CONSTRUCTING DISTRIBUTED GROUPWARE SYSTEMS

A WALK ON THE WILDE[1] SIDE

Stephen Barrett, Brendan Tangney, Vinny Cahill
Distributed Systems Group
Department of Computer Science
Trinity College Dublin

E-mail: Stephen.Barrett@tcd.ie

INTRODUCTION

The term "groupware" is used to describe a wide range of application types, from e-mail systems through shared text editors to videoconferencing systems, all of which aim to support the co-operation of multiple users [Ellis et al., 1991]. Building such systems is difficult. All but the most trivial groupware system are complex distributed multi-user systems. For example, a system might integrate a spreadsheet, a CAD package and a multi-media conferencing application, enforcing a variation of strict turn taking on the part of the participants [Cutkosky et al., 1993]. It is important therefore to find a way to *support* the construction of groupware, so simplifying the design and programming tasks. This support should be generic and comprehensive: it should support, in every important respect, the construction of a very broad spectrum of groupware systems, rather than some subset. We hypothesise that useful groupware support should be based on an *integrated* approach that focuses first on the commonality across the range of groupware systems, rather than targeting specific support issues for one or other class of system.

In [Barrett and Tangney, 1995] we identified a set of five *fundamental CSCW features* that characterise most groupware applications and that should be supported in order to meet the goals outlined. Groupware is very commonly **distributed** and **concurrent**. Furthermore, application parts typically interact **implicitly**, e.g. using a *publish-subscribe* paradigm. It is often desirable to **integrate applications**. For example, a typical conferencing system might integrate a shared whiteboard application with another application providing voice conferencing functionality. Groupware does not operate in isolation: it must be possible to co-ordinate the activities of groupware applications [Liang et al., 1994], and possibly in ways that the application creators might not have anticipated. Furthermore, to be truly useful such integration must be achieved post-compile time, despite the likely necessity for substantial functional change within both applications.

Finally, groupware must **adapt dynamically** in response to a changing environment or changing requirements. Groupware, being distributed multi-user software, must be designed to cope with variable underlying platform quality. For example, [Dourish, 1996] and [O'Grady, 1996] describe how groupware must support flexible data sharing models in order to cope with problems such as variable network bandwidth. Furthermore, as [Reinhard et al., 1994] and [Greenberg, 1991] discuss, the user level functionality a groupware tool provides must be flexible in order to accommodate different working patterns of collaborators and the requirements of the particular collaborative scenario. All this research suggests that effective groupware systems are those that can be customised extensively by end users, even beyond the imagination or ability of the system designer.

Taken together, these five features (distribution, concurrency, implicit communication, post-compile time integration and dynamic adaptation) characterise most groupware. They are fundamental in that they are the building blocks on which more complex groupware functionality is based. Groupware is

[1] In honour of Oscar Wilde - Irish wit, poet, and dramatist and Trinity student.

hard, we surmise, because the features identified indicate that useful groupware applications must be very flexible.

WILDE

Software must be designed to be flexible – flexibility is a designed feature of any system. However, design is not performed in a vacuum and it is the technology used to implement the system that frames the possibilities open to the designer. For example, a component technology, such as DCOM or CORBA, enables the design of a far more open system than would a standard C++ compiler since the former allows for the replacement of functional components without recompilation. When we speak of flexibility, we implicitly mean supporting change, and thus the value of a flexible technology is measurable by assessing the degree to which it supports and encourages the design of systems that can accommodate change. There are two distinct types of change relevant here. Change in a system can be simple replacement of functional components (ie. polymorphism). In this, existing component technologies excel. However, change can also be more extensive, involving change to the overall design of a system, by which we mean changes to the number and types of components and in particular the way these components interact. Current component technologies have nothing to offer the designer faced with this type of problem and therefore do not offer sufficient support for the construction of flexible systems.

Accommodating extensive change to a design requires support for the redefinition of the composition and structure of that system. In terms of object-orientation, this means supporting the introduction of new implementations of existing types, and indeed new types, and more importantly, the redefinition of interactions between instances. Clearly with vanilla object-oriented techniques, this is not possible without modification of code: in current object-oriented technologies, including component technologies, the specifications of inter-object interactions are embedded *implicitly* in the functional code of the implemented system. What is implicit is expensive to modify and therefore a limiting factor on openness [Sullivan and Notkin, 1992]. This is a widely accepted viewpoint in the object-oriented community, as can be seen from the wide range of design patterns that propose the explicit representation of inter-component relationships in order that change might be facilitated [Gamma et al., 1995] [Schmidt, D.C., 1997]. We conclude from this that a truly flexible technology must support the *explicit* representation of an application's design separately from its functional constituents: supporting flexibility necessitates a modified approach to constructing software. Wilde provides just such an approach.

The insight behind our work is that the flexibility required can be achieved by constructing groupware as *open systems,* meaning loose open-ended and re-configurable compositions of autonomous components [Nierstrasz, 1995]. A component in this context is an application sub-system. By initially structuring an application as a collection of autonomous sub-components, it can then later be adapted to meet new requirements by restructuring the way these components interact, or by adding, removing or replacing components. Flexibility is achievable by being open. Accordingly, we have developed **Wilde**; a new *open systems* model characterised by those features that we have identified as being fundamental to groupware. Wilde provides an object model well suited to the construction of open, flexible groupware systems.

Wilde's key feature is its support for the separation of the *functionality* of components from their *co-ordination*. For this, Wilde provides a construct, called a **wildecard,** for describing the types and co-ordination of components. There are two classes of wildecard, *composite* and *non-composite*. A non-composite wildecard describes a component in terms of the services it provides and the services it requires. That is, it describes all possible interaction between the component and the external world. Services are described in terms of standard IDL interface types. A composite wildecard also describes a component in terms of provided and required services, but in addition describes the component as an aggregate of sub-component types. It specifies how these sub-components interact with each other, and how they together provide and require the services specified in the composite component description. Thus, the wildcard construct allows an application design to be described as a nested set of wildcards to whatever level of granularity is useful.

As an example, figure 1 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 that both provide and require and interface (called `ITextEdit`) that simply propagates key presses.

```
// 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{

          // A nested description of a text editor single
     // user interface – components must match this
          wildecard TextEditor{
                    required: ITextEdit;
                    provided: ITextEdit;
     };

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

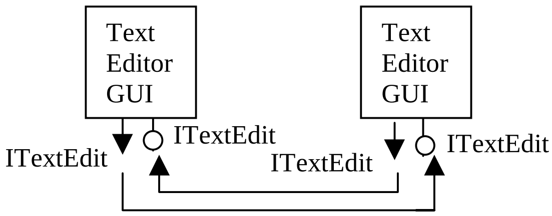

Figure 1 - wildcard specification of a simple shared text editor

Given a wildecard specification of the shared text editor system, the programmer must then 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).

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[2]. This simplifies the task of component construction since component programmers are no longer required to initialise and maintain references. 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)`.

At runtime, a form of service broker executes the wildecard specification for the shared text editor system. In *reflective* terms, Wilde *reifies* the application design, which means that the broker provides enables us to examine and modify the design specification at runtime. Instantiation is simply a matter of instantiating and plugging functional components into this runtime broker. There is no program mainline, instantiation is in a sense declarative, rather than imperative. It is the role of the broker to insure that the instantiated components are co-ordinated according to the wildcard specification. In the

[2] This rule can be broken but the use of explicit references, as explained, limits flexibility. Furthermore, Wilde supports existing ActiveX connection point programming techniques and so ordinary ActiveX components can be integrated into a Wilde application. The CallRequired mechanism is there to simplify component programming for those more accustomed to ordinary pointer techniques.

case of figure 1, its role is simply to insure that all `ITextEditor:NewChar(...)`calls made by either TextEditor component are relayed to its partner.

As the shared text editor system of figure 1 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. However, non-trivial applications have far more structure and an event model would not be appropriate in those cases. Wilde aims to introduce flexibility into far more complex systems. The Wilde model is quite different from contemporary broker models and event models. The Wilde broker acts according to a specific application design that specifies who provides services to whom. The broker implementing the design, rather than the components, is in control. In the following section, we take a more detailed look at more complex shared text editor systems.

YOU USE YOURS AND I'LL USE MINE

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.

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. If applications are initially constructed as open systems, then groupware concepts can be added later. 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. With Wilde, applications can be redesigned 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.

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 [3], if one user select 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.

Figure 2 illustrates a single-user text-editor, which is reasonably decomposed. We have omitted detail for clarity. Though we need not have, we have included external interfaces. This allows us to use this simple single-user text editor in a groupware system, such as that of figure 1. Thus, by a simple reconfiguration of component co-ordination, rudimentary groupware capability (ie. the ability to communicate simple events) can be added to the single-user text-editor. No re-coding of the components is necessary. In fact, strict WYSIWIS type synchronisation of any user actions, and indeed many other variants of this, can be achieved by exposing otherwise internal interactions externally and specifying how the different applications should interact along these new channels. For example, in order to integrate the menu systems of two identical applications such as that described in figure 2, we need only expose interactions between the MenuManager and Interface sub-components externally.

wildecard TextEditor{
    required: ITextEdit; // *The external interfaces*
    provided: ITextEdit;

    components:
      Frame f;
      Interface i;
      MenuManager m;
      DataManager d;
    mappings:
        // ….
}

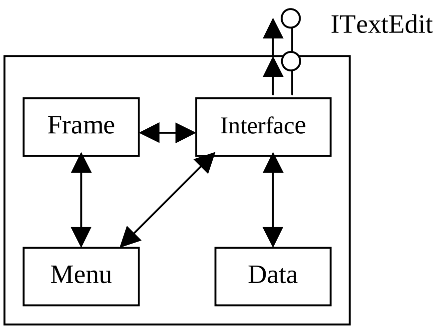

Figure 2 - a decomposed Text editor sytem

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 2, a very simple solution is possible through a combination of reconfiguration and addition of new groupware components. Figure 3 illustrates the addition of a menu filter component to each application.

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

[3] 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.

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.

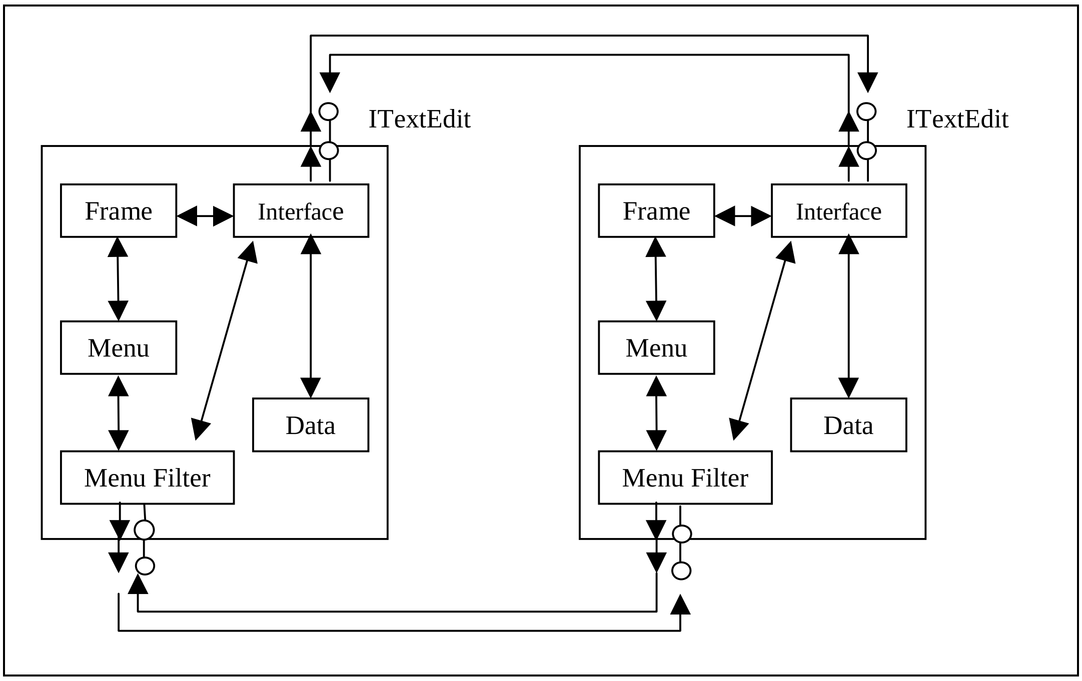

Figure 3 - voting

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 that is more appropriate in some circumstances is for each text-editor instance to make use of a single shared data management component. This is illustrated in figure 4.

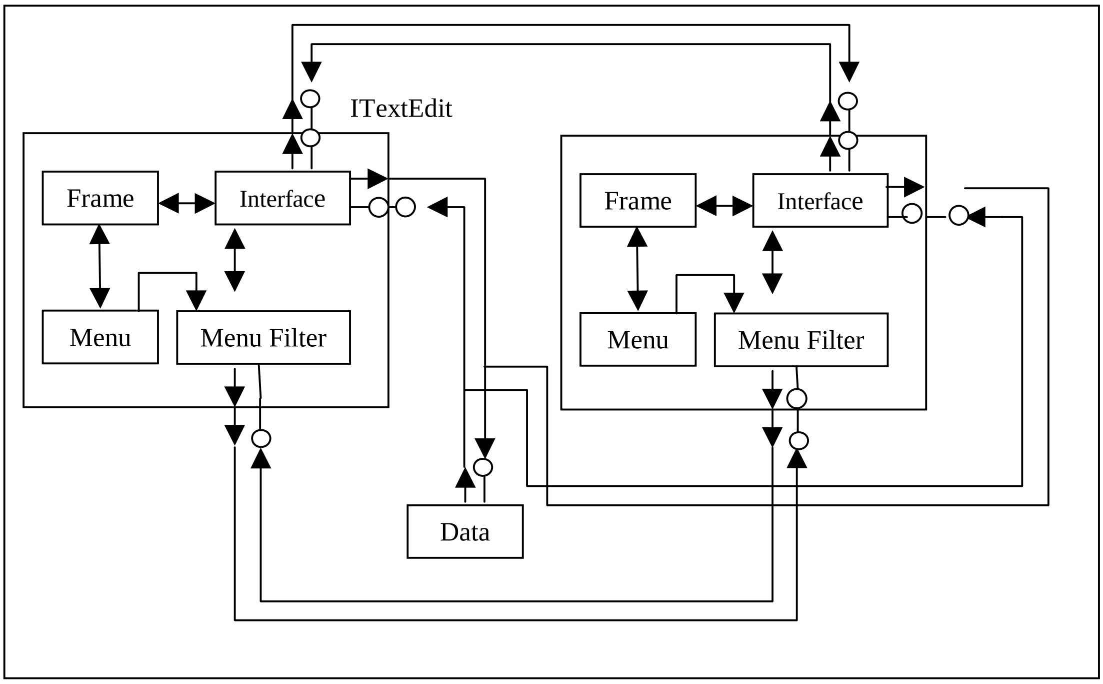

Figure 4- data sharing

We do not claim that Wilde is a panacea for the construction of groupware. 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 with a well-suited underlying object model which provides 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 provide more effective and flexible groupware solutions.

IMPLEMENTATION

Wilde's implementation aims to provide an effective groupware support solution through integration with and enhancement of contemporary component models such as DCOM and CORBA. Wilde's implementation includes a library (currently for CORBA and DCOM on Windows NT) which builds on component technologies taking advantage of their support for constructing distributed binary components. Wilde's component model is roughly equivalent to DCOM in that each component is assumed to support one or more interfaces. Indeed, ordinary ActiveX components may be used as components in DCOM based Wilde applications. The Wilde library provides a generic broker component that can be loaded with a wildecard specification. We are currently exploring graphical rather than textual notations for describing and manipulating nested wildcards. Wilde includes runtime functionality to load a generic broker component with a wildcard specification and to instantiate and plug in a set of specified functional components. Such a broker instance may be used as a component in yet another broker instance. Thus, a Wilde application is implemented as a hierarchy of broker instances managing the interactions of functional components. In this way, Wilde allows applications to be modified dynamically post-compile, and even post-deployment by altering the broker composition and/or replacing or adding components.

Wilde has been implemented to avoid costly indirection in invocations between functional components: once relationships are established by Wilde between components, they interact directly. A local cache of pointers to required interfaces is maintained by Wilde with each component in its own address space. There is an initial overhead of populating the cache of a component plugged into a broker. Following this, runtime overhead for an external invocation is of the order of validated pointer dereferencing or array indexing.

We have to date, implemented a number of Wilde applications such as the range of shared text editor systems described in this paper. We are now exploring the construction of more complex systems with Wilde.

RELATED WORK

We are aware of research in the field of software which is broadly similar to our approach composition [Nierstrasz, 1995] [Helm et al., 1990] [Gelernter and Carriero, 1992]. 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 the flexibility of open systems without resorting to radically new object models or programming languages. Our approach is post compile time and largely independent of the language used to construct components. Most other research focuses on composition as a language construct that is lost at compile-time, and can therefore be regarded as a variant of framework and pattern research.

Our application of open systems to groupware differs radically from platform based approaches, such as [Benford and Mariani, 1993], or toolkit based approaches, such as that of [Roseman and Greenberg, 1993]. We hold that groupware is best supported through an appropriate underlying object model that accommodates the variability of groupware.

CONCLUSIONS

Wilde does not aim to solve the high level problems of groupware, such as what model of data sharing to use in an application or how best to multi-cast user interface events. The goal of Wilde is to provide a better platform from which to address these issues. Wilde provides a very flexible application model based on the features we have identified as being fundamental to groupware. Wilde's model of an application as an open-ended, modifiable construction provides the flexibility necessary to support groupware applications that can *adapt dynamically*, and be *integrated* with other applications. Furthermore, groupware construction is facilitated by the model's inherent support for *concurrency* and *distribution* and *implicit communication* models. By providing a flexible groupware oriented application model we believe that Wilde offers a better base from which to address the challenges of groupware.

REFERENCES

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

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

[Cutkosky et al., 1993] 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.

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

[Ellis et al., 1991] C.A. Ellis, S.J. Gibbs, and R.L. Rein. "Groupware, some issues and experiences," *Communications of the ACM*, 34(1), January 1991.

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

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

[Greenberg, 1991] 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.

[Helm et al., 1990] 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.

[Liang et al., 1994] 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.

[Nierstrasz, 1995] 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.

[O'Grady, 1996] 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.

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

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

[Schmidt, D.C., 1997] 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.

[Sullivan and Notkin, 1992] 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.