Building Distributed Collaboration Tools using Type-Based Publish/Subscribe

Christian Heide Damm and Klaus Marius Hansen

Department of Computer Science,
University of Aarhus,
Åbogade 34, 8200 Aarhus N, Denmark
{damm,marius}@daimi.au.dk

Abstract. We present an extension, *Distributed Knight*, to the *Knight* tool for collaborative, co-located, and gesture-based object-oriented modelling. Distributed Knight supports distributed collaboration and is built using a type-based publish/subscribe mechanism. We discuss how this architecture generalises to the distribution of data instantiated from general MOF-compliant metamodels and argue that type-based publish/subscribe mechanisms are useful for building distributed collaboration tools.

1 Introduction

The motivation for building a distributed collaboration tool for object-oriented modelling comes from two perspectives: from a *Usability perspective*, how can we support distributed teams that need to work collaboratively on object-oriented models? From a *technological perspective*, which technological primitives are appropriate for distributed collaboration tools? We consider the second perspective in the context of the *Knight* tool.

The Knight tool was originally conceived of as a tool to support *co-located* collaborative UML modelling [2, 10]. Based on observations of modelling practice, we designed and implemented a tool that supports the kind of modelling work that is usually performed on traditional whiteboards. Figure 1 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* to create, delete, and modify most elements [2]. Figure 1 shows an example of creating a class using gestures.

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

Distributed Knight extends Knight to supporting collaboration between geographically distributed development sites. The current status of Distributed Knight is that any number of clients are able to work collaboratively on UML

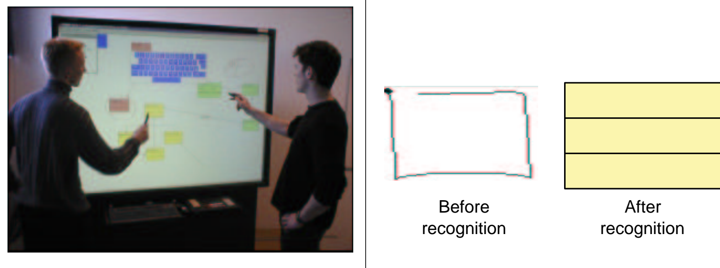

Fig. 1. Left: Use of Knight on an Electronic Whiteboard. Right: Gesture Recognition in Knight

models. We have also implemented awareness of, e.g., who is joining and leaving sessions and of where the other users are working. All this has been implemented using type-based publish/subscribe mechanisms as detailed in the rest of this paper.

2 Type-Based Publish/Subscribe

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

Type-based publish/subscribe (TPS) [3] is a recent *object-oriented* variant of this interaction style. In TPS, events are *objects*, i.e., instances of native types. A 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.

2.1 Implementation and Example

The type-based publish/subscribe variant has been implemented for Java [4, 6], and it was recently proposed to extend Java to directly support TPS (Java_{PS}, [5]). It has been shown that Java_{PS} enables better support for TPS than Java does [1]. We have chosen to implement TPS servers and clients in the Itcl language [8]. However, due to the interpreted nature of Itcl and its powerful reflection mechanisms, it is not necessary to actually extend Itcl in order to obtain the advantages that Java_{PS} offers over Java.

Figure 2 shows an example of an event type from Distributed Knight (using the more familiar Java syntax). The *MouseActionEvent* is an awareness event, and it contains the information that a given user has moved/pressed/... the mouse in a given place. Also shown in Figure 2 are examples of how clients publish and subscribe to MouseActionEvents.

```
// Event type
public class MouseActionEvent extends SessionAwarenessEvent {
  private int actionType;
  private int diagramID;
  private float x;
  private float y;
  public int getActionType() {return actionType;}
  ...
  public MouseActionEvent(int userID, ..., int actionType, ...) {
    SessionEvent(userID,sessionID);
    this.actionType = actionType;
    ...
  }
}

// Publishing
UML.MouseActionEvent event = new UML.MouseActionEvent(userID,
   sessionID, ButtonPress, diagram.ID, mousePos.x, mousePos.y);
publish event;

// Subscribing
Subscription s = subscribe (UML.MouseActionEvent event) {
    // content filter
    if(event.getSenderID() != PublishSubscribe.clientID) {
       // only accept mouse actions in other sessions
       return (event.getSessionID() == sessionID);
    } else {
       // ignore our own mouse actions
       return false;
    }
  } {
    // event handler
    ... indicate the mouse action in the user interface ...
  }
s.activate();
```

Fig. 2. Publishing and Subscribing to Events

3 Knight Architecture

The central component in the Knight Architecture is a *Repository* of UML metamodel objects. The UML metamodel defines classes such as *Association*, *Class*, and *Package*, each of which has a corresponding class in the Knight implementation. Of particular interest are the hierarchical composition structure between model elements: Packages compose model elements (including other Packages), Associations compose AssociationEnds, Classes compose Attributes and Operations, etc. According to the UML, a composed element can only be composed by one composer at any given time.

A number of components, such as the workspace for graphical representation of the model and the radar view for a thumbnail overview of the model, collaborate through the Repository by means of *Commands* and *Observers* [7]: all changes to the Repository are encapsulated in fine-grained hierarchical Commands (using the Composite pattern). Changes to the Repository via Commands trigger notifications to other components that act on these changes. All commands are undoable and redoable and are put in a *CommandHistory*.

The native save format of the Knight tool is the XML-based XMI standard for serialization of metamodels [11]. 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.

4 Implementing Distributed Collaboration

A distribution component in Knight contains most of the distribution functionality. The distribution component is surprisingly isolated from the rest of Knight and has required only minor changes to the existing components.

4.1 Propagating Changes in the Repository

The architecture of Knight, described above, is essential for the approach. 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 Observer structure that is used internally in Knight is ideal for discovering changes to the model elements in the Repository. The distribution component registers itself as an observer on the Repository and will be notified whenever a model element has been created, changed, or deleted.

The distribution component also observes the CommandHistory, so that it knows when a Command has been executed or unexecuted. This is important because Distributed Knight propagates Repository changes according to Commands. A Command may create/change/delete any number of model elements, and these changes should be propagated to the other clients as one unit. For example, when the user creates an Association, two AssociationEnds are automatically created but not shown, and it does not make sense to propagate the Association without the AssociationEnds.

The well-structuredness of UML data (or, in general, any MOF-compliant metamodel data [9]) is essential for how we propagate changes between clients. Each UML model element is serialized into exactly one XML tag with a unique ID. That is, all the data concerning the model element is represented in the XML data, including references to other model elements. The only information that is *not* represented in the XML data is the “context” of the model element, i.e., the identity of the element that composes the model element and the position of the model element among the owner’s composed elements.

Since the save format of Knight already conformed to the XMI standard, it was straight-forward to generalize the existing load/save functionality to be able to deserialize/serialize isolated model elements.

The actual propagation of changes to other clients is done by publishing *SessionDataEvents*, as shown in Figure 3. A SessionDataEvent contains a list of data changes, each representing either a created, changed, or deleted element. In order to create an element, only the type and unique ID of the element are needed, where the type is the qualified class name that is also used by the XMI standard. Once an element has been created, it can be changed any number of times. A change involves all the information described above, i.e., the internal data of the element and the context of the element. Finally, an element can be deleted.

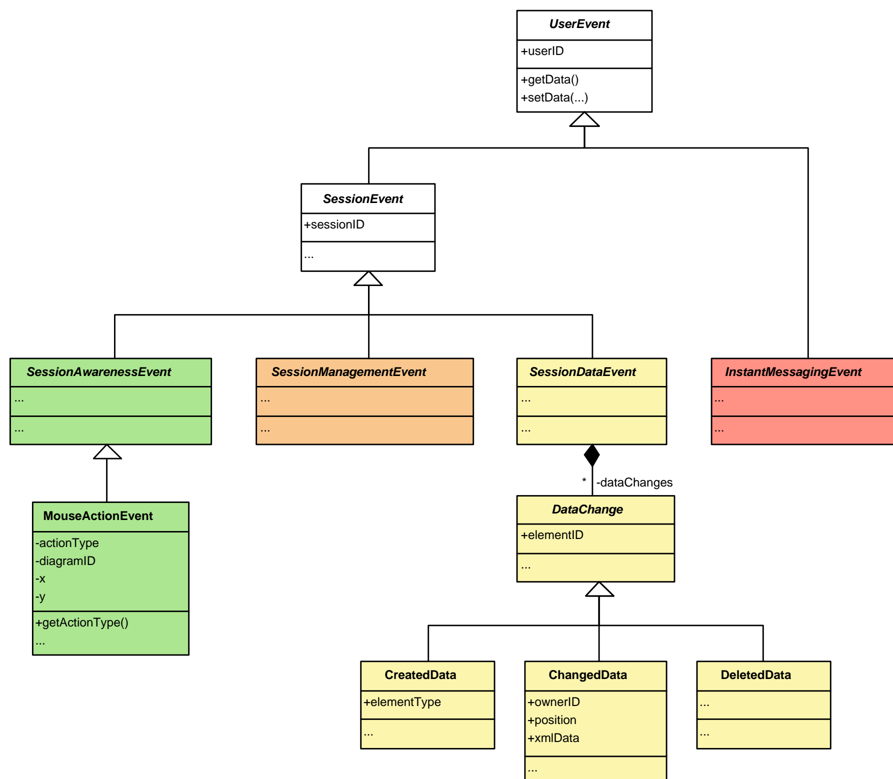

Fig. 3. Part of the Distributed Knight Event Hierarchy

As an example, when the user creates a UML Class, the SessionDataEvent will contain the following data changes:

1. `CreatedData: Class`

```
2. ChangedData: Class
3. CreatedData: ClassView
4. ChangedData: ClassView
```

4.2 The Event Hierarchy

All events are automatically equipped with the unique ID of the publisher (accessible through the `getSenderID` method, cf. Figure 2). 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 3). As shown, events are responsible for implementing *getData* and *setData* methods for serializing and deserializing.

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 the instant messenging client for, e.g., sending chat messages between clients (described below).

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

The *MouseActionEvent* presented above is an example of a *SessionAwarenessEvent* that is used to give awareness of other users' actions.

5 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 pros and cons on aspects of using TPS.

5.1 Decoupling

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: take as an example our implementation of a context-aware instant messenging client: apart from being an ordinary instant messenger (implemented using TPS), the client 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.

Other types of clients that we are considering implementing and that would be both useful and require only moderate effort include a code generation client and a "ticker tape" application showing digests of the actions made in sessions.

The *flow decoupling* of publishers and subscribers helps in achieving interactive performance in a user interface-oriented environment such as Distributed Knight. This puts extra constraints, however, on either a server or individual clients to handle ordering and dependencies between distributed commands.

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 ackward using publish/subscribe alone, and we solve it by combining publish/subscribe with remote method invocation (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.

5.2 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 SessionDataEvents, 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, 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 subscribe, mainly for simplicity reasons. 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.

The current implementation of TPS is based on a central server, again mainly for simplicity reasons. The Java implementation of TPS described in [12] is implemented using IP multicast, with one multicast group per event type, and this is presumably much more efficient than our server-based implementation.

References

1. Damm, C.H., Eugster, P.T., and Guerraoui, R: Abstractions for Distributed Interaction: Guests or Relatives. In progress.
2. Damm, C.H., Hansen, K.M., Thomsen, M., and Tyrsted, M. (2000). Creative Object-Oriented Modelling: Support for Intuition, Flexibility, and Collaboration in CASE Tools. In *Proceedings of ECOOP2000*, Sophia Antipolis and Cannes, France, June.
3. Eugster, P.T.: Type-Based Publish/Subscribe. PhD Thesis, EPFL, December 2001.
4. Eugster, P.T. and Guerraoui, R.: Content-Based Publish/Subscribe with Structural Reflection. In *Proceedings of Usenix COOTS 2001*, pp. 131-146, January 2001.

5. Eugster, P.T., Guerraoui, R, and Damm, C.H.: On Objects and Events. In *Proceedings of ACM OOPSLA 2001*, pp. 131-146, October 2001.
6. Eugster, P.T., Guerraoui, R., and Sventek, J.: Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction. In *Proceedings of ECOOP 2000*, Springer-Verlag, pp. 252-276, June 2000.
7. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995) *Design Patterns. Elements of Reusable Software.* Addison-Wesley.
8. McLennan, M.J. (1993). [incr Tcl]: Object-Oriented Programming in Tcl/Tk. In *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11.
9. Object Management Group (2001). *Meta Object Facility 1.3.1.* OMG document formal/01-11-02.
10. Object Managment Group (2001). *Unified Modeling Language Specification 1.4.* OMG document `formal/01-09-67`.
11. Object Managment Group (2001). *XML Metadata Interchange 1.0.* OMG document `formal/2000-06-01`.
12. Sarni, S.: Design and Implementation of a Type-Based Publish/Subscribe Architecture. Technical Report, EPFL, Lausanne, Switzerland, June 2001.