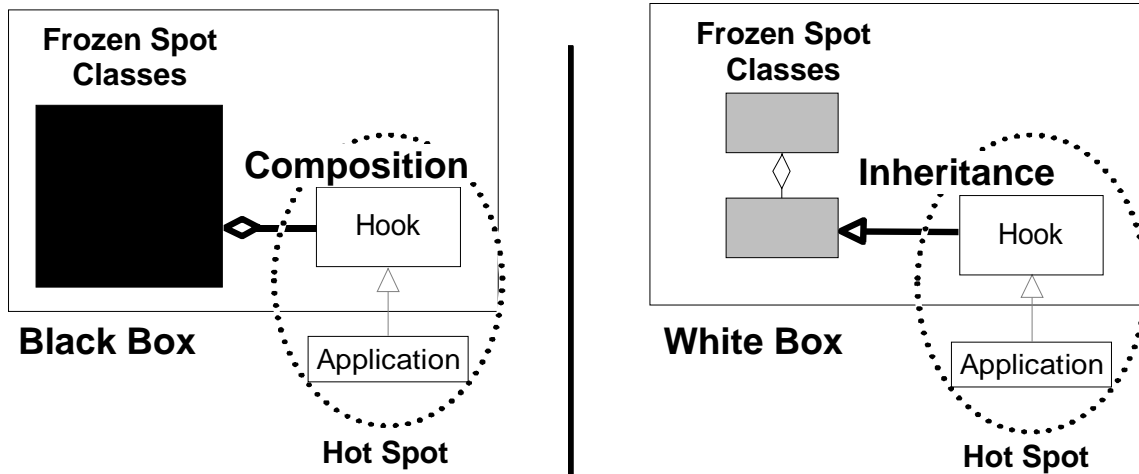


Fig. 1. Hot spots in object oriented frameworks: the black box approach and the white box approach

Actors and their Roles

The development of a framework-based application involves a number of roles. A framework consists of its underlying architecture, plus additional domain specific components combined to produce an application. There is a clear distinction between frameworks and components, although both are at the heart of software reuse techniques. A framework is more customisable and has a more complex interface than a component. While components provide reuse in terms of the intrinsic constructs of some object-oriented language (e.g. inheritance, templates), frameworks define 'super-language' reuse mechanisms which, being domain-specific, perform better than reuse at the language level. Furthermore, a framework can act as the *backbone* to provide a standard way for components to interact. Such a framework also provides facilities to build new components using the existing ones [14]. An end user of an application is dependent on both of these aspects of the system. Based on these different aspects of framework, component and application development, we identify four 'actor' (developer and user) categories. These are:

- Framework Developer
- Component Developer
- Application Developer
- End User

Figure 2 depicts the roles of these various actors. The solid arrows represent the usual roles these actors adopt. The component developer acts as the user of the framework built by the framework developer. He/she uses the framework to build/add new components that are employed by the application developer to create an application for the end user. Consider as an example a graphical user interface framework (built by the framework developer) extended by other developers (component developers) who provide add-on GUI components, allowing application developers to build more powerful applications for end users.

The dashed block arrows in figure 2 indicate that the roles of these actors might not be as isolated as they appear. Individuals or groups may adopt multiple roles. Although the framework developer is primarily responsible for building the framework, his/her role may include some elements of component building. A framework developer is, therefore, essentially the first component developer, providing the initial components. Consider existing systems such as OLE, CORBA implementations and Java Beans that are actually frameworks. In these systems, the framework essentially acts as the backbone to provide interaction among components designed and built by the framework developer. Similarly the component developers, application developers and the end users could be the same people.

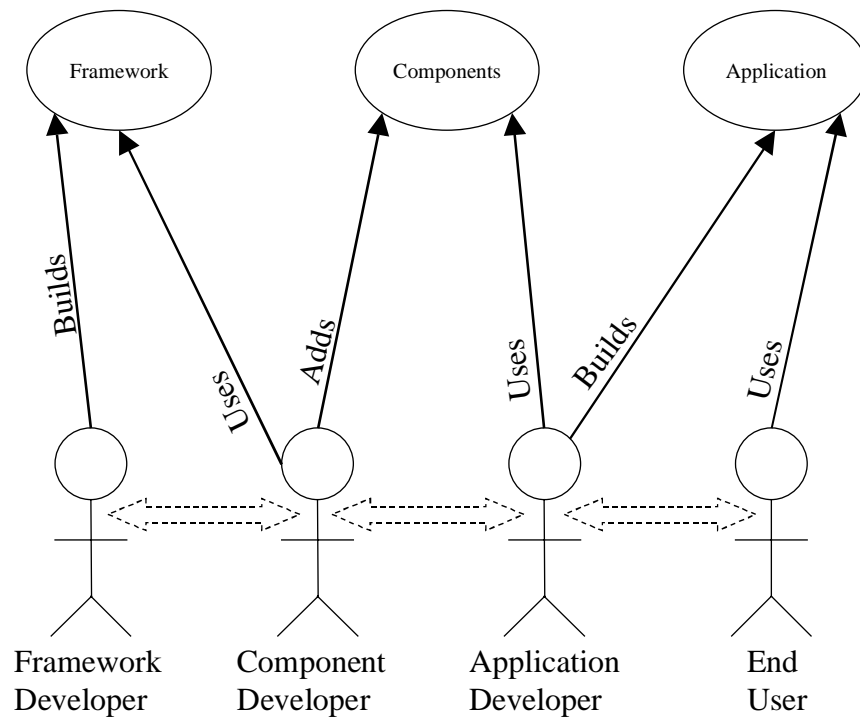


Fig. 2. The various developers/users and their roles in a system

The importance of these roles is that they have the effect of separating the end user from the framework developer; the requirements of the end user are not directly specified but are mediated through the various roles. This makes it difficult for the framework developer to anticipate the way an end user will interact with a final system based on the framework. Nevertheless this is an important consideration in framework design. In any case, there will be a requirement for a component building facility if the framework is to be a flexible and useful tool. The framework developer must build it into the core architecture (backbone) of the framework.

The core challenge for the framework developer is to provide tools for all actors. Although we see a series of roles that apparently separate the end user from the original framework developer, there is in fact no separation. The framework developer must build effective interfaces for all dependant roles, so even though the actors communicate with each other to pass requirements or get services, the framework developer has to implement all the mechanisms they ultimately are based upon.

In addition to providing a framework in which all the presented roles can work, the framework designer must provide mechanisms to allow an easy role transition, as it often happens that the same person needs to frequently switch roles. For example a scientist may write some components as a component developer, then assemble them into an application to test them as an application developer, then ultimately experiment with the application as an end user. To provide a common interface for all these roles, the framework designer must anticipate the union of the user-roles' requirements and make their communication a simple task.

The “Framework”: a Component-Driven Approach

Flexibility

Looking at several frameworks from different application domains from the flexibility perspective, we notice that, regardless of the application domain, these form a continuum between two extremes, one represented by fully compiled systems and the other by fully interpreted systems. Figure 3 shows the relationship between framework flexibility, implementation complexity, run-time performance and the compiled/interpreted behaviour incorporated into the framework.

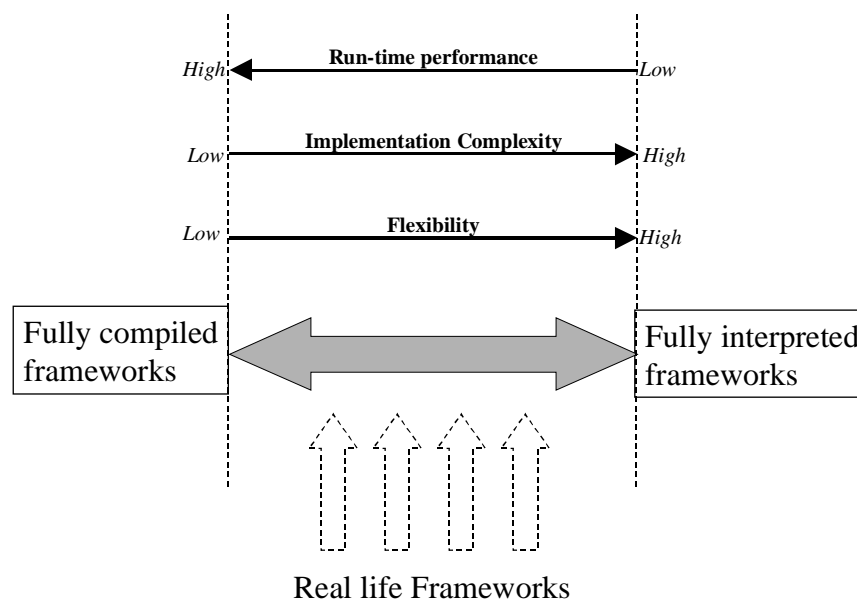


Fig. 3. The compiled/interpreted continuum

At one extreme of the spectrum in figure 3 are fully compiled frameworks while at the other are fully interpreted frameworks. Real life frameworks fall somewhere between the two extremes, and hence their developers have to make some trade-offs on what to hard code and what to leave to be determined at run-time. Since compiled frameworks tend to be *fixed*, they are less flexible than extensible interpreted frameworks. The degree of flexibility, therefore, depends upon the extent to which compiled/interpreted behaviours are present in the system. It is generally the choice of the right mix between the “compiled” and the “interpreted” amounts that makes a certain framework flexible enough without increasing, to a large extent, the complexity of the framework development process. As shown in figure 3 the more flexibility the framework developer wants to provide, the more complex his task will be (as a more complex machinery has to be set up to cope with the run-time “interpreted” decisions), and probably the slower the final system will be. Striking the right balance between the compiled and interpreted functionality is what makes a framework successful. Typically we find a range of interfaces to such systems for the various roles previously identified, from inheritance based interfaces that require extensive coding to composition based interfaces that simply require the instantiation and combination of predefined objects. We also see a demarcation between the level of exposure to the inner workings of the frameworks that different actors

will have, since an inheritance based framework implies that component developers must write new classes, while end users may simply use these classes. Thus the interface between the various roles and the fundamental framework architecture is inconsistent.

A component driven approach, in contrast, does not have to strike the right balance between the compiled/interpreted behaviour at the interface with various roles. Components can plug-in to the existing framework architecture regardless of whether the underlying architecture is compiled or interpreted because these implementation concerns remain encapsulated within the framework 'backbone'. A degree of flexibility, extensibility and run-time performance at least as high as a more exposed framework architecture can be achieved while at the same time the complexity of framework implementation does not increase with the degree of flexibility required. No run-time performance trade-offs are required either.

Elements of a Component-Based Framework

Our component-driven approach aims at building frameworks with highly cohesive but loosely coupled components. These components can then be developed independently of each other and are exchangeable hence providing a high degree of flexibility to the framework.

As shown in figure 4 components, in our framework approach, fall into three categories:

1. Framework Backbone
2. Basic Components
3. Additional Components

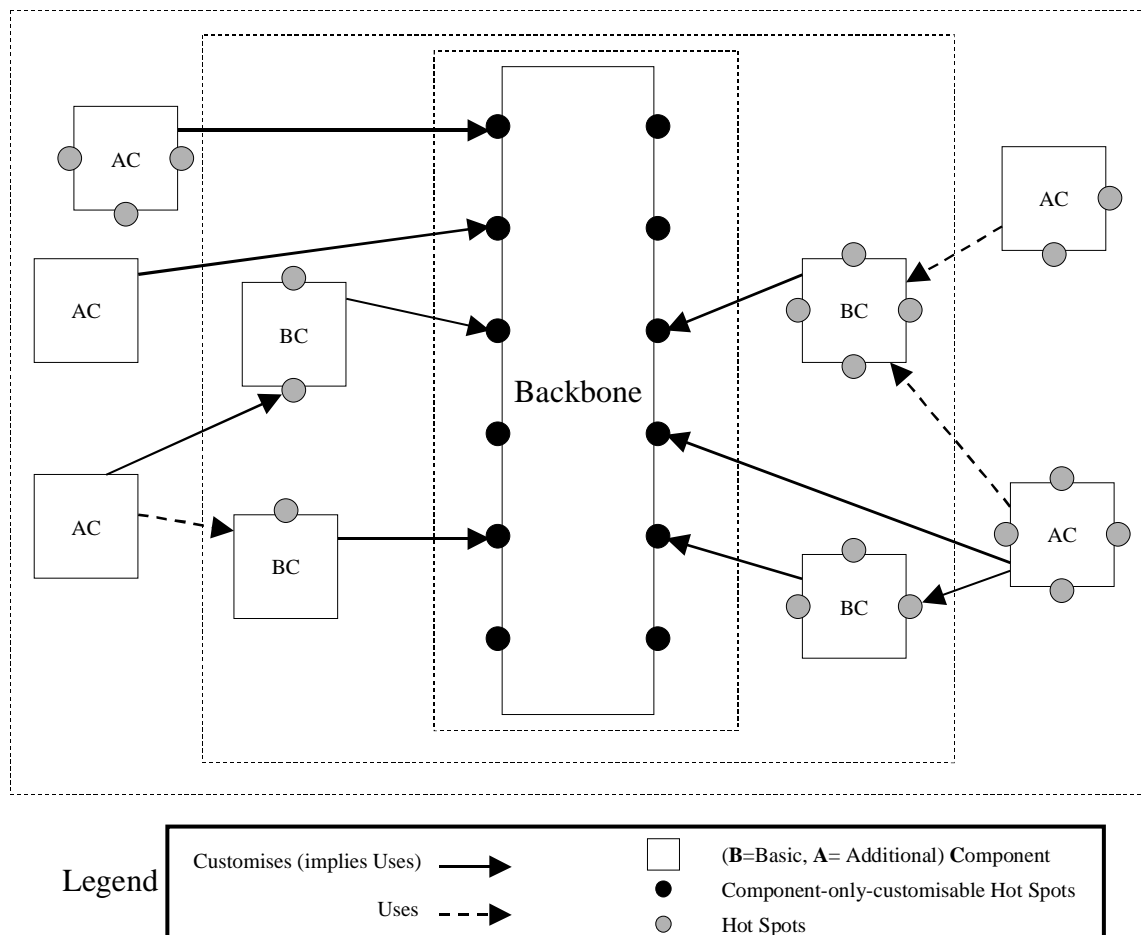


Fig. 4. The framework backbone and the various components plugged-in to it

The backbone is the basic component of a framework. It depends very much on the application domain. In many cases it provides communication and data exchange features as well as synchronisation mechanisms. The backbone has hot spots where the various components can be plugged-in.

Basic components contain the basic functionality of the framework and can be developed either by the framework developer or the component developer. Through the *component-only-customisable* hot spots basic components use the backbone for data exchange and communication. The framework backbone and the basic components are the mandatory elements of a complete framework.

The additional components are not necessarily part of a framework. They may be added by the component developer or application developer in order to adapt the framework to additional requirements.

The methods of the basic and additional components are called by the backbone according to the callback principle. The application can only customise these components and not the backbone directly. The backbone is used only by the basic and additional components.

It is worth noting that the modular encapsulation of the framework in components supports the reuse. The backbone as well as the components can be easily reused by other systems.

The Development Process

The development process is based on a pipeline of user analysis, user categorisation and user requirements to give us framework design guidelines for a given domain. This methodology will, when given a set of users, ultimately produce some guidelines for designing a framework that can support the various kinds of requirements they have and ensure their communication.

As shown in figure 5 the framework development process is incremental and iterative. It comprises the following phases.

1. Identify the domain and the specific part of the domain the framework aims to address
2. Develop the framework backbone (and the hot spots for the backbone. These are the ones where the various components plug-in)
3. Develop the *basic components* and their hotspots
4. *Additional components* can be developed by the component developer, application developer or the end-user

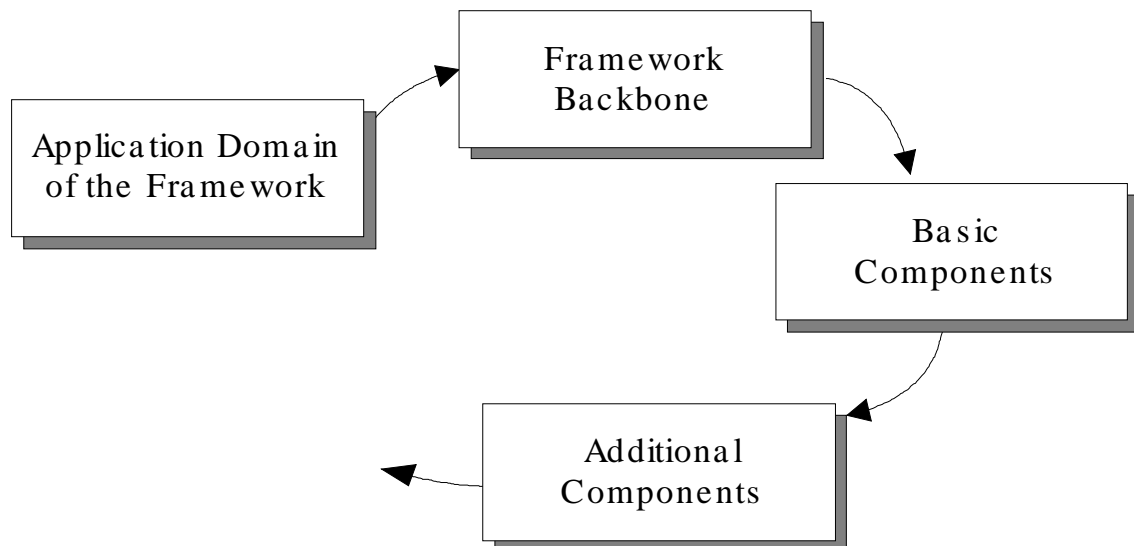


Fig. 5. The framework development process

As discussed below in each phase specific components are developed:

1. Application Domain of the Framework

The application domain and the specific part of the domain the framework aims to address is identified. The requirements of the framework have to be specified as a base for the development [23]. It is not necessary to start with an existing object-oriented design but it is necessary to include all actors' requirements in this analysis.

2. Framework Backbone

When the application domain is specified the framework developer can now begin to develop the framework backbone and the hot spots for the components. Most of the implementation work is done here (see the implementation section for a discussion of techniques).

3. Basic Components

The basic components and their hot spots have to be adjusted to the backbone. These components are developed or adapted after the backbone exists. A crucial aspect is that the backbone provides a standard component interface requirement that all components, both basic and additional, will conform to. This does not constrain the component developer as these interfaces are inferred directly from his/her requirements in step 1.

4. Additional Components

These are not part of the mandatory set of framework components. They are usually developed in accordance with special requirements from the end users. Therefore they can be developed by the component developer, application developer or the end user. It is an important aspect of the approach that end users have the power to extend the system as well as the component and application developers.

The phases of the process have no strict boundary like in the waterfall model [5]. As in other object-oriented software development processes (e.g. [3]) the borders between the phases are very weak and designers may work in two phases on different components at the same time.

Backbone Implementation Issues

The core difficulty in building such a component-based framework is the architecture and implementation of the backbone. Object-oriented design naturally gives us a foothold for component-based development by defining the class as the component building block, and inheritance and aggregation as basic tools for the assembly of these building blocks. In addition a certain amount of run-time flexibility is provided by the dynamic dispatch mechanism present in object-oriented languages, which takes a step to postponing some of the execution decisions to run-time. Class based components are however still developed at 'compile time', i.e. they can not be changed after the framework is running, and the set of components known by the framework is fixed during its operation. A step towards run-time flexibility is made by frameworks that allow the dynamic loading of new component types. Components are still not modifiable in the framework, but new components can be loaded in an existing framework without needing to stop (and maybe recompile) it. This type of framework introduces the notion of an application developer and of an application development environment, as these operations can be now executed dynamically by the framework itself (the component notion is now fully separated from the framework notion). Most scientific simulation, visualisation, animation, virtual reality and similar frameworks fall into this category. It is, however, important to note that in many cases the development of components and their usage in the framework are done using different and sometimes not fully compatible mechanisms. The components can be developed in a certain 'compiled' programming language and then used by the environment using some other 'interpreted' programming language. This frequently creates a problem that the abstractions of the 'compiled' language often cannot be mapped directly to the 'interpreted' language, so the process of making the components available to the application developer can be far from automatic. Several ad-hoc techniques have proliferated in many such 'dual language' systems in order to simulate a dynamic type system using components created with a language that has a compile-time type system. These have included Java or tcl-based class wrappers for C++ or Objective C classes [22] [13], exemplar simulation techniques for C++ classes like the ones presented by [6], adapter classes generating parallel class

hierarchies [11], run time type information mechanisms and various other approaches. All these techniques are inherently complex, hard to automate, and impose several restrictions on the component developer (e.g. having to inherit from a common root class, being restricted to single inheritance, being able to manipulate objects only by reference, etc.). A further problem is that component developers who change roles into application designers have to use two different and usually not fully isomorphic languages for the same abstractions, the 'interpreted' language for the application design and the 'compiled' language for the component design.

Single language frameworks remedy many of these problems because the components created by the component designer are directly understood by the framework. As an extra advantage, the application designer can often use the same language used for component development in 'interpreted' mode to dynamically define his problems so the transition between roles becomes easier. The implementation of such frameworks (and of additional debugging, automatic documentation and/or GUI generation tools) is often supported by features provided by the compiled/interpreted underlying language, such as reflection programming interfaces, persistence mechanisms, bytecode generation and just-in-time compilation, etc. The power of such frameworks is basically restricted only by the power of the underlying programming language. Good examples are Java and Java Beans based frameworks, the application development framework presented by [15] or the C++ interpreter system presented by [12]. Other variants include frameworks based on fully interpreted languages or typeless languages such as Lisp or Smalltalk. They suffer however from a performance problem and they are not attractive for component developers who have to interface to C/C++/Fortran legacy code.

In conclusion, we see that in order to support the various requirements posed by the end user, application developer and component developer roles, and also to make the role transition an easy, automatic process, frameworks have to defer most of their operations to run time. Providing support for the dynamic operation execution that is required often implies a form of 'interpreted' language execution. We believe that the key to the design of an effective framework for a certain application domain is the correct identification of the requirements of the 'actors', which determines the 'interpreted' to 'compiled' ratio in the system architecture. Deciding this immediately after the role analysis phase can save the framework designer from many unpleasant surprises, such as seeing that a given implementation lacks the desired run-time flexibility needed to support the required roles. The key feature of the analysis must be the interface required by the component designer, since this is likely to be the most rigorous requirement. The application developer and end user roles will then be empowered by having access to the same component interface, though this may be deliberately constrained in some way to maintain overall system cohesion.

Comparison with Earlier Work

Various authors have described a number of general aspects of framework design. The 'lifecycle' of a framework is generally assumed to be a reverse engineered solution similar to pattern mining, i.e. we develop three or more domain specific solutions and then attempt to factor out the common framework [20]. Once we have this framework we can use it to forward engineer further solutions within the same domain.

The majority of frameworks fall into the 'white-box' category, where some elements of internal implementation are exposed, typically a set of base classes from which concrete, domain specific classes can be instantiated. These will override certain methods of their base classes to provide specialised behaviour. Separation of abstract and concrete classes in the framework can assist here in isolating the 'hot spots' of a system into a small number of classes [18]. Some frameworks are 'black box', where components are slotted into an architecture that uses object composition as a means of extension rather than the addition of new classes [19]. In both contexts the creation or composition of the system is seen as purely a developer role, building a final software solution for an end user.

There has been some analysis of the roles of those involved at different stages of framework development and use [4] but these follow a traditional model in that there is seen to be a distinct difference between those who develop frameworks and applications and those who utilise the final product. Not only are they assumed to be different people, but also the ways that they interact with the system are fundamentally different. Framework developers, framework users (i.e. application developers) and framework maintainers are involved with the detail of the architectures whereas end users are assumed to treat the end product as if

it were any other piece of software. Indeed end users are not even included in this example of role analysis. Roles are also generally seen as discrete, acting within different layers of the architecture [2] [8].

There are two major limitations to these approaches. First, there is an assumption that a framework is simply a tool for creating the same kind of application that would be developed without a framework. However, one advantage of a framework architecture is its inherent flexibility, a quality that should be carried over into the final product. Frameworks, like any other software, are subject to change over time [1], so flexibility at the application level can reduce the maintenance required. Second, the use of white box frameworks is problematical because they rely on additional source code and compilation tools for extension. This limits the scope for extensibility to the component designer / application designer roles. A better approach is one that takes a black box approach to framework design that is carried through to the end user role. For this to be effective black box framework must be heavily configurable via a common interface at all development stages.

The framework design approach described here recognises through role analysis that a range of activities are required by various actors using the basic application framework but that these activities may well overlap and inform each other. Therefore rather than fixing phases of development where an open framework is closed by a component developer and/or application developer, we have an open process where the framework provides an extensible interface for all actors, including application end-users. This interface should not require knowledge of implementation detail but rather should be at the component level. To change behaviours we may need 'horizontal' interfaces with meta-level support [16] but should not expose recompilation processes. Meta level interfaces might be via visual tools, scripting languages or other means of modifying the system that do not involve direct extension of the framework code.

A framework developed with a flexible role model as described here is one that provides a common interface for component developers, application developers and end users. A single user can thus move transparently between roles using a common set of tools. This should result in more dynamically configurable systems that require less initial work in component development, less maintenance and provide more powerful tools for the end-user.

Summary and Conclusions

The traditional framework development lifecycle involves a set of user requirements met with some combination of compiled and/or interpreted implementation strategies. The style of interface is usually an ad hoc arrangement, perhaps in a permanent state of flux as a reverse engineered white box framework gradually evolves into a more stable black box framework

The development strategy described here begins by recognising that there are several roles in the development and use of a framework, but that the original framework developer is in fact meeting a consistent need for extensibility, preferably with a simple interface that can cater for run time component creation and selection. Thus the user requirements are translated into component interfaces that define the necessary backbone support. The implementation as before must strike an appropriate balance between the compiled and the interpreted but encapsulated within the backbone, not exposed to the other developers/users. The advantage of this approach is that the component notion decouples framework construction from application domain analysis via the production of the component interface. In this way, several application domains can be satisfied by designing several frameworks, all being in fact instances of the same 'meta-framework'. In each case the designer identifies a unique component interface appropriate to the domain.

This 'framework for framework design' depends on a 'logical pipeline', which starts at the end-user's requirements, passes through the application developer and the component developer's ones, and ultimately focuses the burden on the framework developer. Depending on the overall requirements for a component interface, the framework developer will have to provide a certain degree of (run-time) flexibility, the requirements determining the right 'mix' between compile-time and run-time system parts. We can be specific by examples only, as the range of requirements is too large to offer a general solution. However, we can elicit a suitable set of guidelines once we have identified the union of the users' requirements. For each requirement, there is a statement to make about the framework's implementation. For example, a need to introduce new instances at run-time implies the need for a basic interpreter with type instantiation capabilities. A more demanding requirement to introduce new types at run-time means that we need some

kind of interpreter with dynamic type-loading capabilities. If these new types are complex enough to introduce new code at run-time, then we need an interpreter and/or run-time operating system support for dynamic code loading. For the most complex frameworks however where we may need to modify types, instances and code at run-time there are several solutions, such as incremental/on-the-fly compilers, meta-types, single hierarchy (meta-type based) languages, etc. The requirements inform us of the demands likely to be placed on the framework's backbone, and the result is a framework design specification or guideline that provides the optimum cost/benefit trade-offs for a given framework.

Since the kernel of the component-based framework is the backbone this is where all the hard design decisions have to be taken. The way the backbone interfaces with the components gives us the level of framework flexibility. The more 'loosely coupled' the component-framework communication/interface is, the easier is everything for all the user categories, but we limit the strategic design decisions to the backbone alone. Simply put, the framework designer must implement a backbone that can satisfy the component interface(s), derived from the union of the user specifications.

References

- [1] Adair, D., "Building Object-Oriented Frameworks", *AIXpert, Feb. & May 1995*
- [2] Astheimer, P. (1993). "Sonification tools to supplement dataflow visualization" in "Scientific Visualization: Advances and Challenges", Academic Press (1994), pp. 251-263.
- [3] Booch, G., "Object-Oriented Analysis and Design", Benjamin/Cummings, Redwood City, CA, second edition, 1994
- [4] Bosch, J. *et al.*, "Object-Oriented Frameworks - Problems & Experiences",
<http://www.ide.hk-r.se/~michaelm/papers/ex-frame.ps>
- [5] Royce, W. W., "Managing the Development of Large Software Systems: Concepts and Techniques", Proceedings IEEE WESTCON, Los Angeles, 1-9
- [6] Coplien, J., "Advanced C++: Programming Styles and Idioms" Addison-Wesley, 1992
- [7] Demeyer, S. *et al.*, "Design Guidelines for 'Tailorable' Frameworks", *Communications of the ACM, Vol. 40, No.10, Oct. 1997, pp. 60-65*
- [8] Duclos, A. M. & Grave, M. (1993). "Reference models and formal specification for scientific visualization" in "Scientific Visualization: Advances and Challenges", Academic Press (1994), pp. 251-263.
- [9] Fayad, M. E., & Schmidt, D.C., "Object-Oriented Application Frameworks", *Communications of the ACM, Vol. 40, No.10, Oct. 1997, pp. 32-38*
- [10] Gabriel, R. P., "Patterns of Software - Tales from the Software Community", *Oxford University Press, New York, c1996*
- [11] Gamma, E., Helm, R., Johnson, R. & Vlissides, J., "Design Patterns: elements of reusable object-oriented software", Addison-Wesley, 1995.
- [12] Brun, R., & Rademakers, S., "ROOT - An Object Oriented Data Analysis Framework", Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys.Res. A 389 (1997) 81-86. See also <http://root.cern.ch/>
- [13] Gunn, C., Ortmann, A., Pinkall, U., Polthier, K. & Schwarz, U., "Oorange: A Virtual Laboratory for Experimental Mathematics", Visualization and Mathematics, editors H.C. Hege and K. Polthier, Springer Verlag 1997, pp. 249-267
Also see <http://www-sfb288.math.tu-berlin.de/oorange/OorangeDoc.html>
- [14] Johnson, R. E., "Frameworks = (Components + Patterns)", *Communications of the ACM, Vol. 40, No.10, Oct. 1997, pp. 39-42*
- [15] Meyer, B., "Object-oriented software construction", Prentice Hall, 1997
- [16] Mowbray, T. & Malveau, R., "CORBA Design Patterns", *New York: Wiley, 1997*
- [17] Parnas, D. L., "On Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM, Vol. 15, No.12, Dec. 1972, pp. 1053-1058*
- [18] Pree, W., "Design Patterns for Object Oriented Software Development", *Addison Wesley, 1994*
- [19] Richards, B., "Frameworks and Design Patterns" in *Rising, L. (ed.) "The Patterns Handbook: Techniques, Strategies and Applications"*, Cambridge: Cambridge University Press, 1998
- [20] Roberts, D. & Johnson, R., "Patterns for Evolving Frameworks" in *Martin, R. et al (eds.), "Pattern Languages of Program Design 3"*, Reading, Mass: Addison-Wesley, 1998
- [21] Schmid, H. "Design Patterns to Construct the Hot Spots of a Manufacturing Framework" in *Rising, L. (ed.) "The Patterns Handbook: Techniques, Strategies and Applications"*, Cambridge: Cambridge University Press, 1998
- [22] Schroeder, W., Martin, K. & Lorensen, B., "The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics", Prentice Hall, 1995
- [23] Sommerville, I., & Sawyer, P., "Requirements Engineering: a Good Practice Guide", John Wiley and Sons, 1997

- [24] Upson, C., Faulhaber, T., Kamins, D., Laidlaw, D., Schlegel, D., Vroom, J., Gurwitz, R. and van Dam, A., "The application visualization system: a computational environment for scientific visualization". IEEE Computer Graphics and Applications July 1989, p.30-42
- [25] Wright, C. & Coutts, I., "Applying Design Patterns to the Control System Framework for a Tube Inspection Machine", *Proceedings of Patterns 98, Manchester 28/29 October 1998*