Tool Integration: Experiences and Issues in Using XMI and Component Technology

Christian Heide Damm, Klaus Marius Hansen, Michael Thomsen, Michael Tyrsted
Department of Computer Science, University of Aarhus,
Aabogade 34, 8200 Aarhus N, Denmark
Email: {damm, marius, miksen, tyrsted}@daimi.au.dk

Abstract

It is impossible to implement one tool that supports all activities in software development. Thus, it is important to focus on integration of different tools, ideally giving developers the possibility to freely combine individual tools. We discuss how tools can be integrated even in the context of conflicting data models, and provide an architecture for doing so, based on component technology and XML Metadata Interchange. As an example, we discuss the implementation of an electronic whiteboard tool, Knight, which adds support for creative and collaborative object-oriented modelling to existing Computer-Aided Software Engineering through integration using our proposed architecture.

1. Introduction

A skilled craftsman at work uses a variety of tools, each one tailored to the concrete work situation, and he effortlessly changes between these with quick alternations. Software developers, on the other hand, often find themselves constrained to few tools, as tools in general do not integrate well. These tools are often closed with respect to extension, and they often use idiosyncratic file formats and interfaces making the shifting between tools hard.

One tool will never be appropriate for *every* activity in software development. Thus, integration of tools is important. In this paper we report our experience with the integration of two commercially available CASE tools with a tool, *Knight*, that supports collaborative object-oriented modelling. The integration highlights concrete experiences with two current integration strategies: using a standardised interchange format in the form of XML Metadata Interchange (XMI, [25]) and using component technology in the form of Microsoft COM [19]. Our experiences with integrating CASE tools with different, although overlapping, data models also apply to other kinds of tools, and the discussions in this paper will thus cover general aspects of tool integration.

The rest of this paper is structured as follows: section 2 discusses tool integration in general and related work. Section 3 gives a brief overview of the Knight tool, i.e., its use, implementation, and software architecture. Section 4 describes and discusses our two experiments with tool integration and proposes an architecture for integration. Finally, section 5 concludes.

2. Tool Integration

Integration may be viewed on an architectural level, i.e., as cooperation between high-level components interacting via high-level connectors. A distinction can be made between *data* and

processing components, with processing components operating on the data components. The data in the data components may either be shared by several processing components or be separate. In either case, the processing components need to communicate in order to cooperate. If they are running simultaneously, they may do this by changing the shared data concurrently or by communicating changes to the replicated data. If they are running asynchronously they typically cooperate by changing the shared data, or communicating their changes to a shared third party. Table 1 illustrates this taxonomy for tools working on a common, logical core of data, and shows some typical applications. The taxonomy is inspired by Ellis et al.'s taxonomy of Computer Supported Cooperative Work [7].

Time \ Data	**Shared**	**Separate**
Asynchronous	Import/export	Merging configuration management systems
Synchronous	Components in same process	Component technology interaction between applications

Table 1. Component collaboration taxonomy and typical applications

Our case study (described in section 3) focuses on the situation in which the processing components operate on data based on the UML metamodel. The cooperation may be performed in a number of ways, e.g.:

- as separate tools using a common interchange format (*asynchronous, shared*),
- as separate tools working on different aspects of the separate data (*asynchronous, separate*),
- as components collaborating through a well-defined interface to the common, logical core of data (*synchronous, shared*), or
- as interaction between separate tools via component technology (*synchronous, separate*).

Our case study concentrates on asynchronous integration with shared data and synchronous integration with separate data.

2.1. Related Work

Viewed as a development process, integration of components or applications has three important aspects: *designing architecture*, *creating components*, and *using components*. Designing architecture is concerned with choosing an appropriate architectural style, creating components is concerned with assigning functionality to components and relating this functionality to the environment of the components, and using components is concerned with finding, adapting, and assembling components.

In choosing an appropriate architectural style, several possibilities exist [17], including

- *File Level*. In this case the file is the common data abstraction. Using a common interchange format for CASE tools, as we do for asynchronous integration with shared data, builds on this architectural style. This is also a common way of integrating UNIX tools, typically using the Pipes and Filters [1] architectural pattern.
- *Single System*. Many programming environments build on a common interface and common data representation in a monolithic way. This integration is synchronous, uses shared data, and is in general hard to maintain and extend.

- *Program Database*. Using the Blackboard/Repository [1] architectural pattern, a loose coupling between integrated tools may be obtained. Given a common application programming interface to UML data, a synchronous integration with shared data could be implemented using this architectural style.
- *Message Facility*. Several integration efforts, such as the Field environment [17], builds on a Data Abstraction [21] architectural style with an emphasis on message passing. Our integration of separate tools via component technology essentially falls into this category.
- *Hierarchical*. One of the most ambitious efforts to create a reference architecture for tool integration [2] was built on hierarchical Layers [1] with layers for repository services, data-integration services, adding tools, process-management services, and user-interface services all built on a layer of message services. One of the major problems with this architecture was that it was complex and constraining: it fixed the integration of tools instead of allowing for open-ended lightweight integration. Rather than developing a fixed reference architecture for integration, we concur with Tichelaar et. al [22] in that emphasis should rather be put on how to coordinate independent components in flexible ways.

Components need to be designed with adaptation and integration in mind. This process is not well understood. However, guidelines for creating coordination components [22] and tailorable, reusable frameworks [6] may be useful in this.

Using components consists of four phases: *component qualification, component adaptation, component assembly,* and *system evolution* [9]. *Component qualification* tries do determine fitness for use of components using discovery and evaluation techniques. *Component adaptation* is concerned with modifying components for new contexts. The actual adaptation technique is determined by the nature of components (white box, grey box, or black box components), and wrapping, bridging, and mediating are well-known techniques for this [8]. *Component assembly* presupposes a well-defined software architecture in which components can be integrated. *System evolution* has the component as unit of evolution meaning that replacement of components may be problematic. Our integration efforts use a mixture of such techniques for adaptation and discuss experiences with the process of integration. Furthermore, we propose an architecture for synchronous integration based on separate data.

3. The Knight Tool

As an example of tool integration we use the Knight tool. The Knight tool extends existing CASE tools by providing an alternative user interface, which has support for creative, flexible, and collaborative modelling. The tool is implemented in Itcl [14], which is an object-oriented extension of Tcl/Tk [16], and it uses Microsoft COM [19] for tool integration. More information can be found at `http://www.daimi.au.dk/~knight`.

3.1. Use of Knight

Knight uses a touch-sensitive electronic whiteboard (currently a SMART Board – `http://www.smarttech.com`) as input medium (Figure 1a),. Since a major design goal of the Knight tool is to make the interaction with the tool similar to that of an ordinary whiteboard, the user interface is very simple: it is a white surface (Figure 1b), on which users can draw UML diagrams with dry pens.

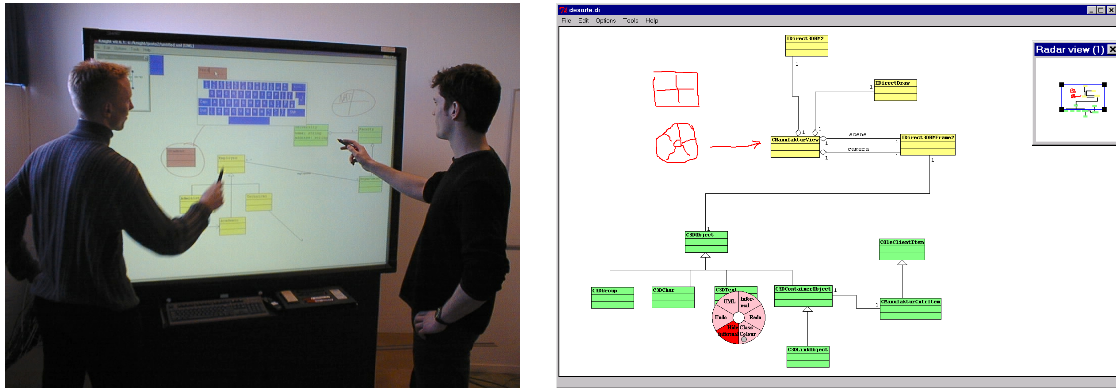

Figure 1. a) Use of Knight on an electronic whiteboard,
b) Knight user interface with radar window

The Knight tool operates in two modes: an informal freehand mode and a formal UML mode. In freehand mode, the user may add arbitrary annotations to the diagram. In UML mode, the strokes of the user are interpreted as UML elements. As an example, to create a new class, a user can draw a rectangle with a pen on the workspace drawing surface, which the tool will then interpret as a class (Figure 2). This results in an interaction that is direct and intuitive [3].

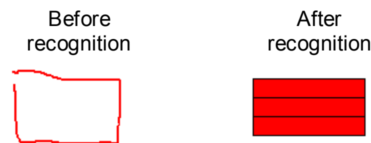

Figure 2. Recognition of the gesture for a class

The same type of interaction is used when the user wants to create other types of UML model elements. If the user wants an association between two classes, the user just draws a line between the classes; if the user wants to change the association to an aggregation, the user draws a diamond at one end of the association. Moreover, there are gestures for common operations such as deletion and movement of elements. To enable easy access to less common operations, a context-dependent pie menu [13] is provided (Figure 3a). The user may either press the pen against the drawing surface for a short while in order for the menu to pop up, or make a short stroke in the direction of the desired command.

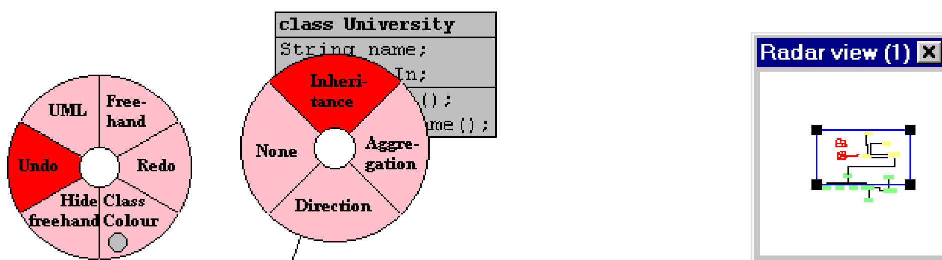

Figure 3. a) Context-dependent pie menus, b) Radar window

A small radar window (Figure 1b, Figure 3b) is used to provide context awareness and a potentially infinite workspace. To pan, the user drags the rectangle in the window. To zoom, the user uses the black handles to resize the rectangle.

3.2. Software Architecture

The software architecture of Knight follows the Repository architectural pattern [21], the Repository in Knight being a set of UML diagrams. The architecture is shown in Figure 4.

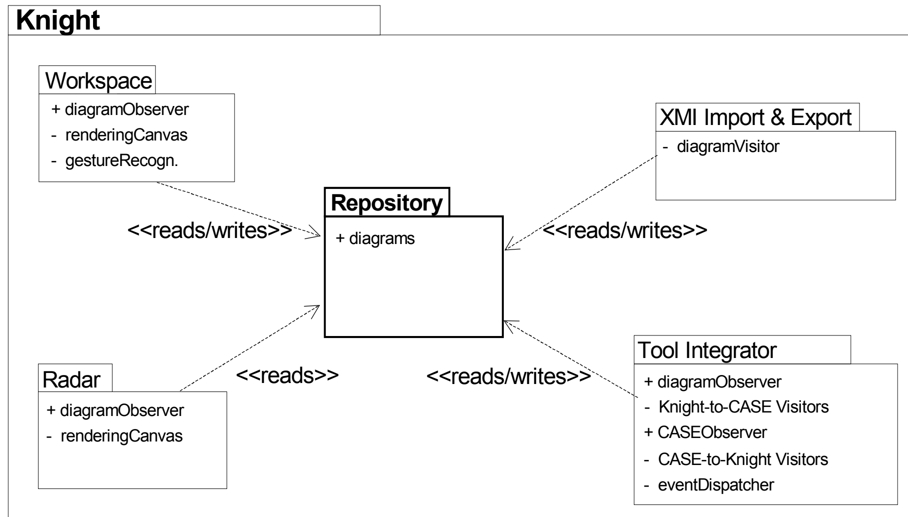

Figure 4. Logical view of the software architecture of Knight

A class diagram in Knight is basically a subset of the UML metamodel for class diagrams, extended with layout information (position, size etc.). The model is shown in Figure 5.

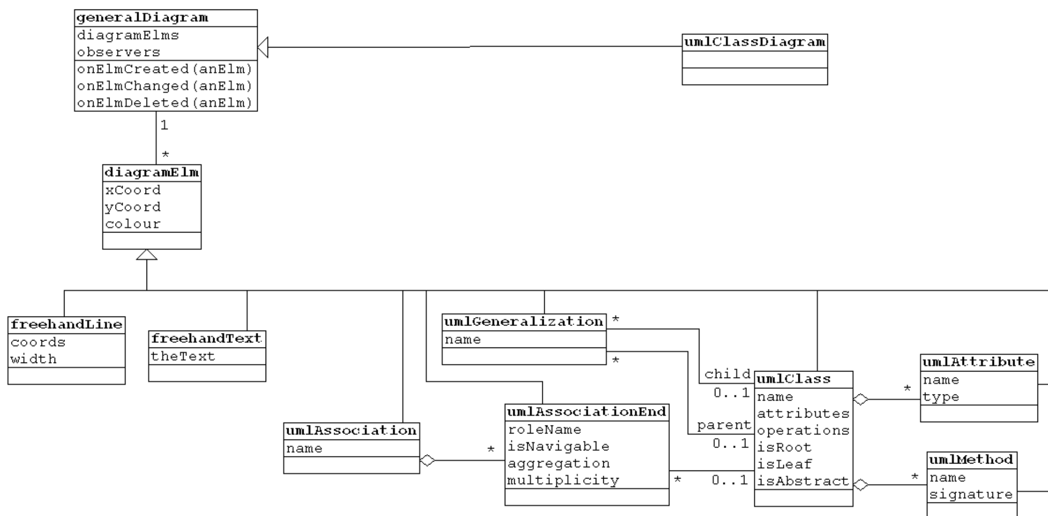

Figure 5. Simplified object model

It is possible to attach an Observer [8] to a diagram, which will be notified when elements have been created, changed, or deleted. Observers are used for many purposes in Knight. For example, the workspace uses an Observer in its implementation: when a diagram element is created, the workspace creates a corresponding visual representation of the new diagram element, etc. The small radar window is also observing a diagram. Observers are used for various other things, such as debugging and integrating with CASE tools.

In addition to observing, the workspace also writes to the Repository, namely when the user interacts with the workspace. Writing in the Repository is implemented by using a combination of the Composite and the Command patterns [8], providing fine-grained multi-level undo/redo functionality.

4. Tool Integration

Having introduced the subject of our case study, we now discuss the details of the integration. As mentioned, the discussion will focus on asynchronous integration with shared data and synchronous integration with separate data. In order to motivate our choice of these two strategies, we introduce three typical use scenarios that will be referred to in the discussions. In all three scenarios, Mike and Sarah are part of a development team that develops an administrative system for a university. Mike is a domain expert, and Sarah is an object-oriented developer. They use the Knight tool for collaborative modelling activities, and Sarah uses a traditional CASE tool for refinement and detailed design.

Use scenario 1: creating a new diagram in Knight. Mike and Sarah are about to model the payroll of the administrative system. To do this they gather other relevant domain experts for a modelling session. At the beginning of the session, Mike creates a blank diagram in Knight. During the session, the topics of the discussion are recorded using a combination of UML class diagram elements and informal sketches on the electronic whiteboard. After the meeting, Sarah transfers the diagram to the CASE tool, in which she elaborates the diagram while making use of the possibilities the CASE tool provides.

Use scenario 2: comments on a model created in Knight. Mike and Sarah have had several modelling sessions during the preceding weeks, but they have a tricky problem they cannot solve. Sarah consults a colleague, Peter, in the company she works for. Peter discusses the model at the company site with a number of modelling experts, and they analyse the model and make a number of changes to it before sending it back to Sarah.

Use scenario 3: working on an aspect of a diagram in Knight. The administrative system has now reached a state in which it is running as a prototype communicating with a central, relational university database. Sarah has used the CASE tool to generate code for accessing the database. A conceptual problem discovered during prototyping leads to a remodelling session. Sarah creates a live link to the Knight tool, and Mike and she then discuss the conceptual problem and try to fix it using Knight. As they make changes, Sarah now and then checks the effects of the changes on the database access code in the CASE tool.

4.1. Asynchronous Integration with Shared Data

Use scenario 1 describes a situation in which there is a need for transferring a model from one tool to another. An interchange format supported by both tools is an effective and simple way of achieving this. The interchange data file represents a shared artifact that the two tools can read asynchronously, or sequentially, in order to collaborate.

Use scenario 2 also describes a situation in which an interchange format is useful, but in contrast to use scenario 1, it also illustrates a situation in which synchronous integration is problematic. Sarah needs to send the model to her colleague Peter, and as Peter does not have the Knight tool, in which the model was created, at the company site, the only option is to exchange the model asynchronously.

XMI. To implement asynchronous integration using shared data, we have implemented support for XML Metadata Interchange (XMI, [25]) in Knight. XMI is an accepted Object Management Group (OMG, `http://www.omg.org`) specification that provides the basis for an interchange format for UML models. The specification is in fact more general, as it specifies a way of creating an interchange format for any data that can be described by a metamodel.

Figure 6. Description of a part of a bank

Consider the simple diagram in Figure 6, which describes a part of a bank using a UML Class diagram. The diagram is a set of data. Since it describes the structure of a set of concrete account and customer objects, it is also metadata. If a set of metadata conforms to a specific semantics and syntax, it is called a ‘model’. Since the diagram actually uses the UML class diagram notation, the diagram is a model. These two levels of abstraction comprise the two lowest levels in the OMG’s Meta Object Facility (MOF, [15]). Two higher levels are also present (see Table 2): first, the UML notation itself can be described. This leads to a so-called metamodel: a set of data that describes a set of models, one of which is our model of a bank. Second, as there are many other metamodels than the UML metamodel, the MOF introduces a meta-metamodel level that can be used to describe all metamodels.

Meta-level	MOF term(s)	Examples	Sample XMI artifacts
M3	meta-metamodel	The "MOF Model"	MOF DTD
M2	metamodel, or meta-metadata	The UML Metamodel	UML DTD, MOF XML file
M1	model, or metadata	A UML model of a bank	UML XML file
M0	data	Concrete bank account objects and customer objects	

Table 2. OMG MOF metadata architecture[1]

Based on a MOF-compliant metamodel such as the UML metamodel, the XMI standard describes a way to produce a grammar corresponding to that metamodel. This grammar can then be used to save and load models (e.g., a model of a bank), resulting in an interchange format for all models conforming to the metamodel. Figure 7 shows an extract of the XMI code exported by Knight for the Bank model. For each element in the model, an XMI element with the same name is present, and inside these elements follow further elements corresponding to the attributes of the element.

```
<Model_Management.Model xmi.id="_1">
   <Foundation.Core.ModelElement.name>New diagram</Foundation.Core.ModelElement.name>
   <Foundation.Core.Namespace.ownedElement>
     <Foundation.Core.Class xmi.id="::53umlClass0">
       <Foundation.Core.ModelElement.name>BankAccount</Foundation.Core.ModelElement.name>
       <Foundation.Core.Classifier.feature>
         <Foundation.Core.Operation xmi.id="::63umlOperation1">
           <Foundation.Core.ModelElement.name>Withdraw()</Foundation.Core.ModelElement.name>
 ...
```

Figure 7. Part of the bank model encoded in XML conforming to the UML DTD

[1] It should be noted that described four-level architecture is only the typical architecture. The number of levels is not fixed by the specification.

To specify a grammar, the XMI standard uses XML (eXtensible Markup Language, [24]) DTD's (Document Type Definitions) and the actual exchange files are then XML files conforming to this DTD. In other words, XMI specifies a set of rules for mapping a MOF compliant metamodel to a DTD, and a way of mapping a model to an XML file conforming to this DTD. The rules are not described here, as they are quite elaborate in their full detail. In this context it suffices to say that for each class in the metamodel, the rules create a grammar rule that can describe both the attributes of the class and references to elements associated to the class.

XMI Implementation. Based on the UML metamodel standardised by the OMG, several companies have produced a UML DTD using the rules in the specification. We use the UML DTD provided as part of the IBM XMI Toolkit (`http://www.alphaworks.ibm.com/tech/xmitoolkit`), as this DTD is based on the *relaxed transformation rules* allowing for the exchange of models having elements that are not fully specified.

The basic import and export is quite simple. During the import the XML file is parsed using the Expat open source XML parser (`http://www.jclark.com/xml/expat.html`). Using the callbacks from the parser, an XML parse tree is built with the XML elements as nodes and their contents as subnodes. A traversal of this tree then creates the diagram. Saving is performed analogously: the diagram is traversed and from this an XML tree is built. This tree is then streamed to a text file.

XMI Experiences. Even though the above implementation description sounds simple, we encountered a number of problems pertaining to the XMI standard. First of all, since the UML DTD is based directly on the UML metamodel it can only express what is in the UML metamodel. This is a problem since the UML metamodel is only concerned with UML *models* and not UML *diagrams* that have appearance. While this issue will most likely be solved in a future version of the UML in which the metamodel will describe diagrams [12], there is currently no standardised way of encoding presentational information in a UML XMI file.

XMI allows for extension elements to be added to each element, which can be used to encode information that is not part of the metamodel. These extensions are thus well suited as a place to store the presentational information, and this was the approach that we chose. However, since the concrete structure of the extensions is not described in the standard, different tools in practise encode this information in different ways. The consequence of this is that different tools can only exchange models and not diagrams. This is a problem, not only for the simple reason that it can be very annoying to have to re-layout a diagram, but also because positional information in a diagram often has semantics: two classes positioned close to each other will, e.g., most likely be closer related than two classes far away from each other.

The extension elements are also problematic for other reasons, especially in combination with round-trip engineering in which a number of tools import, change and then re-export the XMI file. While a tool may add any number of extensions to any element at export, another tool may also ignore their contents at import. However, the standard does not allow tools to discard extensions made by other tools when re-exporting a file. A tool must thus make sure that it stores all extensions during import even though it has no interest in their contents. We solved this in a simple way: at the import, the XML parse tree built during the parsing is scanned, and each node is then either used to create a diagram element, or it is stored in a list of ignored nodes. During the export, new nodes are then created from the current diagram contents and these are then merged with the nodes that were "ignored". Once this merged tree has been built it can be streamed to a file. This technique is also used for UML elements that our tool has no interest in such as elements from diagram types not supported by the tool.

The following scenario illustrates another complication with extensions in combination with round-trip engineering: a tool creates a UML element and an extension containing further information about the element. Another tool then imports this file, changes the state of the UML element and re-exports the file. This might result in the extension exported by the first tool being inconsistent with the new state of the UML element. While this is a general problem in integration, there is no general and simple solution to it. One possibility would be to add a timestamp attribute to each element. This would allow a tool to detect that an element, for which it has made an extension, has been changed since the extension was created.

As a more practical complication, it was difficult to find tools that could validate our exported XMI files. We are only aware of three tools that support the XMI specification and work on UML diagrams: the IBM XMI Toolkit, which can convert XMI files to and from Rational Rose files (`http://www.rational.com`), Rational Rose itself with an extra XMI plug-in (`http://www.rational.com/products/rose/support/patches`), and the open source CASE tool Argo UML ([18], `http://www.argouml.org`). The IBM Toolkit and the Rose plug-in produce XMI files that are compatible, but neither of them is compatible with ArgoUML which uses an earlier version of the XMI specification. The new XMI 1.1 specification [26] will add another format, and so will future versions of the UML metamodel. If the XMI standard is to be used extensively as an interchange format, both the XMI standard and the UML metamodel must become more stable. Otherwise, what was supposed to be a unifying format will turn into a plethora of formats that tools must struggle to support.

4.2. Synchronous Integration with Separate Data

Use scenario 1 and 2 could be well supported by the XMI integration. Use scenario 3 above justifies the need for a synchronous form of integration: different tools may work on different aspects of the same data, and quick alternation between the tools may be needed.

Synchronous integration is typically component based. The most widespread component technologies are CORBA [10] and Microsoft COM [19]. The choice of component technology is not essential for our purpose, but since most CASE tools today only support COM, we chose COM as well. Our implementation of synchronous integration with separate data is thus based on runtime COM connections between the Knight tool and the CASE tools. In this set-up, there are two phases to be considered: the initial synchronisation, i.e., a batch-transfer of data, and, subsequently, a continuous incremental synchronisation between the tools.

Initial synchronisation. If a user wants to work on an existing diagram made in a CASE tool, the initial synchronisation includes transferring the diagram to the Knight tool. It may also be the case that both tools contain diagrams that should be used in a modelling session. In this case the synchronisation is a merge of the two diagrams.

One Visitor for each direction implements the initial synchronisation. The Visitor that traverses the Knight diagram is a trivial application of the design pattern, whereas the Visitor that traverses the object graph of the CASE tool needs to be constructed externally to the CASE tool.

First, the ‘Knight-to-CASE Create Visitor’ traverses the Knight diagram. During the traversal, it builds a similar diagram in the CASE tool: when it visits a class in Knight, it creates a copy of the class in the CASE tool, etc. Second, the ‘CASE-to-Knight Create Visitor’ traverses the CASE tool diagram and copies it to Knight (this time ignoring the elements just copied *from* Knight). A logical view of the architecture for the initial synchronisation is shown in Figure 8.

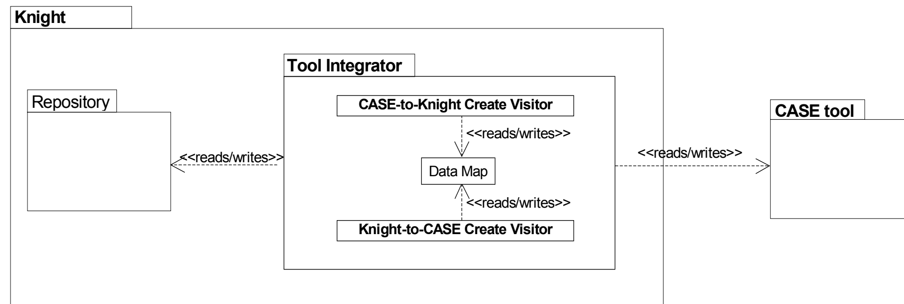

Figure 8. Initial synchronisation

While synchronising the two tools, the Create Visitors build a mapping between the data in the tools. The mapping is used in the incremental synchronisation, allowing, e.g., a change to a class in the Knight tool to be propagated to the corresponding class in the CASE tool. The Create Visitors of the initial synchronisation may also be used to support scenario 1 above: if the following continuous, incremental synchronisation is not invoked they in effect implement a traditional import and export of diagrams.

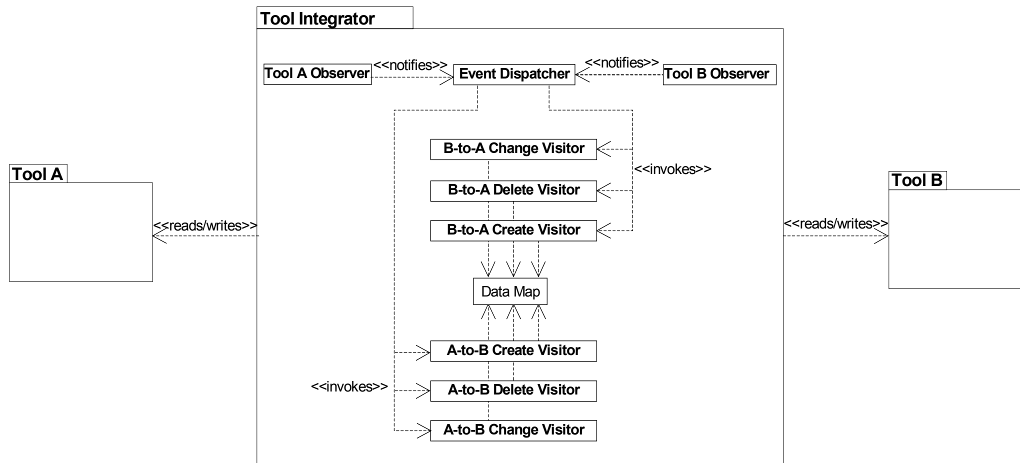

Figure 9. Architecture for synchronous integration with separate data

Continuous incremental synchronisation. In order to keep the diagrams in the two tools synchronised after the initial synchronisation, changes in one diagram should propagate to the other diagram. In our implementation, two integration-specific Observers observe the diagrams. The Observer of the CASE tool relies on COM events. The overall architecture for the continuous incremental synchronisation is depicted in Figure 9. Note that the architecture has been generalised and shows the integration of two generic tools A and B. The architecture can be used for integrating tools that have "compatible" data models, such as the UML metamodel in our case.

If, e.g., a new class is created in tool A, the A Observer will get notified, and it will then make the 'A-to-B Create Visitor' visit the new class. The 'A-to-B Create Visitor' will, as always, create a copy of the class in tool B. In a similar way, changes to, or deletion of, existing elements in tool A will be propagated to tool B, implemented by two other Visitors. Updates in tool B are propagated to tool A in a similar way. The Event Dispatcher mediates the notifications between the tools based on subscriptions.

In Figure 9, the 'Tool Integrator' is shown as a separate component, which is not *part of* any of the tools (in contrast to Figure 8). The integrator component is symmetrical and thus requires the same handles on both tools. The reason that it is a part of the Knight tool in our implementation is that Knight currently has no COM interface.

Component Technology Experiences. The architecture for integrating Knight with CASE tools is simple and effective. It has proved to be adequate for the two CASE tools we have experimented with, namely Rational Rose and WithClass (`http://www.microgold.com`), and we believe it will work for other tools as well. When integrating existing tools, there will, however, often be specific issues to consider as well as problems to overcome. This was also the case for Rational Rose and WithClass, and next we will present our experiences with these problems.

In Knight, a diagram may contain freehand drawings, which are not supported in Rational Rose and WithClass. To ensure that these drawings are not lost, we store them in the CASE tool in a special note element, which we hide from the user. In general, the solution to this problem is to find *somewhere* to store arbitrary information – and it may vary for different tools.

There are also examples of Rational Rose and WithClass diagrams containing more information than Knight diagrams, but because the Knight tool operates directly on the Rational Rose and WithClass diagrams, we chose not to store the extra information in the Knight diagrams.

There are situations, however, in which extra information in the Rational Rose or WithClass diagrams should be handled in a special way. When performing, e.g., a 'delete' followed by an 'undo', the state of the diagrams in both Knight and the CASE tool should be exactly the same as before the deletion. If the diagram in the CASE tool contains extra information, then this information must also be restored in some way. If the CASE tool supports multi-level undo, as does Knight, this is trivial. If not, a reasonable alternative is to serialise objects in CASE tools when a delete operation is performed, using their COM interface, and to de-serialise objects when an undo operation is performed. Failing this, other actions may be taken, such as hiding and showing diagram elements, and removing these elements completely when disconnecting. In Knight we solve the problem by a combination of serialising and hiding: presentational data is serialised and logical data is hidden. Notice that the same problems may occur in other situations, e.g., if elements are copied.

4.3. Discussion

Cooperative work has three aspects: *coordination, collaboration*, and *communication*. *Coordination* is realised through rules, delegation, sign-offs etc. *Collaboration* realises cooperative creation of artifacts. *Communication* is the organisational distribution of information. These three aspects of cooperative work are also relevant in tool cooperation.

Asynchronous integration. In connection to asynchronous integration with shared data, *coordination* is partly on the discretion of the developers using the integrated tools. The XMI file *is* the common artifact worked on by both CASE tools and Knight that is being *collaborated* on. The UML DTD defines the *communication* syntax and the semantics is partly decided by the semantics of the UML metamodel. The communication is open-ended, and tools may add arbitrary extensions using the XMI extension tag.

A standard interchange format is important in enabling different tools to collaborate on the same data. With XMI, a UML model can be stored in a standard way. On the other hand, many tools have needs that are not covered by the standard. These tools will have to add extensions to the standard format, thereby making full integration with similar tools hard. The evolution of HTML and browsers nicely illustrates the problem: different Web browsers and editors have created idiosyncratic mark-up, some of which has later become standard (such as frames) and some of which have remained implemented for one tool only (such as positioning mark-up). Similarly, the UML metamodel cannot describe all types of relationships between entities in class diagrams. Descriptions of method call and attribute access, e.g., are not supported. Several tools, including some program visualisation and refactoring tools, need this kind of information. Thus, it is problematic to use UML as basis for tool integration in general [5]. Just as the UML metamodel currently is being extended to cover diagram specification, the inclusion of code information may be considered. It is obvious, though, that the agreement on a standard will always lag behind the needs for cooperation between tools.

Synchronous integration. In connection to our implementation of synchronous integration with separate data, *coordination* is mainly handled by the Event Dispatcher component, which, among other things, ensures that an update in one tool will not lead to an infinite number of update signals between the integrated tools. The CASE tool and Knight *collaborate* on a common artifact, namely the model that the developers are designing. The used component technology and the interfaces of the tools decide the *communication* protocol; the Observers influence the communication pattern of the integration.

There is currently no standard programming interface that has been adopted by CASE tools. Rational Rose and WithClass have completely different COM interfaces. We argued above, through the use scenarios, that there is a need for both asynchronous and synchronous cooperation between tools. Hence, there is also a need for a standardised programming interface for operating on the data that has already been standardised in the UML. Just as a standardised data interchange format has been created from the UML metamodel, we believe that a standardised programming interface for operating on data conforming to this metamodel should be created. In the case of the UML Metamodel, there should be methods for, e.g., creating and deleting classes, and adding and deleting attributes. Of course, there should also be methods for attaching arbitrary information these elements, just like it is possible to add arbitrary data in the extension elements of the XMI format. A proposal for a standard interface is provided as part of the UML 1.3 specification [23], in the form of a CORBA IDL description. However, we are aware of no tools that implement this specification and thus have no experiences as to whether it is actually usable. While having a standard programming interface for operating on the data itself is important, it is not sufficient. There is still a need for tool-specific functionality, e.g., opening and closing another tool or scrolling a diagram. The situation is thus the same as for common interchange formats: only part of the interaction will be standardised.

Other integration types. In this paper, we have mainly considered the two categories "asynchronous integration with shared data" and "synchronous integration with separate data" of Table 1. One of the other categories, “synchronous integration with shared data”, covers components working together synchronously on the same data. This model is also very common in existing tools, e.g., in integrated development environments with editors, compilers, debuggers etc. working on the same data. It would be easy to modify the Knight tool to work on, e.g., a Rational Rose data structure, instead of having its own data, which, conceptually, is just a copy of the Rational Rose data. However, by having its own data,

Knight can run as a stand-alone tool without Rational Rose. This situation is analogous to exchanging data between Knight and Rational Rose (section 4.1): having to run both tools synchronously is impractical (sometimes even impossible) compared to exchanging the data in a file. Also, if Knight uses Rational Rose's data format, it cannot run with WithClass or any other tool. Of course, having standard programming interfaces would remedy the problem.

The final category in Table 1 is "asynchronous integration with separate data". We have not considered this category in detail in our integration work, but just to complete the table we give an example: a diagram versioning system with a central repository. Each client makes changes to its own copy of the shared diagram. Once in a while, a client commits the changes to the central repository, and the changes are merged into the shared diagram.

5. Conclusion

As an example of tool integration, we described the integration between the Knight tool and existing CASE tools. Two different types of integration were investigated and implemented: asynchronous integration with shared data implemented using a standard interchange format, and synchronous integration with separate data implemented using component technology. The two integration approaches complement each other: asynchronous integration with shared data via a standard interchange format is simple, robust, and lightweight. Synchronous integration with separate data via component technology supports a tight integration between tools.

Several problems and issues to be considered were highlighted: even though the asynchronous integration with shared data was realised using a standard interchange format, only part of the necessary information to be interchanged was standardised. For the synchronous integration with separate data, the situation was even worse. No standard way of defining interfaces for components dealing with UML has been implemented. Thus, even though both integration efforts have been successful, the way to flexible, interoperable, and integratable tools goes through even further standardisation efforts. Irrespective of standardisation, however, tools will need to store arbitrary information that must be handled in specific ways. This caused problems for both types of integration considered.

Nevertheless, we were successful in integrating the tools. We believe that our integration can be used not only as a source of inspiration, but also that the concrete design and the software architectures described are of general applicability, and that they can help developers in integrating other tools.

6. Acknowledgements

The work described in this paper was carried out at the Centre for Object Technology (`http://www.cit.dk/COT`), which has been partially funded by the Danish National Centre for IT Research (CIT).

7. References

1. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture. A System of Patterns,* Wiley.
2. Chen, M. & Norman, R.J. (1992). A Framework for Integrated CASE. *IEEE Software,* 9(2):18-22, March.
3. Damm, C.H., Hansen, K.M., & Thomsen, M. (2000). Tool Support for Cooperative Object-Oriented Design: Gesture Based Modeling on an Electronic Whiteboard. In *Proceedings of Computer Human Interaction (CHI'2000)*, The Hague, The Netherlands.

4. Damm, C.H., Hansen, K.M., Thomsen, M., & Tyrsted, M. (2000) Creative Object-Oriented Modelling: Support for Intuition, Flexibility, and Collaboration in CASE Tools. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'2000)*, Sophia Antipolis and Cannes, France.
5. Demeyer, S., Ducasse, S., & Tichelaar, S. (1999). Why FAMIX and not UML? UML Shortcomings for Coping with Round-trip Engineering. In *Proceedings of <<UML'99>>,* Fort Collins, CO, USA, October 28-30.
6. Demeyer, S., Meijler, T.D., Nierstrasz, O., & Steyaert, P. (1997). Design Guidelines for Tailorable Frameworks, *Communications of the ACM*, pp. 60-64. October, vol. 40, no.10.
7. Ellis, C. A., Gibbs, S. J., & Rein, G. L. (1991). Groupware: Some Issues and Experiences. *Communications of the ACM*, pp. 38-58, 34(1).
8. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Longman.
9. Haines, G., Carney, D., & Foreman, J. (1997). *Component-Based Software Development / COTS Integration.* SRI Software Technology Review.
10. Henning, M., Vinoski, S. *Advanced CORBA Programming with C++.* Addison Wesley Longman, 1999.
11. Jarzabek, S. & Huang, R. (1998) The Case for User-Centered CASE Tools. *Communications of the ACM*, 41 (8).
12. Kobryn, C. (1999) UML 2001: A Standardization Odyssey. *Communications of the ACM,* pp. 29-37, 42(10).
13. Kurtenbach, G. (1993). *The Design and Evaluation of Marking Menus.* Unpublished Ph.D. Thesis, University of Toronto.
14. 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.
15. MOF Revision Task Force. (1999) *Meta Object Facility Specification v. 1.3*. Document ad/99-06-05, Object Management Group.
16. Ousterhout, J. (1990). Tcl: An Embeddable Command Language, in *Proceedings of the Winter 1990 USENIX Conference*, January 22--26, Washington, DC, USA.
17. Reiss, S.P. (1990) Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, July.
18. Robbins, J.E. & Redmiles, D.F. (2000). Cognitive support, UML adherence, and XMI interchange in Argo/UML. In *Information and Software Technology*, 42(2), 79-89, 2000
19. Rogerson, D. (1997). *Inside COM: Microsoft's Component Object Model*. Microsoft Press.
20. Rumbaugh, J., Jacobson, I., & Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison Wesley.
21. Shaw, M. (1996) Some Patterns for Software Architectures. In Vlissides, Coplien, Kerth (Eds.), *Patterns Languages of Program Design*, volume 2, Addison Wesley.
22. Tichelaar, S., Cruz, J.C., & Demeyer, S. (2000). Design Guidelines for Coordination Components. To appear in *Proceedings of ACM SAC 2000 - Track on Coordination*, ACM Press, March.
23. UML Revision Task Force (1999). *OMG UML v. 1.3: Revisions and Recommendations*. Document ad/99-06-10, Object Management Group, June.
24. W3C (1998). *Extensible Markup Language (XML) 1.0.* W3C Recommendation REC-xml-19980210, 10-Feb-98. Available online at `http://www.w3.org/TR/1998/REC-xml-19980210`.
25. XMI Partners (1998). *XML Metadata Interchange (XMI),* OMG Document ad/98-10-05, October 20. Available online at `http://www.omg.org/cgi-bin/doc?ad/98-10-05`.
26. XMI Partners (1999). *XML Metadata Interchange (XMI) 1.1 RTF Final Report.* OMG Document ad/99-10-04, October 20. Available online at `http://www.omg.org/cgi-bin/doc?ad/99-10-04`.