SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2001; **31**:277–300

Design, implementation and evolution of object oriented frameworks: concepts and guidelines[‡]

J. van Gurp[*,†] and J. Bosch

Department of Mathematics and Computer Science, University of Groningen, PO Box 800, 9700 AV Groningen, The Netherlands

SUMMARY

Object-oriented frameworks provide software developers with the means to build an infrastructure for their applications. Unfortunately, frameworks do not always deliver on their promises of reusability and flexibility. To address this, we have developed a conceptual model for frameworks and a set of guidelines to build object oriented frameworks that adhere to this model. Our guidelines focus on improving the flexibility, reusability and usability (i.e. making it easy to use a framework) of frameworks. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: object oriented frameworks; role-based modeling; design guidelines

1. INTRODUCTION

Object-oriented frameworks are becoming increasingly important for the software community. Frameworks allow companies to capture the commonalities between applications for the domain they operate in. Not surprisingly the promises of reuse and easy application creation sound very appealing to those companies. Studies in our research group [1–5] show that there are some problems with delivering on these promises, however.

The term *object-oriented framework* can be defined in many ways. A framework is defined in [1] as a partial design and implementation for an application in a given domain. So in a sense a framework is an incomplete system. This system can be tailored to create complete applications. Frameworks are generally used and developed when several (partly) similar applications need to be developed.

*Correspondence to: J. van Gurp, Department of Mathematics and Computer Science, University of Groningen, PO Box 800, 9700 AV Groningen, The Netherlands.
†E-mail: jilles@cs.rug.nl
‡This article was mostly written at the University of Karlskrona/Ronneby in Sweden.

Received 1 February 2000
Revised 3 May 2000 and 30 September 2000
Accepted 21 October 2000

A framework implements the commonalities between those applications. Thus, a framework reduces the effort needed to build applications [5]. We use the term framework instantiation to indicate the process of creating an application from a specific framework. The resulting application is called a framework instance.

In a paper by Taligent (now IBM) [6], frameworks are grouped into three categories:

1. *Application frameworks*. Application frameworks aim to provide a full range of functionality typically needed in an application. This functionality usually involves things like a GUI, documents, databases, etc. An example of an application framework is MFC (Microsoft Foundation Classes). MFC is used to build applications for MS Windows. Another application framework is JFC (Java Foundation Classes). The latter is interesting from an object oriented (OO) design point of view since it incorporates many ideas about what an OO framework should look like. Many design patterns from the GoF book [7] were used in this framework, for instance.
2. *Domain frameworks*. These frameworks can be helpful to implement programs for a certain domain. The term domain framework is used to denote frameworks for specific domains. An example of a domain is banking or alarm systems. Domain specific software usually has to be tailored for a company or developed from scratch. Frameworks can help reduce the amount of work that needs to be done to implement such applications. This allows companies to make higher quality software for their domain while reducing the time to market.
3. *Support frameworks*. Support frameworks typically address very specific, computer related domains such as memory management or file systems. Support for these kinds of domains is necessary to simplify program development. Support frameworks are typically used in conjunction with domain and/or application frameworks.

In earlier papers in our research group [1,3] a number of problems with mainly domain specific frameworks are discussed. These problems center around two classes of problems:

1. *Composition problems*. When developing a framework, it is often assumed that the framework is the only framework present when applications are going to be created with it. Often, however, it may be necessary to use more than one framework in an application. This may cause several problems. One of the frameworks may, for instance, assume that it has control of the application it is used in and may cause the other frameworks to malfunction. The problems that have to be solved when two or more frameworks are combined are called composition problems. An Andersen Consulting study [8] claims that *almost any OO project must buy and use at least one framework to meet the user's minimum expectations of functionality*, indicating that nearly any project will have to deal with composition problems.
2. *Evolution problems*. Frameworks are typically developed and evolved in an iterative way [4] (like most OO software). Once the framework is released, it is used to create applications. After some time it may be necessary to change the framework to meet new requirements. This process is called framework evolution. Framework evolution has consequences for applications that have been created with the framework. If APIs in the framework change, the applications that use it have to evolve too (to remain compatible with the evolving framework).

In [4] and [8], a number of other problems regarding framework deployment, documentation and usage are discussed. In [9] it is argued that a reason for framework related problems is that the conventional way of developing frameworks results in large, complex frameworks that are difficult to design, reuse and combine with other frameworks. In addition to that we believe that these problems are caused by the fact that frameworks are not prepared for change. Yet, change is inevitable. New requirements will come and the framework will have to be changed to deal with them. One of the requirements may be that the framework can be used in combination with another framework (composition). If a framework is not built to deal with changes, radical restructuring of the framework may be necessary to meet new requirements. To avoid this, developers may prefer a quick fix that leaves the framework intact. Unfortunately this type of solution makes it even more difficult to change the framework in the future. Consequently, over time these solutions accumulate and ultimately leave the framework in a state where any change will break the framework and its instances.

In this article we present guidelines that address the mentioned problems. Our guidelines are largely based on experiences accumulated during various projects in our research group, e.g. [1,3,10]. Our guidelines aim to increase flexibility, reusability and usability. In order to put the guidelines to use, a firm understanding of frameworks is necessary. For this reason we also provide a conceptual model of how frameworks should be structured.

The remainder of this article is organized as follows. In Section 2 we introduce our running example: a framework for haemo dialysis machines. In Section 3 we elaborate on framework terminology and methodology and we introduce a conceptual model for frameworks. This provides us with the context for our guidelines. In Section 4 we introduce our guidelines for improving framework structure. Section 5 provides some additional recommendations, addressing non-structure-related topics in framework development, and in Section 6 we present related work. We also link some of our guidelines to related work. We conclude our paper in Section 7.

2. THE HAEMO DIALYSIS FRAMEWORK

In this section we will introduce an example framework that we will use throughout the paper. As an example we will use the haemo dialysis framework that was the result of a joint research project with Althin Medical, EC Gruppen and our research group [10]. The framework provides functionality for haemo dialysis machines (see Figure 1).

Haemo dialysis is a procedure where water and natural waste products are removed from a patient's blood. As illustrated in Figure 1, the patient's blood is pumped through a machine. In this machine, waste products and water in the blood go through a filter into the dialysis fluid. The fluid contains minerals which go through the filter into the patient's blood. The haemo dialysis machine contains all sorts of control and warning mechanisms to prevent any harm being done to the patient.

These mechanisms are controlled by the aforementioned framework. The framework offers support for different devices and sensors within the machine and offers a model of how these things interact with each other. Important quality requirements that need to be guaranteed are safety, real-time behavior and reusability.

In Figure 2, part of the framework is shown. In this figure the interfaces of the so-called logical archetypes are shown. Using these interfaces, the logical behavior of the components in a dialysis system can be controlled. Apart from the logical behavior, some additional behavior is required of components in the system. This additional behavior can be accessed through interfaces from support frameworks. In the article describing the haemo dialysis architecture [10], two support frameworks are

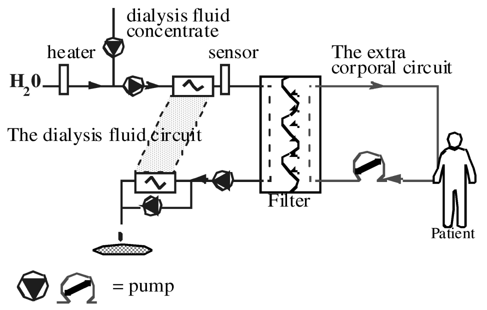
dialysis fluid concentrate
heater
sensor
The extra corporal circuit
H_2O
The dialysis fluid circuit
Filter
Patient
= pump

Figure 1. The haemo dialysis machine.

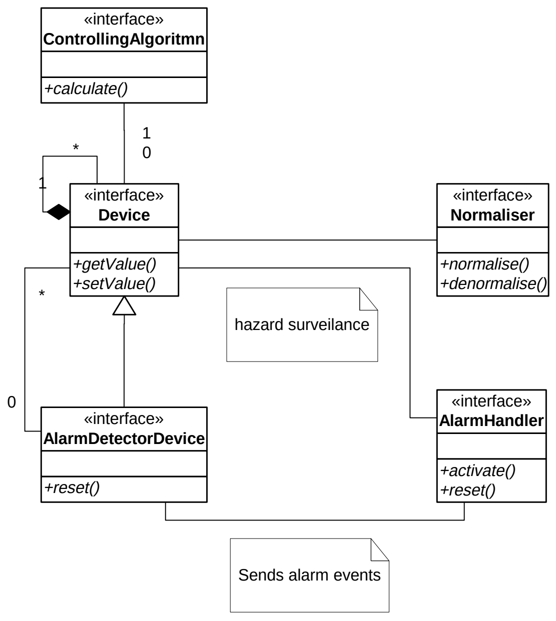

Figure 2. The haemo dialysis core framework.

described: an application-level scheduling mechanism and a mechanism to connect components (see Figure 3).

So, the entire framework consists of three smaller frameworks that each target a specific domain of functionality. Applications that are implemented using this framework provide application specific components that implement these interfaces. The components in the application are, in principle,

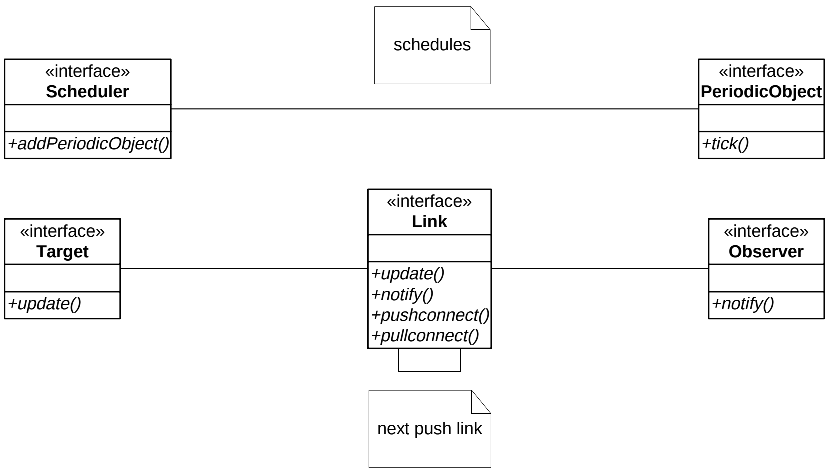

Figure 3. The scheduling and connector support frameworks.

reusable in other applications. A temperature sensor software component built for usage in a specific machine, for example, can later be reused in the software for a new machine. Even the use outside the narrow domain of haemo dialysis machines is feasible (note that there are no dialysis specific interfaces).

3. FRAMEWORK ORGANIZATION

Most frameworks start out small: a few classes and interfaces generalized from a few applications in the domain [11]. In this stage the framework is hard to use since there is hardly any reusable code and the framework design changes frequently. Usually, inheritance is used as a technique to enhance such frameworks for use in an application. When the framework evolves, custom components are added that cover frequent usage of the framework. Instead of inheriting from abstract classes, a developer can now use the predefined components, which can be composed using the aggregation mechanism.

In Szyperski [12], blackbox reuse is defined as the *concept of reusing implementations without relying on anything but their interfaces and specifications*. Whitebox reuse, on the other hand, is defined as *using a software fragment, through its interfaces, while relying on the understanding gained from studying the actual implementation*. Frameworks that can be used by inheritance only (i.e. that do not provide readily usable components) are called *whitebox frameworks* because it is impossible to use them (i.e. extend them) without understanding how the framework works internally. Frameworks that can also be used by configuring existing components are called *blackbox frameworks* since they provide components that support blackbox reuse. Blackbox frameworks are easier to use because the internal mechanism is (partially) hidden from the developer. The drawback is that this approach is less flexible. The capabilities of a blackbox framework are limited to what has been implemented in the set of provided components. For that reason, frameworks usually offer both mechanisms. They have a whitebox layer consisting of *interfaces* and textitabstract classes providing the architecture that can be used for whitebox reuse and a blackbox layer consisting of *concrete classes and components* that inherit from the whitebox layer and can be plugged into the architecture. By using the concrete classes, the developer has easy access to the framework's features. If more is needed than the default implementation, the developer will have to make a custom class (either by inheriting from one of the abstract base classes or by inheriting from one of the concrete classes).

3.1. Blackbox and whitebox frameworks

In Figure 4, the relations between different elements in a framework are illustrated. The following elements are shown in this figure.

1. *Design documents*. The design of a framework can consist of class diagrams (or other diagrams), written text or just an idea in the head of developers.
2. *Interfaces*. Interfaces describe the external behavior of classes. In Java there is a language construct for this. In C++ abstract classes can be used to emulate interfaces. The use of preprocessor directives, such as used in header files, is not sufficient because the compiler does not involve those in the type checking process (the importance of type checking when using interfaces was also argued in Pree and Koskimies [9]). Interfaces can be used to model the different roles in a system (for instance the roles in a design pattern). A role represents a small group of method interfaces that are related to each other.
3. *Abstract classes*. An abstract class is an incomplete implementation of one or more interfaces. It can be used to define behavior that is common for a group of components implementing a group of interfaces.
4. *Components*. The term component is a somewhat overloaded term. Therefore we have to be careful with its definition. In this article, the only difference between a component and a class is that the API of a component is available in the form of one or more interface constructs (e.g. Java interfaces or abstract virtual classes in C++). Like classes, components may be associated with other classes. In Figure 4, we tried to illustrate this by the *are a part of* arrow between classes and components. If these classes themselves have a fully defined API, we denote the resulting set of classes as a *component composition*. Our definition of a component is influenced by Szypersi's discussion on this subject [12]. '*A software component is a unit of composition with contractally specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*'. However, in this definition, Szyperski is talking about components in general, while we limit ourselves to object oriented components. Consequently, in order to fulfill this definition, an OO component can be nothing else than a single class (unit of composition) with an explicit API.
5. *Classes*. At the lowest level in a framework are the classes. Classes only differ from components in the fact that their public API (application programming interface) is not represented in the interfaces of a framework. Typically classes are used by components to delegate functionality to, i.e. a framework user will typically not see those classes since he/she only has to deal with components.

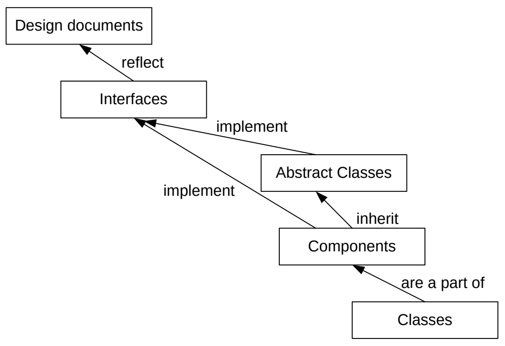

Figure 4. Relations between the different elements in a framework.

The elements in Figure 4 are connected by labeled arrows that indicate relations between these elements. Interfaces together with the abstract classes are usually called the whitebox framework. The whitebox framework is used to create concrete classes. Some of these classes are components (because they implement interfaces from the whitebox framework). The components, together with the collaborating classes, are called the blackbox framework.

The main difference between a blackbox framework and a whitebox framework is that, in order to use a whitebox framework, a developer has to extend classes and implement interfaces. A blackbox framework, on the other hand, consists of components and classes that can be instantiated and configured by developers. Usually the components and classes in blackbox frameworks are instances of elements in whitebox frameworks. Composition and configuration of components in a blackbox framework can be supported by tools and is much easier for developers than using the whitebox part of a framework.

3.2. A conceptual model for OO frameworks

Blackbox frameworks consist of components. In the previous section, we defined a component as a class or a group of collaborating classes that implement a set of interfaces. Even if the component consists of multiple classes, the component is externally represented as one class. The component behaves as a single, coherent entity. We make a distinction between *atomic* and *composed components*. Atomic components are made up of one or just a few classes, whereas a composed component consists of multiple components and gluecode in the form of classes. The ultimate composed component is a complete application which, for example, provides an interface to start and stop the application, a UI and other functionality.

A component can have different roles in a system. Roles represent subsets of related functionality that a component can expose [13]. A component may behave differently to different types of clients. That is, a component exposes different roles to each client. A button, for instance, can have a graphical role (the way it is displayed), at the same time it can have a dynamic role by sending an event when it is clicked on. It also has a monitoring role since it waits for the mouse to click on it.

Each role can be represented as a separate interface in a whitebox framework. Rather than referring to the entire API of a component, a reference to a specific role-interface implemented by the component can be used. This reduces the number of assumptions that are made about a component when it is used in a system (in a particular role). Ideally, all of the external behavior of a component is defined in terms of interfaces. This way developers do not have to make assumptions about how the component works internally but instead can restrict themselves to the API defined in the whitebox framework(s). The component can be changed without triggering changes in the applications that use it (provided the interface does not change).

The idea of using roles to model object systems has been used to create the OORam method [14]. In this method so called *role models* are used to model the behavior of a system. In our opinion this is an important step forward from modeling the system behavior in terms of relations between classes. An important notion of the role models in [14] is that multiple or even all of the roles in a role model may be implemented by just one class. Also it is possible for a class to implement roles from multiple role models. In addition it is possible to derive and compose role models.

A good example of components and roles in practice is the Swing GUI Framework in Java. In this complex and flexible framework the combination of roles and components is used frequently. An example of a role is the Scrollable role, which is present as a Java interface in the framework. Any GUI component (subclasses of JComponent) implementing this role can be put into a so-called JScrollpane which provides functionality to scroll whatever is put in the pane. Currently, there are four JComponents implementing the Scrollable interface out of the box (JList, JTextComponent, JTree and JTable). However, it is also possible for users to implement the Scrollable interface in other JComponent subclasses.

The Scrollable interface only contains five methods that need to be implemented. Because of this it is very simple for programmers to add scrolling behavior to custom components. The whole mechanism fully depends on the fact that the component can play multiple roles in the system. In fact all the Scrollpane needs to know about the component is that it is a JComponent and that it can provide certain information about its dimensions (through the Scrollable interface). Characteristic for the whole mechanism is that it works on a need-to-know basis. The Scrollpane component only needs to know a few things to be able to scroll a JComponent. All this information is provided through the Scrollable interface.

In [9] the notion of *framelets* is introduced. A framelet is a very small framework (typically no more than 10 classes) with clearly defined interfaces. The general idea behind framelets is to have many, highly adaptable small entities that can be easily composed into applications. Although the concept of a framelet is an important step beyond the traditional monolithic view of a framework, we think that the framelet concept has one important deficiency. It does not take into account the fact that there are components whose scope is larger than one framelet. As Reenskaug showed in [14], one component may implement roles from more than one role model. A framelet can be considered as an implementation of only one role model. Rather than the Pree and Koskimies view [9] of a framelet as a component we prefer a wider definition of a component that may involve more than one role model or framelet, as in [14].

Based on this analysis we created a conceptual model that prescribes how frameworks should be structured. In this model all frameworks use a common set of role models. Each framework uses a subset of these role models and provides *hotspots* [15] in the form of abstract classes and implementation in the form of components. In this model a framework is nothing but a set of related

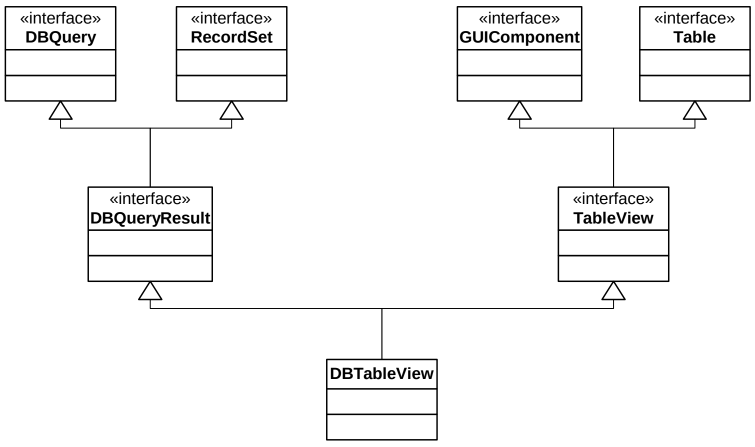

Figure 5. Example of two role models combined in a single component.

classes and components. Interoperability with classes and components from other frameworks is made easier because of the shared role models.

Traditionally, abstract classes have been used where we choose to use interfaces. Consequently the only reason why abstract classes should be used is to reuse implementation in subclasses. As we will argue in our guidelines section, there is no need to use abstract classes for anything else than that.

This way of making frameworks requires some consensus between the different parties that create the frameworks. In particular, there should be no ‘competing’ role models and roles. Instead, competition should take place on the implementation level where interchangeability of components is achieved through the role models they have in common. The enormous amount of API specifications (which are nothing but interfaces) for the Java platform that have appeared over the past few years illustrate how productive this way of developing can be.

As an example, (see Figure 5), consider the case where there is a small database role model, modeling tables and other database related datastructures, and a GUI role model, modeling things like GUI components, tables and other widgets. The simple components these two framelets provide will typically be things like buttons, a table, a tableview. A realistic scenario would be to combine those two frameworks to create database aware GUI components. As a matter of fact database aware GUI components are something Inprise (the former Borland) [16] put in their JBuilder tool on top of the existing Swing [17] GUI framework.

In our model doing such a thing is not that difficult since the already existing interfaces in the role models will need little or no change. Furthermore, interoperability with the two existing frameworks also comes naturally since the new framework for database aware GUI components will implement the same roles as those implemented in the two other frameworks.

The haemo dialysis framework is organized in more or less the same fashion as described above. There are three role models:

(i) a role model that models the logical entities in the domain (devices, alarm mechanisms, etc.);
(ii) a scheduling policy role model;
(iii) a role model for connecting components.

Each of these role models is small, highly specialized and independent of the other role models. We want to create useful components, i.e. components that implement interfaces from the logical entity framework and that can be connected to other components in the system and that can be scheduled. Framelet components, such as suggested in [9], are not enough since they are limited to only one of the role models. A typical component in the system will implement roles from all three role models. This does not mean that framelet components are useless. In fact the composed components can delegate their behavior to framelet components. However, we think that limiting a component to only one role model is not very useful.

3.3. Dealing with coupling

From our earlier research in frameworks we have learned that a major problem in using and maintaining frameworks are the many dependencies between classes and components. More coupling between components means higher maintenance cost (McCabe's cyclomatic complexity [18], Law of Demeter [19]). So we argue that frameworks should be designed in such a way that there is minimal coupling between the classes and components.

There are several techniques to allow two classes to work together. What they have in common is that for object X to use object Y, X will need a reference to Y. The techniques differ in the way this reference is obtained. The following techniques can be used to retrieve a reference.

1. Y is created by X and then discarded. This is the least flexible way of obtaining a reference. The type of the reference (i.e. a specific class) to Y is compiled into the class specifying X, and there is no way that X can use a different type of Y without editing the source code of X's class.
2. Y is a property of X. This is a more flexible approach because the property holding a reference to Y can be changed at run-time.
3. Y is passed to X as a parameter of some method. This is even more flexible because the responsibility of obtaining a reference no longer lies in X's class.
4. Y is retrieved by requesting it from a third object. This third object can, for instance, be a factory or a repository. This technique delegates the responsibility of retrieving the reference to Y to a third object.

A special case of technique 3 is the delegated *event mechanism* such as that in Java [17]. Such event mechanisms are based on the Observer pattern [7]. Essentially this mechanism is a combination of the second and the third technique. First Y is registered as being interested in a certain event originating from X. This is done using technique 3. Y is passed to X as a parameter of one of X's methods and X stores the reference to Y in one of its properties. Later, when an event occurs, X calls Y by retrieving the previously stored reference. Components notify other components of certain events and those components respond to this notification by executing one of their methods. Consequently the event is de-coupled from the response of the receiving components. We also refer to this way of coupling as *loose coupling*.

Regardless of the way the reference is obtained there are two types of dependencies between components:

1. *Implementation dependencies*: The references used in the relations between components are typed using concrete classes or abstract classes.
2. *Interface dependencies*: The references used in the relations between components are typed using only interfaces. This means that in principle the component's implementation can be changed (as long as the required interfaces are preserved. It also means that any component using a component with interface X can use any other component implementing X.

The disadvantage of implementation dependencies is that it is more difficult to replace the objects to which the component delegates. The new object must be of the same class or a subclass of the original object. When interface dependencies are used, the object can be replaced with any other object implementing the same interface. So, interface dependencies are more flexible and should always be preferred over implementation dependencies.

In the conceptual model we have presented, all components implement interfaces from role models. Consequently it is not necessary to use implementation dependencies in the implementation of these components. Using this mechanism is therefore an important step towards producing more flexible software.

3.4. Framework instantiation

Building an application using a framework structured using the approach we have presented in this section requires one or more of the following activities.

1. *Writing glue code*. In the ideal case, when the components in a framework cover all the requirements, the components just have to be configured and glued together to form an application. The glue code can either be written manually or generated by a tool.
2. *Providing application specific components.* If the components do not cover the requirements completely, it may be necessary to create application specific components. If this is done correctly, the new components may become a part of the framework. Once the components have been written, gluecode must be added.
3. *Providing application specific classes*. If the required functionality lies outside the scope of the framework, it may be necessary to create application specific classes. If this solution is chosen often for certain functionality, it may be worthwhile to create a new framework for it or incorporate the classes into the existing framework. In our framework model, the typical approach would be to create additional role models and use those to create new components.

To make application specific classes/components, developers have to extend the framework in the so-called *hotspots* [15]. In Parsons *et al.* [20] frameworks are made up of hotspots and frozen spots (flexible, extensible pieces of code and ready to use code). A hotspot may be one of the following.

1. *An interface in one of the role models.* The mechanism to use such hotspots is to provide classes that implement the interface. Interface hotspots do not lead to any code reuse and only enforce design reuse.
2. *An abstract class.* The mechanism to use these hotspots is inheritance. Classes inherit both interfaces and behavior of the abstract class. Possibly also the first mechanism may be put to use (by implementing additional interfaces). Some code is reused through this mechanism (the code in the abstract class), but most likely a lot of additional code has to be written. In the Swing

GUI [17] framework, included with Java, this type of hotspot is used to provide partial default implementations. If necessary, developers can choose not to use it and implement the interfaces instead.

3. *a component implementation of one or more roles in the role model.* There are two ways of putting components to use: inheritance (i.e. treat the component as a hotspot) and aggregation (i.e. treat the component as a frozen spot). We will argue in our guidelines that the latter approach is to be preferred over inheritance. Reusing components is the ultimate goal for a framework. Both design (components inherit this from the role models) and behavior (the components) are reused.

4. GUIDELINES FOR STRUCTURAL IMPROVEMENT

In this section we present a number of guidelines that aim to help developers deliver frameworks that are compliant with the conceptual model presented in the previous section.

4.1. The interface of a component should be separated from its implementation

Problems. Often there are a lot of implementation dependencies (direct references to implementation classes) between components. This makes it hard to replace components with a different implementation since all the places in the code where there is a reference to the component will need maintenance.

In addition to that, implementation dependencies are also more difficult to understand for developers since it is often unclear what particular function an implementation class has in a system. Especially if the classes are large or are located deep in the inheritance hierarchy, this is difficult.

Solution. Convert all implementation dependencies to interface dependencies. To do so the component API will have to be separated from the implementation. In Java this can be done by providing interfaces for a component. In C++, abstract classes in combination with virtual methods can be used. Instead of referring to the component class directly, references to the interface can be made.

Advantages. Components no longer rely on specific implementations of APIs but are able to use any implementation of an API. This means that components are less likely to be affected by implementation changes in other components. In addition, interfaces are more abstract than implementation classes. Using them allows programmers to program in a more abstract way and stimulates generalizations (which is good for both understandability and reusability).

Disadvantages. Often, there is only one implementation of an API (that is unlikely to change). The creation of a separate interface may appear to be somewhat redundant. Nevertheless the fact remains that many future requirements are unpredictable, so it is usually not very wise to assume this.

If languages without support for interfaces are used (such as C++), the mechanism to emulate the use of interfaces may involve a performance penalty (in C++ calls to virtual methods take more time to execute than regular method calls).

Example. This approach was chosen in the haemo dialysyis framework, where there is a distinct separation between the API (in the form of interfaces) and the implementation (in the form of application specific classes that implement the interfaces).

4.2. Interfaces should be role oriented

Problems. Often only a very specific part of the API of a component is needed. We refer to these little groups of related functionality as roles. Typically a component can act in more than one role (also see Section 3.2). A GUI button, for instance, can act in the role of a graphical entity on the screen. In that role it can draw itself and give information about its dimensions. Another role of the same component might be that it acts as the source of some sort of action event. Other roles that the component might support are that of a text container (the text on the button). Often roles can be related to design patterns [10]. In the observer pattern, for instance, there are two types of objects: observers and observables. Often the objects that fulfill these roles typically fulfill other roles as well.

If the interface that is needed to use a component in a certain role covers more than one role, unnecessary dependencies are created. If, for instance, the button component has an interface describing both the event source role and the graphical role, any component that needs to use a component in its event source role also becomes dependent on the graphical API. These dependencies will prevent the interface being reused in components that have the event source role but lack the graphical role.

Solution. To address this problem, interfaces should not cover more than one role. As a result, most components will implement more than one interface, thus making the notion that an object can act in more than one role more explicit.

Advantages. Small interfaces cause API changes to be more localized. Only components that interact with the component in the role in which the change occurred are affected (as opposed to all components interacting with that component in any role). Often the same role will appear in multiple components. By having a single interface for that role, all those components are interchangeable in each situation where only that role is required.

Disadvantages. Often more than one role is required of a component (i.e. a client is going to use a component in more than one role). We address this issue in Section 4.3. Having an interface for each role will cause the number of interfaces to grow. This growth will, however, be limited by the fact that the individual interfaces can be reused in more places. The total amount of LOC (lines of code) spent in interfaces may even decrease because there is less redundancy in the interface definitions. At the same time it will be easier to document what each interface does since each interface is small and has a clear goal.

Splitting a component's interface in multiple smaller interfaces causes the total number of interfaces to increase considerably (which can be confusing for developers). However, as Riehle and Gross argue [13], component collaborations are easier to understand when modeled using roles.

Example. In the haemo dialysis framework the interfaces are small (see Sections 2 and 3). This indicates that each of them provides an API that is specific to one role, as we suggest in this guideline. The PeriodicObject interface, for instance, provides only one method called tick(). Components implementing this interface will typically implement other interfaces as well. When such components are used in their PeriodicObject role, however, only the tick() method is relevant. So the only assumption a Scheduler object has to make about the components it schedules is that they provide this single tick() method (i.e. they implement the PeriodicObject interface).

4.3. Role inheritance should be used to combine different roles

Problems. If the guideline presented in Section 4.2 is followed, the number of interfaces each component implements generally grows considerably. Often when a component is used, more than one of its roles may be required by the client. This poses a problem in combination with the guideline in Section 4.1, which prescribes that only references to interfaces should be used in order to prevent implementation dependencies. This, however, is not possible: when a reference to a particular interface (representing a role) is used, all other interfaces are excluded.

There are several solutions to this problem.

1. Use a reference to the component's main class (supports all interfaces). However, this way implementation dependencies are created and it should therefore be avoided wherever possible.
2. Use typecasting to change the role of the component when needed. Unfortunately, typecasts are error prone because the compiler cannot check whether run-time type casts will succeed in all situations.
3. Merge the interfaces into one interface. This way the advantages of being able to refer to a component in a particular role are lost.

Neither of these solutions is very satisfying. They all violate our previous guidelines, resulting in a less flexible system.

Solution. What is needed is a mechanism where a component can still have role specific interfaces but can also be referred to in a more general way. An elegant way to achieve this is to use interface inheritance. By using interface inheritance new interfaces are created that inherit from other interfaces. By using interface inheritance, roles can be combined into a single interface (by using multiple inheritance). By using interface inheritance, a role hierarchy can be created. In this hierarchy, very specific role specific interfaces can be found at the top of the hierarchy while the inheriting interfaces are more general.

Advantages. All the previous guidelines are still respected. Yet it is possible to refer to multiple roles in a component by creating a new interface that inherits from more than one other interface. Interface inheritance gives developers the ability to use both fine-grained referencing (only a very small API) and coarse-grained referencing (a large API).

Disadvantages. The number of interfaces will increase some more, potentially adding to the problem mentioned in our previous guideline. Also the interface inheritance hierarchy may add some complexity. In particular, multiple inheritance of interfaces may make the hierarchy difficult to understand. Another problem may be that not all OO languages support interface inheritance (or even interfaces). C++, for instance, does not have interfaces (and thus no interface inheritance). It does, however, support abstract classes. Interfaces can be simulated by creating abstract classes without any implementation. Since C++ supports multiple inheritance, interface inheritance can also be simulated. Java, on the other hand, offers support for interfaces and interface inheritance.

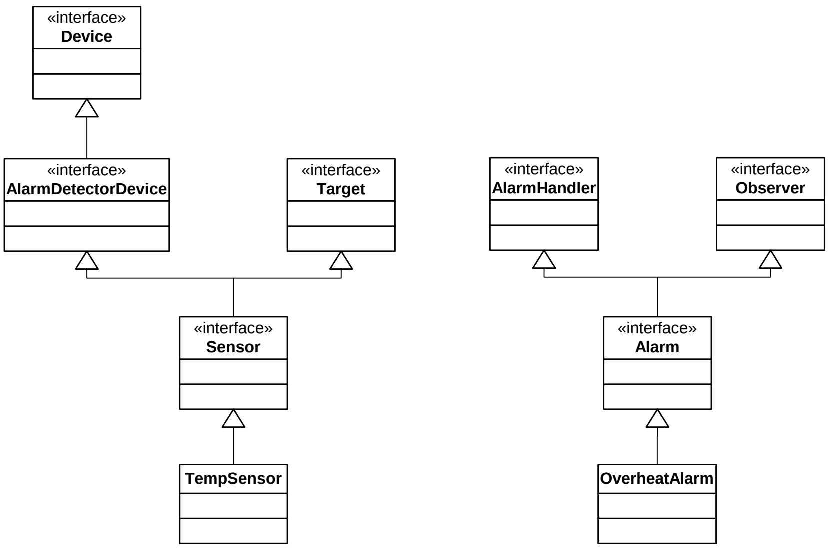

Figure 6. Example of interface inheritance.

Example. In [10] an example of a haemo dialysis application architecture based on the haemo dialysis framework is presented. Part of this architecture is an OverHeatAlarm component that responds to the output from a Tempsensor. In Figure 6, an example is given on how these two components could have been implemented. In this example, both the TempSensor and the OverHeatAlarm have one parent interface that inherits from other interfaces. The OverHeatAlarm implements the role of an observer (from the connector framework) and that of an AlarmHandler (from the core framework). The new Alarm interface makes it possible to refer to the component in both roles at the same time. Note that the scheduling framework is left out of this example. It is likely that both components also implement the PeriodicObject interface. It is unlikely, however, that any component referring to the components in that role would need to refer to those objects in another role.

4.4. Prefer loose coupling over delegation

Problems. In Section 3.3 we discussed several forms of obtaining a reference to a component in order to delegate method calls. We made a distinction between loose coupling (in the form of an event mechanism) and delegation and we also showed that some forms of delegation are more flexible than others. In order to be able to delegate methods to another component, a reference to that component is needed. With normal delegation (one of the four ways described in Section 3.3), a dependency is created between the delegating component and the component receiving the method call. These dependencies make the framework complex.

Solution. A solution to this increased complexity is to use loose coupling. When using loose coupling, components exchange messages through the events rather than calling methods on each other directly. The nice thing about events is that the event source is unaware of the target(s) of its events (hence the name loose coupling).

Advantages. By using loose coupling, developers can avoid creating direct dependencies between components. It also enables components to work together through a very small interface which further reduces the amount of dependencies between components. Furthermore, most RAD (rapid application development) tools support some form of loose coupling, thus making it easier to glue components together.

Disadvantages. Loose coupling can be slower than normal delegation. This may be a problem in a fine-grained system with many components. In these situations one of the other delegation forms, we discussed earlier, may be used.

Example. Through the connector mechanism in the haemo dialysis framework, the designers of that framework aimed to establish loose coupling. By introducing a third component, the Target is made independent of the observer (see Figure 3 while still allowing them to interact (through a notification mechanism). Through this mechanism, observer-implementing components can be connected to target-implementing components at run-time. This eliminates the need for observers to be aware of any other interface than the target interface.

4.5. Prefer delegation over inheritance

Problems. Complex inheritance hierarchies are difficult to understand for developers (empirical data that supports this claim can be found in [21]). Inheritance is used in object orientation to share behavior between classes. Subclasses can override methods in the super class and can extend the superclass' API with additional methods and properties. Another problem is that inheritance relations are fixed at compile time and can only be changed by editing source code.

Solution. Szyperski [12] argued that there are three aspects to inheritance: inheritance of interfaces, inheritance of implementation and substitutability (i.e. inheritance should denote an is-a relation between classes). We have provided an alternative for the first and the last aspect. Roles make it easy to inherit interfaces and since roles can be seen as types they also take care of substitutability. Consequently the main reason to use class inheritance is implementation inheritance.

When it comes to using inheritance for reuse of implementation there are the problems, we indicated previously, of increased complexity and less run-time flexibility. For this reason we believe it is better to use a more flexible delegation based approach in most cases.

Advantages. The main advantage of delegation is that delegation relations between objects can be changed at run-time. The flatter structure of the inheritance hierarchy, when using delegation, is easier to understand than an inheritance hierarchy. Components are more reusable than superclasses since they can be composed in arbitrary ways. An additional advantage for frameworks is that it allows for more of the intercomponent relations (both is-a and delegation relations) to be defined in the role model part layer of the framework. This allows for a better separation of structure and implementation.

Disadvantages. A straightforward migration from inheritance based frameworks to a delegation based framework may introduce method forwarding (calls to methods in super classes are converted to calls to other components). Method forwarding introduces redundant method calls, which affects maintainability negatively. Method forwarding is the result of straightforward refactoring inheritance relations into delegation relations. If delegation is used from the beginning this is not so much a problem.

Another problem is that an important mechanism for reusing behavior is lost. Traditionally, inheritance has been promoted for the ability to inherit behavior. Our experience with existing frameworks [2,5,10] has caused us to believe that inheritance may not be the most effective way of establishing implementation reuse in frameworks. Most frameworks we have encountered require that a considerable amount of code is written in order to use the framework. In those frameworks, inheritance is used more as a means to inherit APIs rather than behavior. Of course, abstract classes in the whitebox framework can still be used to generalize some behavior. A third problem may be that delegation is more expensive than inheritance in some languages (in terms of performance). Method inlining and other techniques that are applied during compilation or at run-time address this problem.

Finally, this approach may lead to some redundant code. This is especially true for large components (our next guideline argues that those should be avoided as well).

Example. The haemo dialysis framework does not use class inheritance very extensively. The whitebox framework, as discussed in [10], does not contain any classes (only interfaces). The example application architecture shown in the same architecture consists of several layers of components that are linked together by loose coupling and other delegation mechanisms.

4.6. Use small components

Problems. Large components can be used in a very limited number of ways. Often, it is not feasible to reuse only a part of such a component. Therefore, large components are only reusable in a very limited number of situations. It is difficult to create similar components without recreating part of the code that makes up the original. The problem is that large components behave like monolithic systems. It is difficult to decompose a large component into smaller entities. For the same reason, it is difficult to use the inheritance mechanism to refine component behavior.

Solution. The solution for this problem is to use small components. Small components only perform a limited set of functionality. This means that they have to be plugged together to do something useful. The small (atomic) components act as building bricks that can be used to construct larger (composed) components and applications (also see Section 3.2). In effect, large monolithic components are replaced by compositions of small, reusable components.

Advantages. Just like small whitebox frameworks, small components are easier to comprehend. This means that components can be developed by small groups of developers. The blackbox characteristics of the small components generally scale up without problems if they are used to build larger components. Individual small components are likely to offer more functionality than their counterparts in large components.

Disadvantages. Szyperski [12] argued that *maximizing reuse minimizes use*. With this statement he tried to illustrate the delicate balance between reusability (flexible, small components) and usability (large, easy to use components). While this is true, we have to keep in mind that the ultimate goal for a framework is increased flexibility and reusability. Therefore it is worthwhile considering shifting the reusability–usability balance towards reusability.

In addition, large components hide the complexity of how they work internally. The equivalent implemented in a network of small components is very complex, though. To make such a network of components accessible, some extra effort is needed. Luckily, only a few (or even just one) of the components in the network have to be visible from the outside in most cases. Externally the composite components are represented by one component while internally there may be a lot of components. In the example below, a temperature device uses several other components to do its job. Yet there is no need to access those components from the outside.

A real problem is the fact that the glue code tying together the small components is not reusable. To create new, similar networks of components, most of the gluecode will have to be written again. In large components, the glue code is part of the component. This does not mean that large components have an advantage here because large components lack the flexibility to change things radically. Solutions to the ill reusability of glue code can be found in automatic code generation. Automatic code generation is already used by many RAD (rapid application development) tools like IBM's VisualAge [22] or Borland's Delphi [16] to glue together medium- to large-grained components. Alternatively, scripting languages [23] can be used to create the networks of components.

Example. The strategy of using small components was also used in the haemo dialysis framework. In their paper [10], Bengtsson and Bosch describe an example application consisting of multiple layers of small components working together through the connector interfaces. In our example there is a TemperatureDevice which monitors and regulates the temperature of the dialysis fluids. To do so, it has two other components available: a TempSensor and a FluidHeater. The policy for when to activate the heater is delegated to a third component: the TempCtrl. Each of these components is very simple and reusable. The sensor is not concerned with either the heater or the control algorithm. Likewise, the control algorithm is not directly linked to either the sensor or the heater. In principle, upgrading either of these software components is trivial. This might, for instance, be necessary when a better temperature sensor comes available or when the control algorithm is updated. If this component would have been implemented as one large component, the code for TempSensor and the FluidHeater would not have been reusable. Also the controlling algorithm would be hard to reuse.

5. ADDITIONAL RECOMMENDATIONS

In addition to improving the structure of frameworks, we believe that there are several other issues that need to be addressed. The guidelines presented in this section should not be seen as the final solution for these issues. However, we do believe they are worth some attention when developing frameworks.

5.1. Use standard technology

Problems. The *not invented here syndrome* [24], that many companies suffer from, often causes 'reinvention of the wheel' situations. Often developers do not trust foreign technology or are simply unaware of the fact that there is a standard solution (standard in the sense that it is commonly used to solve the problem) for some of the problems they are trying to address. Instead, they develop a proprietary solution that is incorporated in the company's framework(s). In a later stage, this proprietary solution may become outdated, but by then it is difficult to move to standard technology because the existing software has become dependent on the proprietary solution.

Solution. When developing a framework, developers should be very careful to avoid reinventing the wheel. We recommend that developers use standard technology whenever possible unless there is a very good reason not to do it (price too high, missing functionality, performance too low or other quality attribute deficits). In such situations, the chosen solution should be implemented in such a way that it can easily be replaced later on.

An approach that is particularly successful at the moment is the use of standardized APIs. This allows for both standard implementations and custom implementations. Our approach to developing frameworks complements this nicely. Developers could standardize (or use standardized) versions of the interfaces in the role models of the framework.

Advantages. Standard technology has many advantages. It is widely used; so many developers are familiar with it. It is likely to be supported in the future (because it is used by many people). Since it is widely used, it is also widely tested. For the same reason, documentation is also widely available.

Assuming that the framework under development is going to be used for a long time, it is most likely to be counterproductive to use non-standard technology. It is important to realize that, in addition to the initial development cost, there is also the maintenance cost of the propietary solution that has to be taken into account when using non-standard technology.

Disadvantages. Standard technology may not provide the best possible solution. Another problem may be that generally no source code is available for propietary standard solutions. A third problem may be that the standard solution only partially fits the problem.

Also, standard technology should not be used as a silver bullet to solve complex problems. In their Lessons Learned paper [24], Schmidt and Fayad note that '. . . *the fear of failure often encourages companies to pin their hopes on silver bullets intended to slay the demons of distributed software complexity by using CASE tools or point and click wizards*. . . .' Despite this the use of standard technology still offers the advantage of forward compatibility (i.e. it is less likely to become obsolete), which may outweigh its current disadvantages.

Based on these disadvantages we identify the following legitimate reasons not to use standard technology.

1. There is an in-house solution which is better and gives the company a competitive edge over companies using the standard solution.
2. The company is aiming to set a standard rather than using an existing standard solution.
3. It is much cheaper to develop in house than to pay the license fees for a standard solution.

Example. In the haemo dialysis framework, a proprietary solution is introduced to link objects together (see the connector framework in Figure 3). This mechanism could get in the way if it were decided to move the architecture to a component model like Corba or DCOM, which typically use standard mechanisms to do this. Since the haemo dialysis framework apparently does not use a standard component model right now, a proprietary solution is necessary. In order to simplify the future adoption of these component models, the proprietary solution should make it easy to migrate to another solution later on, e.g. by making implementations of the connector framework on top of, say Corba, easy.

5.2. Automate configuration

Problems. If the guidelines presented so far are followed, the result will be a highly modularized, flexible, highly configurable framework. The process of configuring the framework will be a considerably more complex job than configuring a monolithic, inflexible framework. The reason for this is that part of the complexity of the whitebox framework has been moved downwards to the component level and the implementation level. Flexibility comes at the price of increased complexity.

Solution. Fortunately the gained flexibility allows for more sophisticated tools. Such tools may be code generators that generate gluecode to stick components together. They may be scripting tools that replace the gluecode by some scripting language (also see Roberts and Johnson's framework patterns [11].

Advantages. The use of configuration tools may reduce training cost and application development cost (assuming that the tools are easier to use than the framework). Also, configuration tools can provide an extra layer of abstraction. If the framework changes, the adapted tools may still be able to handle the old tool input.

Disadvantages. While tools may make life easier for application developers, they require an extra effort from framework developers for development and maintenance of these tools. Also, a tool may not take advantage of all the features provided by the framework. This is a common problem in, for instance, GUI frameworks where programmers often have to manually code things that are not supported by the GUI tools, thus often breaking compatibility with the tool.

Example. In the haemo dialysis framework, the connector framework could be used to create a tool to connect different components together. All the tool would need to do is create Link components (several different types of these components may be implemented) and set the target and observer objects.

5.3. Automate documentation

Problems. Documentation is very important in order to be able to understand and use a framework. Unfortunately, software development is often progressing faster than the documentation, leading to problems with both consistency and completeness of the documentation. In some situations, the source code is the only documentation. Methods for documenting frameworks are discussed in detail in Mattsson's licentiate thesis [4]. The problem with most documentation methods is that they require additional effort from the developers, who are usually reluctant to invest much time in documentation.

Solution. This problem can be addressed by generating part of the documentation automatically. Though this is not a solution for all documentation problems, it at least addresses the fact that source code often is the only documentation. Automatic documentation generation can be integrated with the building process of the framework.

Automated documentation is also important because, as a consequence of the guidelines in Section 3, the structure of frameworks may become more complex. Having a tool that helps in making a framework more accessible is therefore very important.

Advantages. If the tools are available, documentation can be created effortlessly, possibly as a part of the build process for the software. Another advantage of automating documentation is that it is much easier to keep the documentation up to date. A third advantage is that it stimulates developers to keep the documentation up to date.

Disadvantages. There are not so many tools available that automatically document frameworks. If documentation is a problem it might be worthwhile to consider building a proprietary tool. Higher level documentation such as diagrams and code examples still have to be created and evolved manually. Additional documentation (e.g. design documents and user manuals) is needed and cannot be replaced by automatically generated documentation. Most existing tools only help in extracting API documentation and reverse engineering source code to UML diagrams. Both types of tools usually do not work fully automatically (i.e. some effort from developers is needed to create useful documentation with them). In addition, the documentation process needs to have attention from the management.

Example. A popular tool for generating API documentation is JavaDoc [25]. JavaDoc is a simple tool that comes with the JDK. It analyzes source code and generates HTML documents. Developers can add comments to their source code to give extra information, but even without those comments the resulting HTML code is useful. The widespread acceptation and use of this tool clearly shows that simple tools such as JavaDoc can greatly improve documentation.

6. RELATED WORK

Roberts and Johnson's framework patterns [11] inspired several elements of the framework model we presented in Section 3. For instance, the notion of whitebox and blackbox frameworks also appears in their paper. Furthermore, they discuss the notion of fine-grained objects where we use the term atomic components. Finally, they stress the virtue of language tools as a means to configure a framework (guideline 5.2). The idea of language tools and other configuration aides is also promoted in Schappert *et al.* [26].

Also related is the work of Johnson and Foote [27]. Their plea for *standardized, shared protocols* for objects can be seen as a motivation for the central set of roles in our conceptual model. However, they do not make explicit that one object can support more than one role (or protocol in their terminology). In addition they argue in their guidelines for programmers that *large classes should be viewed with suspicion and held to be guilty of poor design until proven innocent*, which is in support of our guideline 4.6. Interestingly, they also argue that inheritance hierarchies should be deep and narrow, something which has been proved very bad for complexity and understandability in empirical research [21]. However, in combination with their ideas about standard protocols, it provides some arguments for our idea of role inheritance (guideline 4.3).

Our idea of role models somewhat matches the idea of framework axes as presented in Demeyer *et al.* [28]. The three guidelines presented in that paper aim to increase interoperability, distribution and extensibility of frameworks. To achieve this, the authors separate the implementation of the individual axes as much as possible, similar to our guidelines 4.1, 4.2, 4.4 and 4.6. Pree and Koskimies [9] introduce the idea of a framelet: a small framework (*small is beautiful*). Again this matches our idea of role models, but our notion of components extends their model substantially.

In Parsons *et al.* [20], a different model of frameworks is introduced. They introduce a model where basic components are hooked into a backbone (resembles an ORB—object request broker). In addition to these basic components there are also additional components. The main contribution of this model seems to be that it stresses the importance of an ORB (i.e. loose coupling of components) in a framework architecture. However, contrary to our view of a framework, it also centralizes all the components around the backbone (giving it whitebox framework characteristics), something we try to prevent by having multiple, independent role models.

The significance of roles (guidelines 4.2 and 4.3) in framework design was also argued in Riehle and Gross [13]. In this article, the authors introduce roles and role models as a means to model object collaborations more effectively than is possible with normal class diagrams. In their view frameworks can be defined in terms of classes, roles that can be assigned to those classes and roles that need to be implemented by framework clients. In Reenskaug's book [14], the OORam software engineering method is introduced which uses the concept of roles. A similar methodology, Catalysis, is discussed by D'Souza and Wills [29]. In Bosch's paper [30], roles are used as part of architectural fragments.

Guidelines 4.4 and guideline 4.5 are inspired by Lieberherr and Holland's law of Demeter [19], which aims at minimizing the number of dependencies of a method on other objects. The two guidelines we present aim to make the dependencies between components more flexible by converting inheritance relations into delegation and delegation relations into loose coupling.

7. CONCLUSION

In this article we presented a conceptual model for frameworks. The model includes definitions of terms such as class and component. In addition it promotes a role oriented approach to framework development. Based on this model we provide a set of guidelines and recommendations. The aim of our guidelines is threefold:

(i) increased flexibility;
(ii) increased reusability;
(iii) increased usability.

The guidelines are mostly quite practical and range from advice on how to modularize the framework to a method for documenting a framework. Key elements in the development philosophy reflected in our guidelines is that *small is beautiful* (applies to both components and interfaces), hardwired relations are bad for flexibility and ease of use is important for successful framework deployment. Of course our guidelines are not universally applicable since there are some disadvantages for each guideline that may cause it to break down in particular situations. However, we believe that they hold true in general.

7.1. Future work

Essentially our solution for achieving flexibility results in a large number of small components that are glued together dynamically. By having small framelets/role models, a lot of the static complexity of existing frameworks is transformed in a more dynamic complexity of relations between components. These complex relations bring about new maintenance problems since this complexity no longer resides in frameworks but in framework instances. Large components are not a solution because they lack flexibility, i.e. they can only be used in a fixed way. So, a different solution will have to be found. One solution may be found in scripting languages like JavaScript or Perl, as discussed in Ousterhout's article on scripting [23]. Scripting languages are mostly typeless, which makes them suitable to glue together components. That typing can get in the way when gluing together components was also observed in Pree and Koskimies' work [9], but there reflection is used as an alternative.

A second issue that we intend to address is how to deal with existing architectures. Existing architectures most likely do not match our framework model. It would be interesting to examine whether our guidelines could be used to transform such architectures into a form that matches our model. It would also be interesting to verify if such transformed architectures do deliver on the promises of reuse and easy application creation as mentioned in our introduction.

Thirdly we aim to widen the scope of our research from frameworks to so-called software product lines [31]. We will examine whether our conceptual model for frameworks is applicable to software product lines and whether this model can be refined further.

ACKNOWLEDGEMENTS

We would like to thank Michael Mattsson for proofreading this article and for providing useful suggestions. Also we would like to thank the reviewers of this journal for their valuable input. In addition we would like to thank Per Olof Bengtsson for allowing us to use his illustration of the haemo dialysis machine (Figure 1).

REFERENCES

1. Bosch J, Molin P, Mattson M, Bengtsson P. Object oriented frameworks—problems and experiences. *Building Application Frameworks*, Fayad ME, Schmidt DC, Johnson RE (eds.). Wiley & Sons, 1999.
2. Bosch J. Design of an object-oriented framework for measurement systems. *Object-Oriented Application Frameworks*, Fayad ME, Schmidt DC, Johnson RE (eds.). Wiley & Sons, 1999.
3. Mattsson M, Bosch J. Framework composition problems, causes and solutions. *Proceedings Technology of Object-Oriented Languages and Systems*, U.S.A., August 1997.
4. Mattsson M. Object-oriented frameworks survey of methodological issues. *Licentiate Thesis*, Department of Computer Science, Lund University, 1996.
5. Mattsson M, Bosch J. Evolution observations of an industrial object oriented framework. *International Conference on Software Maintenance (ICSM) '99*, Oxford, England, 1999.
6. Nemirovsky AM. Building object-oriented frameworks. http://www.ibm.com/software/developer/library/oobuilding/index.html. Last verified November 2000.
7. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns—Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.
8. Sparks S, Benner K, Faris C. Managing object oriented framework reuse. *Computer* 1996; **29**(9):53–61.
9. Pree W, Koskimies K. Rearchitecting legacy systems—concepts and case study. *First Working IFIP Conference on Software Architecture (WICSA'99)*, San Antonio, TX, February 1999; 51–61.
10. Bengtsson P, Bosch J. Haemo dialysis software architecture design experiences. *Proceedings of the 21st International Conference on Software Engineering*, May 1999.
11. Roberts D, Johnson R. Patterns for evolving frameworks. *Pattern Languages of Program Design* 1998; **3**:471–486.
12. Szyperski C. *Component Software—Beyond Object Oriented Programming*. Addison-Wesley, 1997.
13. Riehle D, Gross T. Role model based framework design and integration. *Proceedings of OOPSLA '98*. ACM Press, 1998; 117–133.
14. Reenskaug T. *Working With Objects*. Manning Publications Co., 1996.
15. Pree W. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley: Reading, MA, 1994.
16. Inprise (previously Borland). http://www.inprise.com/. Last verified November 2000.
17. Javasoft. http://www.javasoft.com/. Last verified November 2000.
18. McCabe TJ. A complexity measure. *IEEE Transactions on Software Engineering* 1976; **2**:308–320.

19. Lieberherr KJ, Holland IM. Assuring good style for object oriented programs. *IEEE Software* 1989; **6**(5):38–48.
20. Parsons D, Rashid A, Speck A, Telea A. A framework for object oriented frameworks design. *Proceedings of TOOLS '99*. IEEE Computer Society, 1999; 141–151.
21. Daly J, Brooks A, Miller J, Roper M, Wood M. The effect of inheritance on the maintainability of object oriented software: An empirical study. *Proceedings of International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos, CA, U.S.A., 1995; 20–29.
22. *VisualAge*, IBM. http://www.software.ibm.com/.
23. Ousterhout JK. Scripting: Higher level programming for the 21st century. *IEEE Computer Magazine* 1998; **31**(3):23–30.
24. Schmidt DC, Fayad ME. Lessons learned—building reusable OO frameworks for distributed software. *Communications of the ACM* 1997; **40**(10):85–87.
25. JavaSoft. *JavaDoc homepage*. http://www.java.sun.com/j2se/javadoc/index.html. Last verified November 2000.
26. Schappert A, Sommerlad P, Pree W. Automated support for software development with frameworks. *Proceedings of the 17th International Conference on Software Engineering*, 1995; 123–127.
27. Johnson RE, Foote B. Designing reusable classes. *Journal of Object Oriented Programming* 1988; **1**(2):22–35.
28. Demeyer S, Meijler TD, Nierstrasz O, Steyaert P. Design guidelines for tailorable frameworks. *Communications of the ACM* 1997; **40**(10):60–64.
29. D'Souza D, Wills AC. Composing modelling frameworks in catalysis. *Building Application Frameworks—Object Oriented Foundations of Framework Design*, ch. 19, Fayad ME, Schmidt DC, Johnson RE (eds.). John Wiley & Sons, 1999.
30. Bosch J. Specifying frameworks and design patterns as architectural fragments. *Proceedings Technology of Object-Oriented Languages and Systems ASIA'98*, July 1998; 268–277.
31. Bosch J. *Design and Use of Software Architectures—Adopting and Evolving a Product Line Approach*. Addison-Wesley, 2000.