Wilde: A dynamic, post compilation co-ordination platform for groupware applications.

Stephen Barrett♦ and Brendan Tangney
Distributed Systems Group
Trinity College Dublin

Abstract

A co-ordination language technology, called Wilde, is presented which supports the construction of component-based systems open to dynamic redesign. Wilde provides a co-ordination language that supports the design of software as compositions of reusable, generic, simple design patterns. Wilde provides a means whereby designs can be transformed, or modified, in a safe and tractable fashion. A distinguishing feature of Wilde is that it allows for **post-compilation** modification of components and their interactions.

Wilde has been implemented over DCOM and CORBA as a runtime library and associated co-ordination language compiler allowing Wilde based applications to be constructed from standard CORBA and DCOM components. A number of sample Wilde applications are described as vehicles for demonstrating the strength of the paradigm. The paper concludes that the Wilde paradigm, based on redesign of otherwise inflexible systems, offers great potential for implementing flexible (groupware) software systems.

1 Introduction

Providing flexibility in software is about constructing systems that can adapt to changing requirements. New software domains such as Groupware, or computer supported co-operative work, are particularly challenging domains for software design because it is very difficult to determine the exact requirements of these types of applications [1][7][10][18]. For example [6][15], a groupware user may spontaneously decide that he needs to integrate a spreadsheet application into a conference system! Because the requirements of groupware systems are so fluid, cannot be reliably predicted in advance, and can even change dynamically from execution to execution, current general-purpose solutions are failing to adequately support user's needs. In response to this, researchers are engaged in the development of new techniques for creating more modifiable, customisable and adaptive software in order to design for flexibility.

Many contemporary approaches to providing such flexibility are based upon generalising designs so that they cater for as many scenarios as possible. For example common desktop applications, such as Microsoft Word, provide extensive features to cater for a wide range of uses with the typical user only ever availing of a subset of all the features. In terms of software design methodologies, **frameworks** seek to provide a general

♦ email: Stephen.Barrett@tcd.ie

solution that can later be specialised to meet the needs of a particular instantiation.

Other more radical solutions have been suggested which support the selective re-implementation of software. Examples include polymorphic-based component solutions, which enable the replacement of functional code, and reflective techniques, which allow for the extension of implementations, typically for non-functional aspects such as security or distribution and so forth. These approaches are promising although the software's basic design must still be broadly appropriate to the scenario.

The contemporary approaches mentioned have proved useful because they provide a relatively inexpensive means of generalising features in a system's design, so enabling the software to be tailored to some degree when necessary. The cost of this is to increase the complexity of software thus making it ever more difficult to build and use. It is generally excepted that Frameworks, for example, take time to mature and must go through a number of iterations before they are considered useful [8]. With each iteration, the addition of new requirements invalidates some previous design decisions, necessitating the *redesign* of the framework. As the complexity of the software increases, it becomes more difficult to adapt the system to new requirements. The accumulative effect of each new design feature added is to make the design more rigid generally, increasing the likelihood that additional new requirements might not be supportable without *substantial* redesign. Thus, contemporary approaches suffer from a paradox: *sophisticated design limits flexibility.* This occurs notwithstanding design pattern techniques aimed at introducing flexibility. In volatile and complex software domains such as groupware, the conflict between maintaining flexibility while increasing complexity will demonstrate the inadequacy of contemporary techniques.

An alternative approach to generalising the design of software is to build simple software that satisfies the singular needs of the problem at hand, while at the same time allowing that software to be easily, and dynamically redesigned to support new requirements when necessary. This approach necessitates support for dynamic redesign. However, in mainstream software technologies design is a pre-implementation activity. Redesign necessitates re-implementation of code, and is therefore very expensive. New approaches are needed which make the redesign of software a workable proposition. A useful approach will support the modelling of designs in a manner such that redesign is both understandable and inexpensive to do. For example, it would be advantageous to support the selective redesign of systems post-compile, and indeed dynamically while executing, over an approach that requires re-compilation of code modules.

Four existing research areas have important bearing on a solution. First, component technologies such as DCOM and CORBA provide the designers with the tools to type, package and dynamically bind system sub-components in a heterogeneous distributed environment. Second, co-ordination language research suggests ways in which system designs can be specified independently of a system's functional code [16]. Third, reflection research provides a framework for exposing certain features and properties of a system in order to dynamically inspect and even modify them independently of the system as a whole. Fourth, research in patterns indicates that the design task would benefit from the ideas of reuse commonly associated with object-oriented programming

languages [8][20].

The work presented here derives from research in these four areas. A new component oriented approach called *Wilde* is presented, that supports the construction of component-based applications that can be dynamically redesigned without recompilation[1]. Wilde provides a co-ordination language which supports the description of component-based software designs as object oriented patterns that specify how an application's components should interact. It supports the specification of an application's design separately and independently from the application's functional components, which can be implemented in any language for which a CORBA or DCOM binding exists. At runtime, Wilde supports the dynamic inspection and redesign of applications. The co-ordination language provides a rich type-model for specifying design patterns and for reasoning about redesign in a type safe way. In the remainder of this paper, Wilde's implementation is discussed alongside early results from its application to flexibility problems. These early results, while not conclusive, nevertheless indicate great potential in the Wilde approach for constructing very flexible software systems.

2 Wilde

Wilde supports the construction of applications that can be redesigned, even while executing. The Wilde approach is based on the observation that complex and powerful software can be constructed from components that encapsulate functionality behind interfaces. Object-oriented and object-based component technologies based on this, among other principles, work well. It is practical therefore to capture a system's design as a model of the communication links between a set of typed components. In keeping with other systems such as Darwin [14] and Olan [2], a co-ordination language is used to specify a system design as a self-contained abstraction, rather than dispersing it throughout functional code, as is the case with standard implementation methodologies.

However, unlike such co-ordination language approaches, a Wilde application design can be modified after the system has been compiled and even dynamically while the system is executing. A Wilde application design is compiled separately from functional components and the compilation of such a design has no transformational effect on component code. At runtime, the compiled design is implemented by a set of objects instantiated from a support library, and is thus made available for dynamic manipulation. In reflective terms, the design is reified.

To execute a Wilde application, a set of appropriate components is plugged into the runtime representation of the design, which is then responsible for managing the connectivity of the components. Thus, a Wilde application takes the form of a set of independent components managed by a runtime representation of the application design. The benefit of this approach is to preserve the tractability of the co-ordination language approach beyond compile time so that its advantages can be applied to the problem of

[1] What Nierstratsz[16] refers to as 'open systems'.

making component software more flexible. In the following section, the Wilde co-ordination language is explained in more detail with the aid of a simple example derived from the groupware domain – a shared text editor. Following this, in section 5 we examine a more useful and comprehensive example.

3 The Wilde Co-ordination Language

Wilde's co-ordination language allows generic design patterns to be specified, and for these to be specialised and composed to implement more complex designs. The language is built on top of the interface definition languages of CORBA and DCOM. It provides a single new construct, called a 'wildecard' similar in form to an interface or class. This construct has a number of uses. First, it is used to type functional components in terms of the interfaces (ie. typed sets of operations) the component provides, and the interfaces the component will require while executing. Thus, the wildecard is used to completely specify the possible interactions between the components and all other parts of a system. Second, the wildecard is used to specify design patterns as *composite* components, involving instances of one or more component types, co-ordinated according to a set of interaction rules. Third, the wildecard may be used to declare a set of specific components to be used in instantiating a design pattern.

3.1 The Wildecard construct

Wildecards can be specialised through inheritance, and can be used as components within other, more complex wildecards. Thus, Wilde enables complex application designs to be built up from more primitive design patterns, and provides an elegant means of specifying individual applications via specialisation of and aggregation of generic design patterns.

The wildecard construct is similar in form to the class construct of C++, or Java. The wildecard syntax is illustrated in figure 1. The 'TextEditor' wildecard describes how two components (a graphical user interface and a data store) are co-ordinated. There are a number of optional sections to a generic wildecard and these are described below.

```
wildecard TextEditor{
        required:   // the interfaces needed
        provided:  // the interfaces provided
        components:
           DataStore data;   // a DataStore component called data
           FrontEnd gui;     // a FrontEnd component called gui
        mappings:
           gui.ITextEdit > data;          // data provides the ITextEdit interface to gui
           data.IDataCallback > gui;     // gui provides callback interface to data
        instantiation:                    // not all components specified
           data = MyDataComponent;    //data is instantiated from another wildecard
};
```

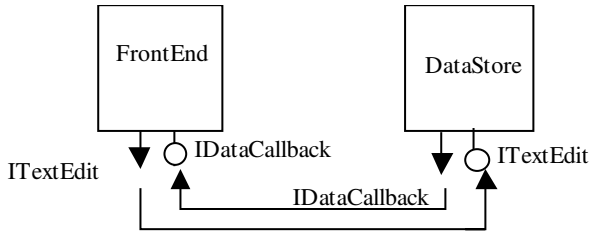

Figure 1 – A simple text editor wildecard

Components

All composite wildecards must include a component section. Each constituent component of the 'TextEditor' is listed in the 'components' section. A wildecard type identifies each component listed. For example, in reference to figure 1, TextEditor contains a component data of type DataStore. A wildecard description of each component must be available to the Wilde runtime. Such wildecards may be nested, or may be defined separately from the TextEditor wildecard. Note that the data component does not indicate any particular implementation, but is simply a name. The instantiation section, discussed below, can be used to bind particular implementations to component names.

Required and provided

All wildecards that are used as sub-components within another wildecard must include a 'required' and/or 'provided' section. Required and provided sections list the set of IDL interfaces the component calls and thus requires of other components, and the set of interfaces that this component can provide to other components. Wilde adopts a multiple interface per component model similar to the object model of DCOM or Java. The required and provided sections of the TextEditor wildecard of figure 1 indicate that it cannot be used as a component within another wildecard since no interfaces are listed there. In the section 3.2 we describe the specialisation of the TextEditor wildecard to

include an external interface.

Mappings

All composite components must include a 'mappings' section, in which the co-ordination of sub-components is described. Each statement in that section of the wildecard specifies how some interface requirement of one component is provided by some other component. Each interface name in these statements is scoped in order to indicate which component is being referred to. Non-scoped interface names refer to the required and provided interfaces of the containing wildecard itself. However, most mapping statements are of the form *component1.interface > component2*, meaning component2 provides

```
wildecard multicom{
  required: ITarget; // multiple sources ok
  provided: single IEvent;   // only one client allowed
};

wildecard MyApp{
  components:
     multicom mc;
          …
   mappings:
     mc.ITarget > tg;
     mc.ITarget > tg2; // legal because multiple mappings allowed
     mc.IEvent < tg1;
     mc.IEvent < tg2; // illegal – mapping already exists
};
```

Figure 3 - legal and illegal mappings

component1 with the interface. Some mapping statements can be of the form *component1 <> component2*, which is shorthand for all possible mappings between the components. Wilde will create a mapping between each provided interface required by the other.

A particular mapping statement does not preclude another mapping statement involving a required interface declared in the first statement. However, if the declaration of the interface is prefixed with the 'single' keyword in the required section of the relevant component wildecard, additional mapping statements are illegal. The same is true of provided interfaces. This is illustrated in figure 3.

Instantiation

The 'instantiation' section is used to declare bindings between sub-component types within a wildecard, and real component classes that can be instantiated to play the role of sub-components within the wildecard. If the instantiation section is not present in a wildecard or if does not specify a real component class for every wildecard sub-component, then that wildecard cannot be instantiated. However, it may be used as a base

implementation for further specialisation (see section 3.2).

The syntax of the instantiation section is illustrated in figure 1. Each statement takes the form of an assignment, *componentType = [qualifiers,…] componentClass*. 'ComponentType' refers to a declared wildecard sub-component type. The component class is an identifier for a real functional component that can fulfil the necessary role. The component class may be prefixed with qualifiers such as the machine to instantiate the component on and so forth.

3.2 Specialisation of wildecards

In order to be able to reuse and tailor existing patterns where appropriate, wildecards can be specialised. Figure 4 illustrates a specialisation of the TextEditor wildecard. This specialisation achieves an important result – it opens up the TextEditor composition, providing it with an external interface, so that now it may be used as a component within a larger system. The new mapping statements are in addition to those inherited from TextEditor.

There is more to specialisation of wildecards than presented here. In general, when specialising a wildecard one may add to any section. In addition, one may modify the type of any existing sub-component if modifying the type does not break an existing mapping statement. Specialisations can be qualified. For example, the 'single' keyword can be used to alter any interface of a specialised wildecard, if this modification does not invalidate existing mapping statements. Supporting inheritance brings all the benefits of resuse, as seen in object-oriented languages, to design specification. Specialisation is a useful tool for describing relationships between designs. In the case of TextEditorComponent, inheritance allows us to describe a design that is clearly a TextEditor, but which extends that design. It is this sort of typed relationship between designs that provides a type safe model for changing designs.

```
wildecard TextEditorComponent: TextEditor{
        required:   ITextEditor;
        provided:  ITextEditor;
        mappings:
                gui.ITextEdit > this;//gui required ITextEdit is part of external interface
                this.ITextEdit > data;//external provided ITextEdit is provided by data
        instantiation: TextEditor.gui = mygui;
};
```

Figure 4 – specialisation of TextEditor

The wildcard construct allows an application design to be described as a nested set of wildcards to whatever level of granularity is useful. Figure 5 below outlines a wildcard specification for a very simple shared text editor (The drawing to the right illustrates this system. The system consists of two text editor components. Those components are declared as being of type TextEditorComponent, which was described above as being a specialisation of TextEditor, such that both provide and require the ITextEdit interface).

Wilde supports the dynamic application of wildecards like SharedTextEditor to existing systems. Designs can be instantiated from executing components and component compositions. If necessary, the design of an executing application such as the TextEditor can be modified so that it can be used as a component within the new system. Thus, even at runtime, two applications adhering to the TextEditor wildecard can be integrated by typed transformations (from TextEditor to TextEditor component) to form a multi-user shared text editor. The components used to implement the single-user text editor are used without modification to implement a shared text editor system. All that has changed is the manner in which they interact with each other.

Given a wildecard specification of the shared text editor system, the programmer must source or implement the functional components. A Wilde component is simply a component that adheres to a wildecard specification (e.g. the `TextEditor` wildecard of figure 1).

4 Wilde Components

However, Wilde imposes one new programming rule on component programmers: references to other components are not allowed, since all inter-component co-ordination is managed by Wilde. Thus, Wilde supports the ActiveX connection point programming model and so ordinary ActiveX components can be integrated into a Wilde application. The programming rule can be broken but the use of explicit references limits flexibility [21].

The rule simplifies the task of component construction since component programmers are no longer required to initialise and maintain references. Wilde provides support for constructing new components so that they adhere to the rule. For example, in the current implementation of Wilde, a typical C++ invocation via a class pointer, such as `pITextEdit->NewChar(a)`, is replaced with a call to the Wilde runtime which takes a Wilde interface name and returns an appropriate pointer. The Wilde call equivalent to the C++ invocation above would `be CallRequired(ITextEdit)->NewChar(a)`. This comes at the cost of an indirection. We are exploring reflective techniques for improving transparency for the programmer and removing this cost.

```
// A standard IDL interface shared
// by both text editors.
interface ITextEdit{
        void NewChar([in] unsigned char c);
};

// The specification of the Shared
// Text Editor system
wildecard SharedTextEditor{


    required: none;    //  This Shared Text Editor system neither requires
    provided: none;    //  nor provides and interface functionality
    components:
        TextEditorComponent editor1;// Only two components in this system
        TextEditorComponent editor2;
    mappings:
        editor1.ITextEdit > editor2.ITextEdit; // Mapping statements describe
        editor2.ITextEdit > editor1.ITextEdit; // the component interactions
}
```

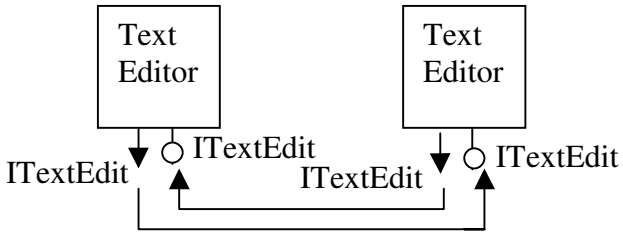

Figure 5 – A shared text editor

5 Constructing Wilde Applications

As the shared text editor system of figure 5 is an extremely simple application with very simple interactions on the part of the two components, it could easily be constructed around an event model for example. However, applications such as text editors, conferencing tools are complex, highly structured systems, and mainstream procedural and object-oriented paradigms are more appropriate tools for modelling their complexity. Wilde aims to introduce flexibility into far more complex systems. The Wilde model is quite different from contemporary broker models and event models. In this section, in order to provide examples that more clearly demonstrates the benefits of the Wilde approach, we expand the shared text editor of section 3.2 into a more comprehensive and complex system.

Shared text editors, as a class of groupware based on the single-user text-editor concept, seem on first examination to be a simple evolution of the latter application type.

However, the complexity of shared text editors becomes apparent if one considers the numerous ways in which single user text editor concepts (such as menus, scrollbars, and the immediate display of typed text) can be handled in the multi-user scenario. For example, if one user scrolls his view of the text, should all other user views be scrolled to synchronise the display of text? Should the selection of a menu option affect all user interfaces? Should a vote be performed on the selection of a menu option to allow a group decision be made? In fact, the possible permutations and variations on the basic shared text editor concept are in effect limitless. It is because of this variability that much groupware, which tends not to accommodate such an extensive range of configurations, fails to support user's needs.

Consider a co-operative editing scenario between two users who wish to use two different single-user applications for their collaboration. One application, being used by a journalist, is a simple text tool. The other application is a sophisticated graphics system being used by a layout artist. The journalist will be concerned principally with content, the layout artist with layout. The collaboration will centre on rewriting the text in order to fit the article into available page space. The collaboration takes a number of different forms. There will be periods when the two participants will collaborate in real time in order to discuss the merits of this change or that change. There will also be periods of asynchronous activity, when for example the journalist will take some time alone to rewrite a paragraph before returning to continue the collaboration.

The scenario is highly dynamic and complex and there are many aspects to a solution. There are two basic existing approaches to solving the problem. First, we might rewrite the two applications as a single groupware tool. That is, we might produce a new groupware application that provides different interfaces and functionality to different users and that encompasses the functionality of both tools. Second, we might design a standard for application interaction and rewrite the two applications to adhere to that standard. The first solution is simply impractical. A groupware system cannot be extensively rewritten every time we wish to integrate a new application into the collaboration (even if source code of the various applications is available to do so). This is why we seek to support application integration rather than burden the application programmers with this problem. The second solution is limited because applications are too diverse to hope that a single standard can be imposed that does not severely limit a particular class of application, or that is simple enough to be generally applied. Without a universal groupware standard that is simple (and thus that economically justifies the standard inclusion of groupware functionality), application designers will not provide groupware functionality in their systems.

Integrating two applications is a difficult task, but Wilde does provide the flexibility to construct a solution. By constructing the application as a Wilde system from reasonably fine-grained components, the original application designers can ignore groupware and yet provide us with the flexibility we need to modify the way the applications work. The applications can then be redesigned later in a targeted manner, by replacing or augmenting application components with pre-defined groupware components where possible or with new bespoke components where necessary. In this way, one ends up with two flavours of each application: the basic application and a groupware *variant* that can

be used as part of a groupware system. The groupware variant is just that: the majority of the applications structure and functionality will remain unaffected.

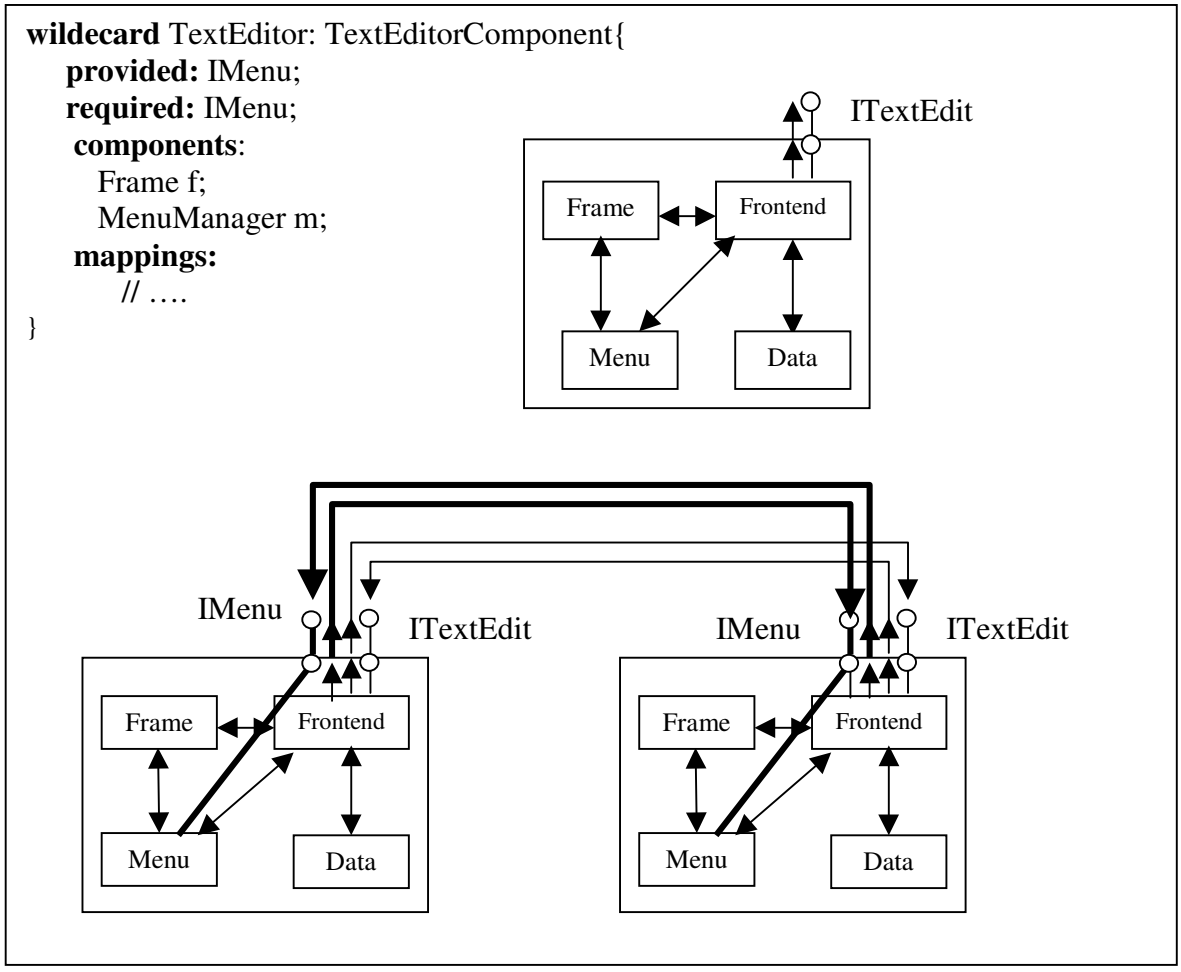

Figure 6 – A decomposed text editor used in multi-user shared text editor system

To illustrate the approach we will discuss some specific problems in a typical integration. The first is that of integrating the menu systems of two separate applications, a specific example of general problem of integrating user interfaces. In a shared text-editor system that enforces strict WYSIWIS [2], if one user selects a menu all other users should see that menu selected. In a more sophisticated groupware system, the selection of a menu option should perhaps require a vote on the part of the participants. For example, the layout artist should not be allowed cut the selected text unless the journalist agrees. Whatever level of sophistication required, the core solution is nonetheless the same: transform or replace the single user application's menu system so that it becomes a part of a co-operative multi-user menu.

---

[2] What You See Is What I See – meaning that every change in the user interface of one user of a groupware application should be reflected in the user interfaces of every other user. If one user scrolls the text of a shared text editor, every other interface should display the text scrolling in the same manner.

Figure 6 illustrates a single-user text-editor, which is reasonably decomposed. We have further specialised the TextEditorComponent of figure 4 to include menu system and window frame components, but for clarity, we have omitted the detail. In order to integrate the menu systems of two identical applications such as that described in figure 5, we need only expose interactions between the MenuManager and Interface sub-components externally as illustrated in figure 6. To do this we would specialise the SharedTextEditor wildecard as indicated.

Clearly, the co-ordination patterns discussed so far are quite simple. We mentioned a more complex scenario in which voting might be required on a menu selection such as, for example, the cutting of text by the layout artist in our motivating example. Existing functionality within the two single-user applications would not support this and so reconfiguration of the two applications will not achieve a solution by itself. However, because the two systems may be assumed to be decomposed, as in figure 6, a very simple solution is possible through a combination of reconfiguration and addition of new groupware components. Figure 7 illustrates the addition of a menu filter component to each application.

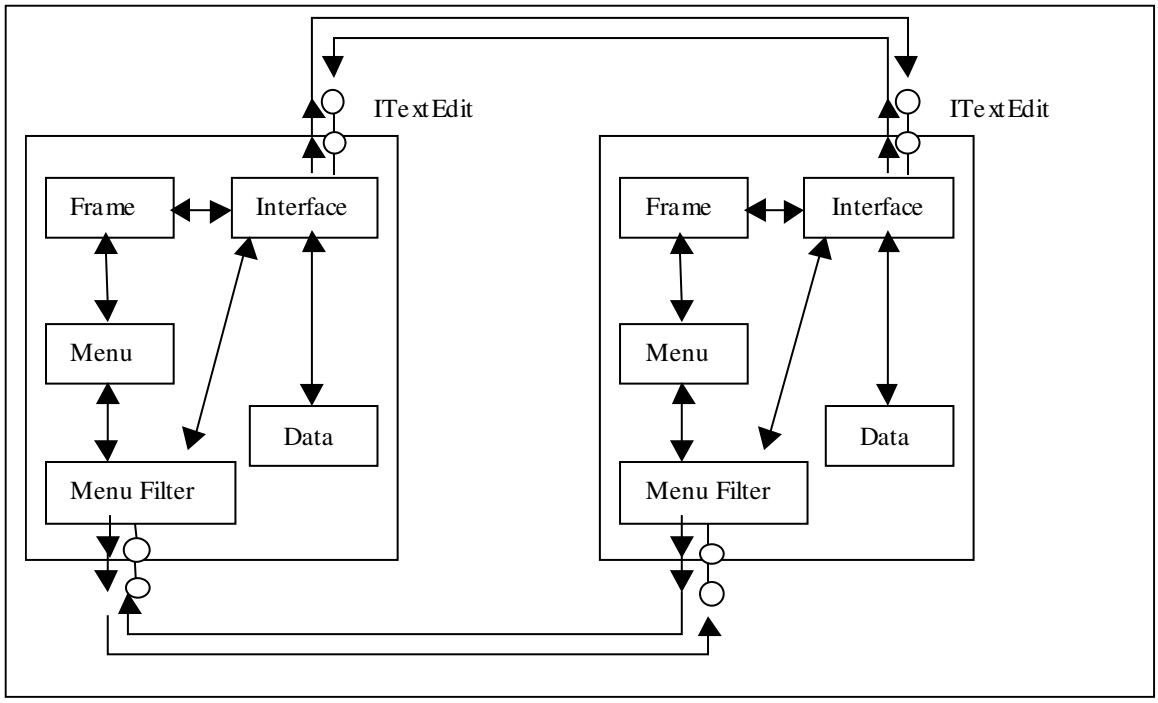

Figure 7 – menu integration

Menu integration is a relatively simple example but demonstrates the principle advantage of Wilde. The range of transformations we can perform on a Wilde application are constrained only by the degree to which the application is decomposed. Integration of functionality to include voting or any other form of user co-ordination can be achieved in a similar manner. The greater the decomposition, the more open the application is to modification. Indeed, because it supports bottom-up construction of components from more fundamental sub-components, Wilde encourages the fragmentation of sub-systems. This improves the likelihood that whatever the needs of the redeveloper, he will find a

solution through targeted redevelopment of the sub-systems in the application.

As a final example, consider the issue of data management. In the scenarios we have illustrated so far, each text-editor instance maintains its own data, managed by a data management component. An alternative design, more appropriate in some circumstances, is for each text-editor instance to use a single shared data management component. This is illustrated in figure 8.

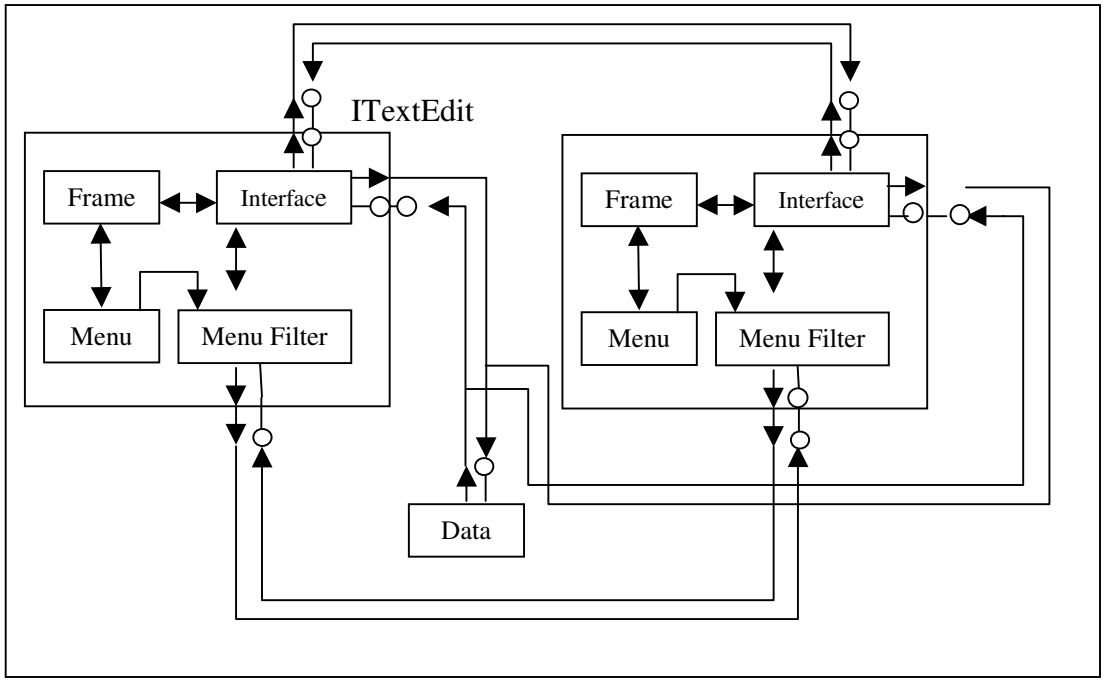

Figure 8- data sharing

In the examples presented above we have sought to demonstrate that flexibility through redesign is a useful software paradigm. We do not claim, however, that Wilde is a panacea for the construction of groupware, or any other demanding software domain. For example, Wilde says nothing about how shared text editors should be built, about how interfaces should be integrated or data shared. On the contrary, the goal of Wilde is to provide the designer tools provide the *unconstrained* flexibility necessary to address fundamental groupware problems as he or she sees fit. We hold that libraries, services and applications built using Wilde, in being open to redesign, will prove to be more effective and flexible groupware solutions.

6 Current implementation and further work

Wilde has been implemented over DCOM and CORBA for the Windows NT platform as a runtime library, allowing Wilde based applications to be constructed from standard CORBA and DCOM components. The library has been written in C++, and has been designed with portability to other platforms and component technologies in mind. This library includes functionality for type checking designs and can produce a compiled intermediate type checked form of a design for faster instantiation. At runtime, a Wilde

application design is loaded and instantiated from a set of library components. These then initialise and manage the interactions of the application's components. The library's overhead is low, as interactions between functional components are direct, and do not pass through a mediating layer. The role of the instantiated design (implemented by library components) is simply to configure component interconnections, and reconfigure these if redesign is performed dynamically. There is therefore a start-up and redesign overhead.

We have implemented a set of shared text editor based systems, including those discussed in this paper, constructed from a small set of reusable components and design patterns. These applications are simple but serve to demonstrate the potential of the Wilde paradigm. We are actively developing more sophisticated systems such as "office procedure systems", and are exploring the specification of common design patterns in our co-ordination language. We intend ultimately, to provide a library of useful reusable components and designs for the construction of groupware in particular.

We are currently examining the extension of the Wilde co-ordination language to provide an enriched type model for describing component's behaviour. We see our model, which supports the specification required and provided interfaces as a first step. We intend to next examine how to extend our notion of component interface to describe black-box behaviour.

Initial work has begun on a graphical development environment for the design, instantiation, and modification of Wilde applications. We are considering supporting UML as a graphical notation for the description of application designs.

7 Related Work

We are aware of research in the field of software composition that is broadly similar to our approach [16] [12] [9]. However, our approach focuses on open systems construction based on component oriented, interface-based distributed technologies such as DCOM and CORBA. We are interested in achieving flexibility without resorting to radically new object models, programming languages or paradigms. Transparency is our goal. Our approach supports the post compile-time redesign of applications constructed using binary components written in any language supporting supported by the supported distributed technologies. Although research systems, such as Darwin [14] and Olan [2], advocate a similar approach, they do not preserve a system's design post compile-time, and so designs are fixed at compile-time.

Many Linda oriented systems have been suggested in which components communicate anonymously via a tuple space [4]. Event based and message passing models offer similar decoupling, albeit in a synchronous rather than asynchronous paradigm [13][11]. However, as Olan, Darwin, and indeed our research shows, the necessary decoupling is achievable without resorting to such programming models: procedural and object-oriented paradigms can be supported. This offers an advantage when constructing complex systems for which these paradigms are the most natural.

Our approach differs radically from those that provide or tailor the underlying platform [3] [5], or toolkit based approaches [19]. We hold that flexibility is best supported through support for system redesign.

8 Summary

Providing flexibility in software is about constructing systems that can adapt to changing requirements. New software domains such as Groupware are particularly challenging domains for software design because it is very difficult to determine the exact requirements of such applications. In seeking to provide some flexibility, contemporary approaches tend to increase the complexity of software thus making it ever more difficult to build and use. An alternative approach to generalising the design of software is to build simple software that satisfies the singular needs of the problem at hand, while at the same time allowing that software to be easily, and dynamically redesigned to support new requirements when necessary. This approach necessitates support for dynamic redesign.

Wilde was presented as a technology that supports the construction of component-based systems open to dynamic redesign. Wilde provides a co-ordination language that supports the design of software as compositions of reusable, generic, simple design patterns. A distinguishing feature of Wilde is that it allows for **post-compilation** modification of components and their interactions in a safe and tractable manner.

Wilde has been implemented over DCOM and CORBA, allowing Wilde based applications to be constructed from standard CORBA and DCOM components. In the context of shared text editors, we explained the strength of the Wilde paradigm in terms of the flexibility it affords. We hold that the Wilde paradigm, based on redesign of otherwise inflexible systems, offers great potential for implementing flexible (groupware) software systems.

9 References

[1] Stephen Barrett and Brendan Tangney, "Aspects - Composing CSCW Applications," in *OOIS (Object-Oriented Information Systems) '95*, Springer-Verlag, Dec 1995.

[2] L. Bellissard, S. Ben Atallah, A. Kerbrat and M. Riveill, "Component-based programming and aplication management with Olan," in *Object based parallel and distributed computation, Frane-Japan worksho, OBPDC '95,* J. Briot, J. Geib and A. Yonezaru (eds.), LNCS 1107, Springer Verlag, 1995.

[3] Benford, S. and Mariani, J., eds, *Requirements and metaphors of shared interaction.* Lancaster University, Esprit Basic Research Report project 6225, D4.1., 1993.

[4] R. Bjornson, N. Carriero, D. Gelernter, "From weaving threads to untangling the web: A view of coordination from Linds's perspective," in *Proceedings of the second international conference on Coordination languages and Models, COORDINATION '97,* Berlin Germany, Sep 1997, LNCS 1282,, Springer Verlag, 1997.

[5] C. Bidan and V. Issarny, "Security benefits from software architecture" in *Proceedings of the second international conference on Coordination languages and Models, COORDINATION '97,* Berlin Germany, Sep 1997, LNCS 1282, Springer Verlag, 1997.

[6] Cutkosky, M. R., Engelmore, R. S., Fikes, R. E., Genesereth, M. R., Gruber, T. R., Mark, W. S., Tenenbaum, J. M., And Weber, J. C. "PACT: An experiment in integrating concurrent engineering systems," IEEE Computer Magazine (Jan. 1993), 28-37.

[7] Paul Dourish "Open implementation and flexibility in CSCW toolkits", PhD thesis, University College London, 1996.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[9] Gelernter, D. and Carriero, N., "Coordination Languages and Their Significance," *Comm. ACM,* Vol. 35, No. 2, Feb. 1992, pp 97-107.

[10] Greenberg, S., "Personalisable Groupware: Accommodating Individual Roles and Group Differences," in Proc. European Conference on Computer-Supported Co-operative Work CSCW'94, Amsterdam, Netherlands, pp. 17-32.

[11] N. Haly, H. Minsky and V. Vingureanu, "Regulated coordination in open distributed systems," in *Proceedings of the second international conference on Coordination languages and Models, COORDINATION '97,* Berlin Germany, Sep 1997, LNCS 1282, Springer Verlag, 1997.

[12] Helm, R., Holland, I.M. and Gangopadhyay, D., "Contracts: Specifying behavioural compositions in object-oriented," in *Conf. On Object-Oriented Programming: Systems, Languages and Applications. European Conference on Object-Oriented Programming,* pp. 169-180, ECOOP/OOPSLA, ACM Press, 1990.

[13] A. Holzbacher, M. Perin and M. Sudholt, "Modelling railway control systems using graph grammars: a case study," in *Proceedings of the second international conference on Coordination languages and Models, COORDINATION '97,* Berlin Germany, Sep 1997, LNCS 1282, Springer Verlag, 1997.

[14] J. Kramer and J. Magee, "Darwin – Exposing the skeleton in the coordination closet," in *Proceedings of the second international conference on Coordination languages and Models, COORDINATION '97,* Berlin Germany, Sep 1997, LNCS 1282, Springer Verlag, 1997.

[15] Liang, T., Lai, H., and Chen, N., "When Client/Server isn't enough: Co-ordinating Multiple Distributed Tasks", IEEE Computer, vol. 27, no. 5, May 1994.

[16] Oscar Nierstrasz, "Requirements for a composition language," in *Proceedings of the ECOOP 94 workshop on Models and Languages for Coordination and Parallelism and Distribution,* LNCS, Springer Verlag, 1995.

[17] O'Grady, T., Flexible Data Sharing in a Groupware Toolkit. M.Sc. thesis, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada. November 1996.

[18] W. Reinhard, J. Schweitzer and G. Volksen, "CSCW Tools: Concepts and Architectures", IEEE Computer, vol. 27, no. 5, May 1994.

[19] Roseman, M. and Greenberg, S, "Building flexible groupware through open protocols," in *Conf. On Office Information Systems*, pp. 279-288, ACM, 1993.

[20] D.C. Schmidt, "Acceptor and Connector: Design Patterns for Initialising Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Addison-Wesley, 1997.

[21] Kevin J. Sullivan and David Notkin, "Behavioural relationships in object oriented analysis." Technical Report 92-03-08, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195 USA, February 27 1992.