

Building Flexible, Distributed Collaboration Tools using Type-Based Publish/Subscribe — The Distributed Knight Case

Klaus Marius Hansen and Christian Heide Damm
Computer Science Department
University of Aarhus
Aabogade 34, DK-8200 Aarhus N
email: {marius,damm}@daimi.au.dk

ABSTRACT

Distributed collaboration is becoming increasingly important also in software development. Combined with an increasing interest in experimental and agile approaches to software development, this poses challenges to tool support for software development. Specifically, tool support is needed for flexible, distributed collaboration. We introduce the *Distributed Knight* tool that provides flexible and lightweight support for distributed collaboration in object-oriented modelling. The Distributed Knight implementation builds crucially on the *type-based publish/subscribe* distributed communication paradigm, which provides an effective and natural abstraction for developing distributed collaboration tools.

KEY WORDS

Tool support for object-oriented modelling, distributed collaboration, publish/subscribe

1 Introduction

Distributed collaboration tools support multiple users in cooperating, coordinating, and communicating distributed in space and possibly also in time. As more and more people are working together while being geographically separate [3], such tools are going to become increasingly important. The domain of software development environments is an example of an application domain where distributed collaboration capabilities are becoming more important, due to the increasing globalisation also of software development.

Moreover, a number of recent system development approaches have been developed to handle system development situations characterized by high uncertainty and shifting requirements. This includes various “agile” software development approaches (<http://www.agilealliance.org>) such as Extreme Programming (XP; [2]), and the Crystal Family [4].

In combination, this poses the challenge of providing lightweight, flexible, and distributed collaboration support. The main contribution of this paper is the introduction of a tool, *Distributed Knight* that attempts to provide such support for the case of object-oriented modelling. Furthermore, Distributed Knight is implemented

using the *type-based publish/subscribe* distributed communication abstraction, and we propose that this abstraction provides a suitable foundation for building such interactive, distributed collaboration tools.

1.1 Paper Structure

The rest of this paper is structured as follows: Section 2 presents type-based publish/subscribe. Based on this, Section 3 introduces Distributed Knight and Section 4 discusses its implementation. Section 5 discusses status and related work, and finally Section 6 summarizes.

2 Type-Based Publish/Subscribe

Publish/subscribe is an event-based, distributed communication style in which *publishers* publish events, and *subscribers* subscribe to and receive the events they are interested in.

The traditional publish/subscribe systems are based on the *subject-based* (or *topic-based*) variant [14]. In subject-based publish/subscribe, events are published to particular named subjects, and subscribers can then subscribe to subjects and will receive all events that are published to those subjects. The *content-based* publish/subscribe variant [5] is a more dynamic variant, in which a subscriber can specify the runtime properties that events should have. Only the events that satisfy these properties will be delivered to the subscriber. *Type-based* publish/subscribe [7] is a recent *object-oriented* variant of this style. In type-based publish/subscribe, events are *objects*, i.e., instances of native types in an object-oriented programming language. This potentially leads to higher type safety and better encapsulation of events than traditional publish/subscribe communication styles and enables the programmer to focus on object-oriented abstractions and modelling in contrast to lower level details of, e.g., object serialization.

In type-based publish/subscribe, the subscriber of a particular type of objects will only receive instances of that type and its subtypes. Subscriber-specified *content filters* further limit the events that will be delivered to the subscriber. Content filters are specified in the native language

based on the events' public attributes and methods, and they may be processed remotely to reduce network load and processing load at subscribers.

2.1 Implementation of Type-Based Publish/Subscribe

Type-based publish/subscribe has originally been implemented for Java as a library and subsequently as an extension to Java [7]. The reflection mechanisms of Java alone are not sufficient to implement libraries for type-based publish/subscribe; a clean integration into the Java language requires language extensions [7].

Our current implementation of type-based publish/subscribe uses an object-oriented version of Tcl/Tk [13] since a large part of the Knight tool has been implemented in this language. Due to the interpreted and reflective nature of Tcl/Tk, we can easily extend the language with primitives to support type-based publish/subscribe without changing the language. In principle, there should be no difference between using our Tcl/Tk-based implementation and a Java-based implementation.

Figure 1 shows an example of an event type (using familiar Java syntax). This *MouseEvent* is an awareness event, and it contains the information that a given user has, e.g., moved or clicked the mouse in a given place. It may be used in building distributed collaboration tools by, e.g., replicating users' cursors at remote sites. Also shown in Figure 1 are examples of how clients publish and subscribe to *MouseEvent*s.

Our current implementation uses a central server that dispatches events from publishers to subscribers interested in these events.

3 The Knight Tools

This section presents the Knight tool and introduces how it has been extended with distributed collaboration capabilities using type-based publish/subscribe.

3.1 Stand-Alone Knight

The Knight tool was conceived of as a tool to support co-located, collaborative, lightweight and flexible modelling [6] using the Unified Modeling Language (UML; [12]). Based on observations of real-world modelling practice, we designed and implemented a tool that supports the kind of modelling work that is usually performed on traditional whiteboards.

Figure 2 shows the Knight tool in use on an *electronic whiteboard*. An electronic whiteboard has a large, pressure-sensitive surface that displays a computer screen and on which users may draw using pens. Alternatively, Knight may be used with more traditional input devices such as mice or track balls. To support an interaction much like that on an ordinary whiteboard, Knight uses *gestures*

```
// Define a new event type extending
// SessionAwarenessEvent
public class MouseEvent
    extends SessionAwarenessEvent {
    private int actionType;
    public int getActionType() {return actionType;}
    ...

    public MouseEvent
        (int userID, ..., int actionType, ...) {
        super(userID, ...);
        this.actionType = actionType;
        ...
    }
}

// Publishing
MouseEvent event = new MouseEvent
(userID, ..., ButtonPress, ...);
publish event;

// Subscribing
Subscription s = subscribe (MouseEvent event) {
    // Content filter - return true iff we
    // want to handle this event
    if (event.getSenderID()
        != PublishSubscribe.clientID) {
        // only handle mouse actions performed
        // in the same session as us
        return (event.getSessionID() == sessionID);
    } else {
        // ignore our own mouse actions
        return false;
    }
}
// Event handler: Indicate the mouse action
// in the user interface
...
}
s.activate();
```

Figure 1. Publishing and Subscribing to Events

to create, delete, and modify most elements [6]. Figure 2 shows an example of creating a class using gestures.

Furthermore, the Knight tool supports flexibility in modelling by allowing for incomplete UML elements and freehand drawing to be mixed with formal UML models [6].

The Knight tool has been commercialized by Ideogramic ApS as *Ideogramic UML* (<http://www.ideogramic.com/products/uml/>).

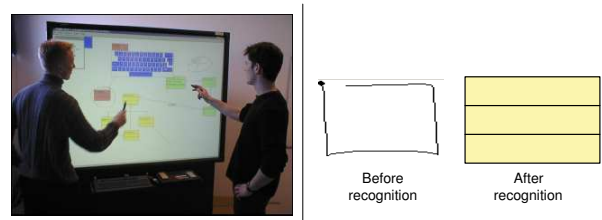


Figure 2. Left: Use of Knight on an Electronic Whiteboard. Right: Gesture Recognition in Knight

3.2 Distributed Knight

The *Distributed Knight* tool is an extension to the Knight tool, and has been implemented to support synchronous, distributed collaborative modelling in which modellers at geographically separate locations may work on the same UML model and diagrams at the same time. If two or more users are working on the same diagram, changes to the diagram made by one user is immediately reflected in the other users' visual representation of the diagram. Furthermore, presence and awareness of other users is supported by, e.g., showing tele cursors, remote selection, by providing remote visualizations of users editing elements (<http://www.groupware-patterns.org/>).

Knight uses existing, well-known technology such as a whiteboard as metaphor for interaction in order to benefit from the usability characteristics of these tools. Similarly, Distributed Knight uses a tailored, stand-alone instant messaging client, *AwareMessenger* as session management user interface [9] (Figure 3). Apart from being

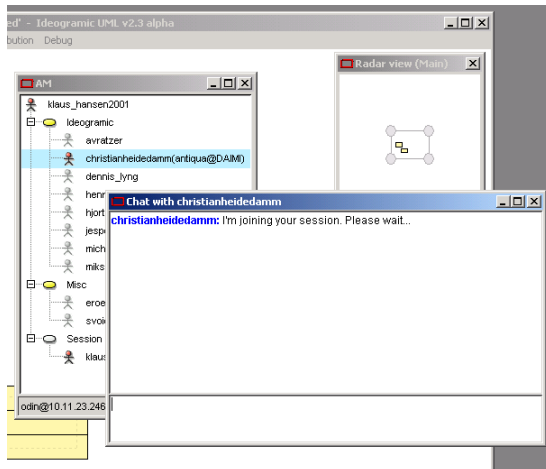


Figure 3. AwareMessenger

an ordinary instant messenger (implemented using type-based publish/subscribe), *AwareMessenger* also subscribes to join session and leave session events. In this way, potential collaborators get awareness of which sessions are ongoing, and they might want to join a relevant session. The figure shows a concrete example of this: since *klaus_hansen2001* has started a session (and invited *christianheidedamm*), a transient *session group* is created in *AwareMessenger*. *AwareMessenger* also provides a user interface for creating, joining, and inviting to sessions.

By combining the usability characteristics of the stand-alone Knight tool with a lightweight distributed collaboration approach, Distributed Knight potentially provides flexible and powerful distributed modelling support.

3.3 Architecture

The central component in Distributed Knight is a *Repository* of instances of UML metamodel classes. The UML metamodel defines classes such as *Association*, *Class*, and *Package*, each of which has a corresponding class in the Knight implementation.

A number of components, such as the *workspace* for graphical presentation of the model and the *radar view* for a thumbnail overview of the model, collaborate through the *Repository* by means of *Commands* and *Observers* [8]: all changes to the *Repository* are encapsulated in fine-grained hierarchical Composite Commands, and changes are propagated to *Observers*. All commands are undoable and are put in a *CommandHistory*.

The native save format of the Knight tool follows the OMG XML Metadata Interchange (XMI; [11]) standard for serialization of metamodels. This means that elements in the *Repository* are serialized according to the exact structure of the UML metamodel, e.g., with composed elements being serialized as nested XML tags.

3.3.1 Implementing Distributed Collaboration

Each stand-alone Knight instance participating in a collaboration session contains a *Repository* component that contains model and diagram data for that instance. In Distributed Knight, the *Repository* objects are replicated for each client of a distributed session, and changes to the *Repository* must therefore always be propagated to all the other clients in order to maintain consistency. The replication is necessary to achieve the responsiveness expected for an interactive application[1].

A *Distribution* component in Distributed Knight contains most of the distribution functionality and makes use of type-based publish/subscribe to implement distributed communication. It should be noted that stand-alone Knight is not based on type-based publish/subscribe. The *Distribution* component observes the *Repository* as well as the *CommandHistory*. When a *Command* is executed, the *Distribution* component will learn which elements have changed through its observer on the *Repository*, and the *Distribution* component will know how to group the changes together and publish them through its observer on the *CommandHistory*. This is needed since it may not make sense to publish individual changes to the *Repository*. For example, when the user creates an *Association*, two *AssociationEnds* are automatically created but not shown, and it does not make sense to publish the *Association* without the *AssociationEnds*.

The actual propagation of changes to other clients is done by publishing *SessionDataEvents* (Figure 4). A *SessionDataEvent* contains a list of data changes, each representing either a created, changed, or deleted element. These data changes are straightforwardly represented as XMI

data. The next section will explain the events in Distributed Knight in more detail.

3.3.2 The Event Hierarchy

All events are automatically equipped with the unique ID of the publisher (accessible through the `getSenderID` method, cf. Figure 1). This is useful when a publisher is also a subscriber on the same event types, since in many situations, the subscriber should ignore the events it has published itself.

The *UserEvent* basically keeps track of which user creates an event (Figure 4). As shown, events are responsible for implementing *getData* and *setData* methods for serializing and deserializing. These methods are not necessary in a language such as Java that provides default serialization and deserialization.

Events for actual collaboration in Distributed Knight are all connected to sessions in which users participate (*SessionEvent*). An example of non-session events are *InstantMessagingEvents* used by *AwareMessenger* for, e.g., sending chat messages between clients (see Section 4.2).

SessionManagementEvents handle invitations to sessions, requesting lists of active sessions, joining and leaving sessions, etc.

The *MouseEvent* presented above (Figure 1) is shown as an example of a *SessionAwarenessEvent* that is used to give awareness of other users' actions.

4 Discussion

Generally, type-based publish/subscribe has worked well for implementing Distributed Knight, especially in terms of decoupling and ease of development. Below, we discuss advantages and disadvantages on aspects of using type-based publish/subscribe.

4.1 Integration into the Object-Oriented Paradigm

Type-based publish/subscribe allows developers to work with normal types and objects from native object-oriented languages. This is in contrast to the traditional subject-based and content-based variants of publish/subscribe that force developers to shift to a more primitive communication level based on name/value pairs. By avoiding this shift, type-based publish/subscribe aids in developing distributed applications, and potentially makes the result more understandable and maintainable.

The notion of *event hierarchy* is a natural successor to the nested subjects in subject-based publish/subscribe, even though the event hierarchy in type-based publish/subscribe does not have to be strictly hierarchical. It is our experience that the event hierarchy provides a natural coarse-grained filtering. The *AwareMessenger*, which is only concerned with session management and awareness, can ignore the

part of the event hierarchy that is related to session data, and this is accomplished simply by only subscribing to the relevant parts of the event hierarchy.

An important part of allowing developers to work with native classes, is that type-based publish/subscribe allows the developer to *model* the “communication mechanisms” of an application, just like one can model a problem or solution domain in general. This model is reflected in the event hierarchy, which may then, e.g., be reused across applications. An example of such a reuse — apart from *AwareMessenger* — is our preliminary prototype of a distributed version of Rational Rose using the mechanisms described in this paper and building on the event hierarchy [10].

In this respect, the event hierarchy may eventually evolve into a part of a general distribution library – something that is hard to imagine for the traditional publish/subscribe variants. This in fact further emphasizes the strong relation of type-based publish/subscribe to the object-oriented paradigm, through a principle of code and model reuse.

4.2 Decoupling

Using publish/subscribe results in *space decoupling*, since publishers do not know the location of subscribers and vice versa. This has been useful for creating new client types without modifying existing client types; the *AwareMessenger* and the distributed version of Rational Rose are examples of this.

The *flow decoupling* of publishers and subscribers helps in achieving interactive performance in an interactive environment such as Distributed Knight. Publishers publish events without waiting for results and subscribers receive data in the form of events without explicitly waiting for data. This puts extra constraints, however, on either a server or individual clients to handle ordering and dependencies between distributed commands.

Other kinds of decoupling that (type-based) publish/subscribe supports, but which have not been investigated fully in the context of Distributed Knight, are *time decoupling* in which publishers and subscribers do not need to be available at the same time, and *data decoupling* in which subscribers only receive data that they are interested in, and in which middleware may modify data if needed. These types of decoupling would be interesting to investigate further in the future.

Decoupling can also be problematic in some respects: for the initial synchronization of a client joining a session, exactly one client already in the session should send its data to the joining client. This is awkward using publish/subscribe alone, and we solve it by combining publish/subscribe with RMI: publish/subscribe is used to discover the location of one of the session clients, and RMI is used for transferring data from that location.

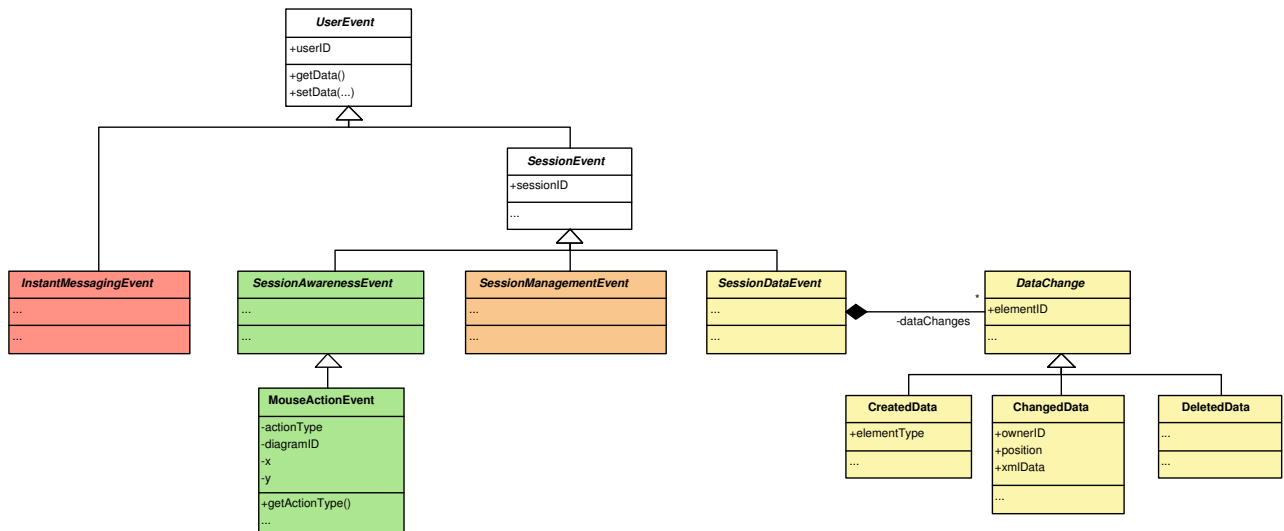


Figure 4. Part of the Distributed Knight Event Hierarchy

4.3 Performance and Scalability

Many events need to be *pruned* for performance reasons. One example is the MouseActionEvents: clients may produce a very large number of such events as the result of user actions, and it is inefficient to publish all of them as events, since the handling of the events involves updating a display. Another example, this time taken from the Session-DataEvents, is when the user moves an element in the user interface. During the move, the element will be changed many times, but it is inefficient to publish all the intermediate states. Instead, special awareness events are used in order to, e.g., show a ghosted outline of the element being moved. Due to this inefficiency, we prune all but the last ChangedData object. It is important to note that pruneable events must be idempotent; in the case of the MouseActionEvent, this means that absolute mouse positions are transmitted instead of deltas.

A promise of type-based publish/subscribe is scalability, based among other on the use of remote content filtering. Our current implementation of publish/subscribe evaluates content filters locally at the subscriber. It can be argued that this is not a problem for an interactive application such as Distributed Knight: sessions and servers will serve only a small number of users in our settings, and except for initial synchronization only small amounts of data are transferred. If larger amounts of data were to be transmitted, or a large number of clients were to be connected it would be necessary to reconsider this. A major reason for choosing the current strategy is, however, the simplicity and flexibility of decentralization.

5 Status and Related Work

Distributed Knight has been implemented in a prototype version supporting class diagram and freehand annotations

as presented in this paper and has been successfully evaluated in a series of usability studies.

The largest area for future work is work on concurrency control. Currently concurrency control in Distributed Knight is (very) optimistic and even though our user studies indicate that collaboration awareness — e.g., awareness of which gesture another user is currently drawing — hinders conflicts to a large extent, some of the many solutions to conflict resolution and ensuring consistency have been proposed for group editors will need to be investigated and implemented. However, we expect the actual implementation, and suitable solutions, to be dependent on the hierarchical nature of UML models. It would be possible to take advantage of this, e.g., to provide a relatively fine-grained determination of which actions are in conflict with each other based on the hierarchical structure of UML.

In the case of object-oriented modelling, and software development in general, few systems exist for distributed, collaborative work. To our knowledge, only two commercial object-oriented modelling applications support distributed, collaborative modelling to some extent: the one is Cittera by CanyonBlue (<http://www.canyonblue.com>), and the other is Embarcadero Describe (<http://www.embarcadero.com/products/describe>). Both applications allow distributed developers to collaboratively and synchronously edit Unified Modeling Language (UML; [12]) models and diagrams. But even though these applications provide support for synchronous editing of diagram and model elements with immediate updates, they require extensive setup and have little support for awareness of others' actions and support for communication *about* models. Distributed Knight tries to complement them in the area of lightweight and flexible modelling. Moreover, our usability studies indicate that articulation work is important also for object-oriented modelling.

A major part of the research presented in this paper has been to provide the first, large-scale use of type-based publish/subscribe. Previous examples of use, as in [7], have been small examples, even though other variants of publish/subscribe have been used for large systems. We have obviously built upon the research of Eugster et al., but our contribution with respect to type-based publish/subscribe is to discuss *how* and *why* type-based publish/subscribe may be used for distributed collaboration in interactive systems when seen in relation to other publish/subscribe abstractions.

6 Summary

This paper has introduced the implementation of the *Distributed Knight* tool for synchronous, collaborative distributed modelling. Distributed Knight extends the Knight tool for co-located modelling, and allows modellers to collaborate in lightweight and flexible ways. In particular, Distributed Knight uses an augmented instant messaging client as the main user interface for collaboration management and allows developers to use the variety of input devices that the Knight tool supports including electronic whiteboards, tablet PCs and ordinary desktop PCs.

Distributed Knight has been implemented using the type-based publish/subscribe distributed communication abstraction that allows publishers to publish events that subscribers may receive based on the details of their subscriptions. Type-based publish/subscribe extends traditional publish/subscribe abstractions by basing events on objects, i.e., instances of application-defined types. This has the advantage that publishing and subscribing to events fit into the native object-oriented language that a developer is using.

An important consequence of the integration of type-based publish/subscribe and object-oriented languages is that events may be *modelled*. The Distributed Knight case presents a specific instance of this in the form of an event hierarchy that models the communication between distributed collaboration tools.

Finally, the Distributed Knight tool has been successfully evaluated in a series of usability studies pointing towards that the tool may eventually enhance distributed, collaborative modelling practice.

Acknowledgements

This work described in this paper has been partly supported by the software part of the ISIS Katrinebjerg competency centre <http://www.isis.alexandra.dk/software/>.

References

- [1] M. Beaudouin-Lafon, editor. *Computer Supported Cooperative Work*. Wiley, 1999.
- [2] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
- [3] CNN. Telework trend rooted in convenience. [http://www.cnn.com/2001/TECH/ptech/11/12/telework.comdex.idg/index.html%](http://www.cnn.com/2001/TECH/ptech/11/12/telework.comdex.idg/index.html%2001), 2001.
- [4] A. Cockburn. *Agile Software Development: Software Through People*. Addison-Wesley, 2001.
- [5] G. Cugola, E.D. Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of ICSE'98*, pages 261–270, 1998.
- [6] C.H. Damm, K.M. Hansen, M. Thomsen, and M. Tyrsted. Creative object-oriented modelling: support for intuition, flexibility, and collaboration in CASE tools. In *Proceedings of ECOOP 2000*, pages 27–43, 2000.
- [7] P.T. Eugster, R. Guerraoui, and C.H. Damm. On objects and events. In *Proceedings of ACM OOPSLA 2001*, pages 131–146, 2001.
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Software*. Addison-Wesley, 1995.
- [9] K. M. Hansen and C. H. Damm. Instant collaboration: Using context-aware instant messaging for session management in distributed collaboration tools. In *Proceedings of NordiCHI 2002*, pages 279–282, 2002.
- [10] K.M. Hansen. Activity-centred tool integration: Using type-based publish/subscribe for peer-to-peer tool integration. In *Proceedings of the ESEC 2003 Tool Integration in System Development Workshop*, 2003.
- [11] OMG. XML Metadata Interchange 1.0. Technical Report formal/2000-06-01, Object Management Group, 2000.
- [12] OMG. Unified Modeling Language specification 1.4. Technical Report formal/01-09-67, Object Management Group, 2001.
- [13] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
- [14] TIBCO. TIB/Rendezvous white paper. Technical Report <http://www.rv.tibco.com/>, TIBCO Inc., 1999.