Journal of Universal Computer Science, vol. 14, no. 19 (2008), 3265-3278
submitted: 23/7/08, accepted: 29/10/08, appeared: 1/11/08 © J.UCS

Integrating Dialog Modeling and Domain Modeling – the Case of Diamodl and the Eclipse Modeling Framework

Hallvard Trætteberg
(Norwegian University of Science and Technology, Norway
hal@idi.ntnu.no)

Abstract: For most applications, data-intensive applications in particular, dialog modeling makes little sense without a domain model. Since domain models usually are developed and used outside the dialog modeling activity, it is better to integrate dialog modeling languages with existing domain modeling languages and tools, than inventing your own. This paper describes how the Diamodl language, editor and runtime have been integrated with the Eclipse Modeling Framework.

Keywords: Model-based user interface design, dialog modeling, domain modeling, Eclipse Modeling Framework.
Categories: H.1.1, H.5.2

1 Introduction

Domain modeling is one of the most fundamental software engineering activities. Besides being a tool for reaching a common understanding of the domain, a domain model may be utilized for engineering the actual system, both as a guidance for design and more directly as input to code and schema generators. There exist many languages that are well suited for this, and object-oriented class diagrams, as found in the Unified Modeling Language (UML), are very popular and well supported by modern tools.

A dialog model is an abstract description of the structure and behavior of a user interface, and in particular what information that each part mediates to the user as output and from the user as input. Hence, the domain model plays an important part of a dialog model, as the information that the user senses and manipulates by means of the user interface, should be part it. This does not mean that the domain modeling activity must fully precede dialog modeling. E.g. in a user-centered design process, prototypes are often used to elicit domain knowledge and requirements. If models are used for rapid prototyping, the domain model may result from or be refined by the dialog modeling process.

Since domain models are so important for dialog models, a dialog modeling language needs to include domain modeling concepts, like entity or class, attributes and associations, operations etc. This does not mean you need to include a full domain modeling language in your dialog modeling language, but you must be able to reference, operate on and utilize elements in existing models. At the language level this means identifying every concept and construct where the domain model is relevant and where such references may be necessary. At the tool level this means defining the role and scope of your own tools and existing ones, and their level of integration and interoperation. At the process level this means identifying good practices for reaching specific goals.

This paper details how the Eclipse Modeling Framework (EMF) [EMF] has been integrated into our Diamodl [Trætteberg, 03][Trætteberg, 07] dialog modeling language, at the language level, tool level and process level. Section 2 gives an overview of related work. Section 3 and 4 briefly describe the Diamodl language and tool architecture, respectively, while section presents EMF. Section 5 explains how EMF concepts are integrated in the Diamodl language and how the EMF framework is utilized in the Diamodl tools. Section 6 and 7 outlines how the developer utilizes both EMF and Diamodl tools for developing and executing models, before we conclude in section 8.

2 Related work

The topic of domain modeling has received some, but surprisingly little attention in the model-based user interface design (MB-UID) community in recent years. As the focus has shifted towards task-based design, domain modeling and data management get less focus than before. [Paternò, 00] lists domain models as one of several important abstractions, but it does not actually include a single domain model, although it does discuss UML, including class diagrams. Domain objects are mentioned as something operated upon by tasks, but little is said about how the structure of tasks relate to the structure of domain models, e.g. specialization of task along inheritance relations, navigation along associations, etc.

CTT, the most widely used task modeling notation, does not include objects in task diagrams itself, although it includes operators for data-based enablement. Several tools exist for working with CTT models, e.g. CTTE [Mori, 02] and Dialog-Graph-Editor [Reichart, 07], and although they both are able to generate running prototypes, they do not seems to be integrated with domain modeling languages or tools commonly used by industry.

Just-UI, with its heritage from CASE method, acknowledges domain models by including a conceptual modeling stage using the object-oriented OO-Method, to obtain a conceptual schema of an Information System [Molina, 02]. Code generation techniques are used for building a complete system, and it cannot be considered a rapid prototyping tool.

Methods with roots from object-oriented design and software engineering, are perhaps the most data-centric. OVID [Roberts, 98] is a method that combines object-oriented analysis and user interface design, to bridge the gap to software engineering. Hence, it has potential for integrating with domain modeling tools, but unfortunately, there is no tool support for OVID.

CanonSketch [Campos, 04] is a tool for user-centered prototyping of user interfaces, where sketches of interfaces are evolved through canonical prototypes to executable HTML-based prototypes. In some ways, it is similar to the Diamodl approach presented in [Trætteberg, 07], in the focus on concrete representations and prototyping. However, their focus is on usability of developer-centric tools, while Diamodl is what they would call formalism-centric. CanonSketch is based on the Wisdom method [Nunes, 01] and hence supports UML, both for describing the logical structure of the user interface, the domain model and the link between them.

However, CanonSketch does not seem to have support for combining the prototypes with rich example data, based on the domain model, e.g. to aid user-centered evaluation.

UsiXML [Limbourg, 04] is a family of XML-based languages, with a rich set of sub-languages and accompanying tools. UsiXML includes concepts (and tags) for task modeling, abstract and concrete user interface modeling and domain modeling [UsiXML]. The domain model supports object-oriented modeling with classes, inheritance, attributes, associations and methods. From the documentation on the web site it is difficult, however, to understand how the domain model is handled by the many editors, generators and interpreters, e.g. if code is implemented for representing the domain objects, like EMF does. The domain model has the concept of object and attribute, there seems to be insufficient support for using rich example data in the generator and protyping tools.

3 Diamodl and domain modeling

The Diamodl dialog modeling language is a hybrid language based on dataflow and state logic. The dataflow part of the language includes concepts for storing, transforming, computing and transmitting data, and is by nature data-intensive. The main data-oriented language concepts are shown in Table 1.

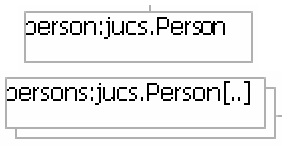
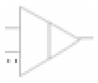
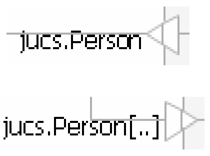
Concept	**Notation**	**Description**
Variable		A *variable* holds a single value or object (top) or a set of objects (bottom) of a specified type.
Computati on		A *computation* contains a function that takes a set of arguments and computes a value. The value is recomputed whenever one of the arguments change.
Gate		A gates is a special computation attached to the edge of an interactors. The direction of the gate determines if it represents output or input. Often the function is just used for type checking.
Connectio n	Line connecting the flow graph nodes.	A *connection* controls the flow of values from a node to another. A connection may contain a unary function that transforms the value flowing along the connection.

Table 1: Data-oriented Diamodl concepts

Figure 1 shows a simple model fragment, with two variables, several connections, a computation and two gates attached to an interactor (labeled rectangle). This particular fragment models how a person (object of type jucs.Person) may be selected from a set of persons by means of a list widget. The value of the variable named "persons" flows into the gate pointing into the interactor. This gate is an output-receive gate, meaning that the interactor should make the value available for sensing by the end-user. In the case of the fragment in Figure 1, a list widget has been chosen, as it presents a set of values to the end-user and supports selection. When the gate receives the set of persons, their names are extracted and used as the list widget's new content. If the interactor supports input of (new) values, as in this case, these values will flow out of the input-send gate, pointing out of the interactor. In the case of the fragment in Figure 1 the selected item will flow through some internal logic and out of the lower gate and into the variable named 'person'.

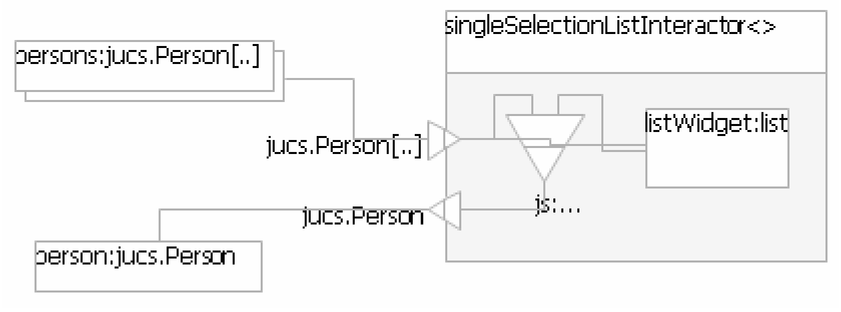


Figure 1: Diamodl model fragment

All these language elements are related to the domain model, e.g. variables and gates are typed with class names, connections may access attributes to compute related values, and all of them (including the connections) operate on actual domain data.

While not shown in this model fragment, the Diamodl language supports states and transitions for controlling when various parts of the user interface are active. At first glance, these concepts are unrelated to the domain model. However, transitions are triggered by events and guarded by conditions, and they may refer to the domain data flowing into and out of variables and computations. In addition, transitions include actions that are executing when they are triggered, and states include actions that are executed upon entry and exit. These actions may operate on the domain data accessible in the context of the triggering model element.

4 Diamodl system architecture

Prototyping with Diamodl is supported by an editor and a runtime. The editor is based on GMF, with separate models for the language, notation and mapping between them. Custom actions are used for inserting model fragments, performing specialized editing and for launching a runtime and executing the model. The behavior of the running prototype is animated in the editor to make it easier to understand the relationship between concrete and abstract structure and behavior. It is also possible to capture an event trace and inspect the execution in greater detail.

The Diamodl runtime is based on several open source libraries. EMF is used for managing both domain models and sample instance data. The dataflow behavior is based on JFace Data Binding [JFaceDataBinding] and includes extensions for binding EMF data to SWT widgets. Statechart logic is implemented by means of the Apache SCXML engine [SCXML], while GUI execution utilizes an XML format and renderer for SWT named XSWT [XSWT]. Finally, Javascript support is implemented by the Mozilla Javascript library [MozJavascript] (formerly called Rhino). All these libraries have been adapted and integrated to work well with Diamodl and each other, as illustrated in Figure 2.

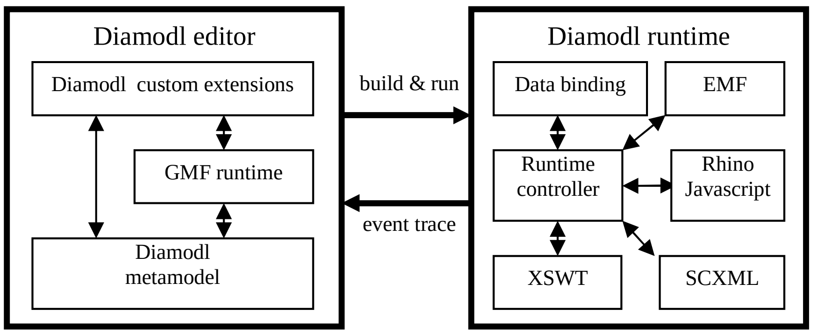


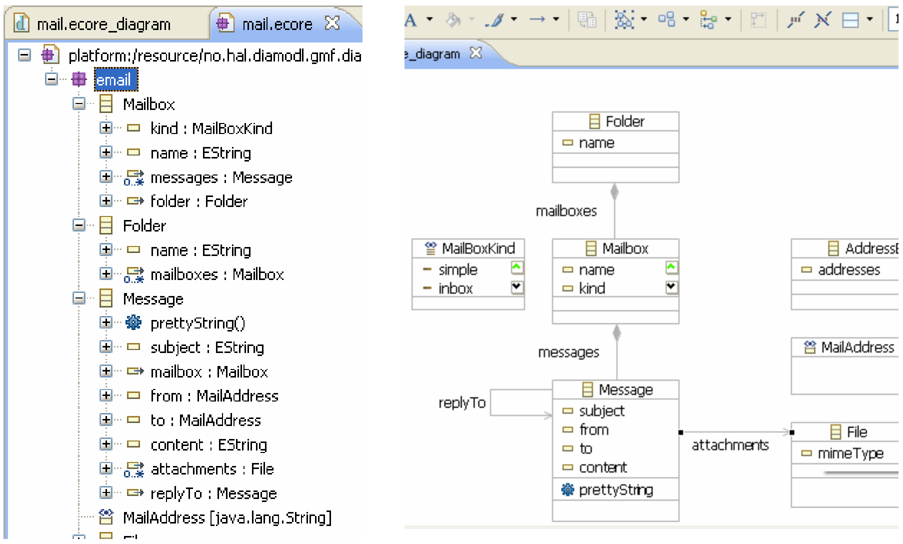
Figure 2: Diamodl editor and runtime architecture

5 Eclipse Modeling Framework (EMF) concepts and tools

EMF is a comprehensive Java-based framework and set of tools for modeling and managing domain data. Its core consists of a modeling language called Ecore and a compact runtime for managing models and instance data. Ecore supports a small subset of UML's class diagrams, corresponding to the eMOF core of Object Management Group's MetaObject Facility [OMGMOF], that is well suited for both in-memory manipulation and code generation. Since Ecore is its own meta-model, a fairly small set of concepts and a correspondingly small API may be used for understanding and managing Ecore objects.

An Ecore model describes a set of *classes* with *attributes*, *references* and *operations*. Generalization, multiple inheritance, interfaces, enumerations and Java data are supported. In the typical case, a developer will generate a complete Java implementation of the model, and use the generated code in further development, e.g. by filling in the body of operation stubs. Two standard languages, Java and OCL, are supported for specifying an operation's implementation. Both kinds rely on *annotations* on the operation that are recognized by the code generator, and appropriate code templates. Although a special, generated object factory is used to create the actual instances, the instances themselves will look & feel like ordinary Java objects and the application programmer will mostly use them as such.

EMF also supports a dynamic, reflective, meta-object-based API for managing instances, without any code generation. This API is typically used by generic tools that are independent of a specific Ecore model. In this case, the application typically loads one or more Ecore models and uses the contained meta-objects for creating instances and accessing attributes and references.



```
mail.ecore_diagram   mail.ecore
platform:/resource/no.hal.diamodl.gmf.dia
  email
    Mailbox
      kind : MailBoxKind
      name : EString
      messages : Message
      folder : Folder
    Folder
      name : EString
      mailboxes : Mailbox
    Message
      prettyString()
      subject : EString
      mailbox : Mailbox
      from : MailAddress
      to : MailAddress
      content : EString
      attachments : File
      replyTo : Message
    MailAddress [java.lang.String]
```

Figure 3: Tree-based (left) and graphical (right) Ecore editors

Whether you use the generated or reflective API, a comprehensive set of features is provided by the framework core, including change notification and recording, automatic management of inverse references and XML-serialization. Several other Eclipse projects utilize and contribute EMF technology. The EMF Tools (EMFT) project contributes support for validation (e.g. using OCL), querying and transactions.

EMF provides Eclipse-based views, editors and wizards for working with both data and models. There are wizards that let you create an Ecore model from scratch, or built from Java code or an XML schema. The tree-based Ecore editor provides menu commands for creating model element according to context, a property sheet for editing details and drag'n drop for moving elements. There also exists a graphical Ecore editor contributed by the Ecore Tools project, based on the Graphical Modeling Framework (GMF) project. A screenshot of both the tree-based and the graphical editor is shown in Figure 3.

Given an Ecore model it is possible to create and edit a data file containing instance data in accordance with the model. For instance, if the model contains the classes Library, Author and Book, the data file may contain instances representing a specific Library containing a set of specific Books by specific Authors. The data editor is purely reflective, i.e. it does not rely on generated code, only on the model. The editor makes sure only valid instance structures can be created, based on the rules defined in the model and implemented by custom validation code.

Figure 4 shows how the editor ensures that only a message may be created as a child of a Mailbox. Although the editor is simple and tree-based, it works well for creating example data for a domain.

EMF has extensive support for XML, in two ways. First, all instance graphs, including Ecore models are serializable as XML. Second, any XML schema may translated to an Ecore model. Later, when EMF serializes instance graphs for this model, it will make sure the XML is compliant with the original XML schema.

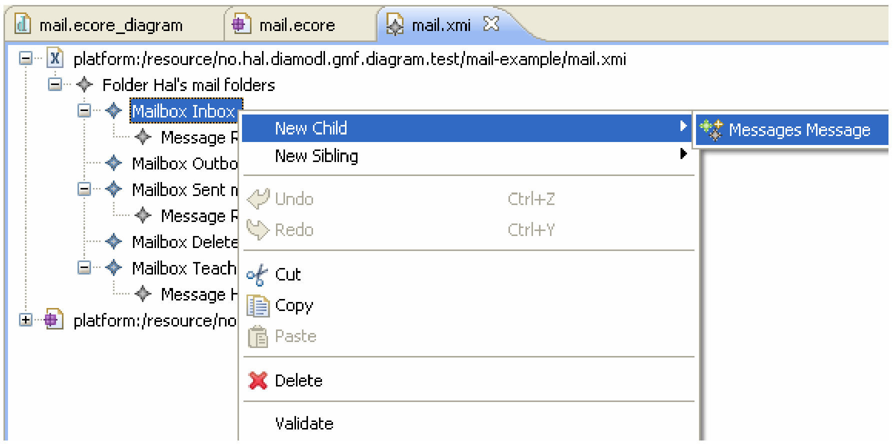


Figure 4: Tree-based data editor

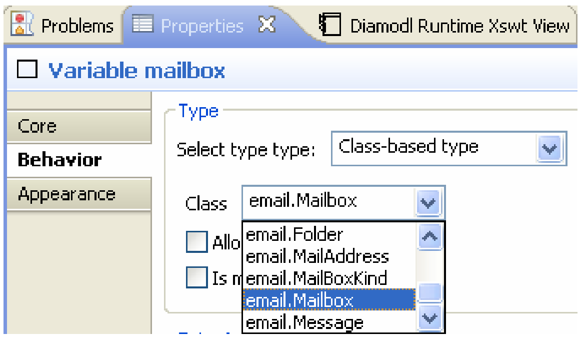
EMF-based editors utilize a standard three-tier application architecture with persistence based on XML, a model layer with centralized command execution and support for undo/redo, and the user interface as the view and controller. The EMF API supports extending all relevant aspects of this architecture and many projects provide more advanced features, like support for database-based persistence (Teneo), command execution with validation and transaction support (EMFT) and multiple clients sharing a server-based model (CDO). Custom editors, like the graphical ones build using GMF, will typically extend relevant layers themselves, and/or utilize other's extensions, like those from the mentioned projects. Diamodl replaces the user interface with two layers, a dialog layer and a surface layer, and builds on the other layers, without constraining them.

6 Utilizing EMF in the Diamodl language and runtime

Types and *functions* are the most important diamodl modeling concepts where elements from the domain model need to be referenced.

A type specifies a limited range of values that a variable may hold, that a function may take as argument or return, or that may flow along a connection. A type is either a basic type like `Integer`, `Boolean` or `String`, a user-defined class name or an array/list of one of the previous kinds. Diamodl does not support the definition of new classes; instead it relies on Java's classpath, for looking up Java classes, or Ecore models that are associated with the Diamodl model, for looking up Ecore classes. I.e. to introduce domain specific data in a Diamodl model, you must either add the necessary Java classes to your project or model the data using EMF's tools and associate the resulting Ecore model with your Diamodl model. As mentioned above, defining new Ecore classes may be done using the editors provided by the EMF and GMF projects. Once the Ecore model is associated with the Diamodl diagram, Ecore class names may be used as the type of variables by selecting from a menu, as shown in Figure 5.

Functions are used for defining the behavior of computations. Unary functions may in addition be attached to connections, to transform values as they pass through. There are many kinds of functions, and there are several kinds that utilize the domain model in its definition.



Variable mailbox

Type

Select type type: Class-based type

Class email.Mailbox

email.Folder
email.MailAddress
email.MailBoxKind
email.Mailbox
email.Message

Figure 5: *Selecting an Ecore class as the type of a variable.*

So-called *property* functions may refer to attributes and associations defined in the domain model. Such functions take an instance of the class containing the attribute or association as the only argument and may be used on both computations and connections. When executed, the property function finds the (Ecore) class of the argument and returns the value of the specified attribute or association. The computation named `messages` in Figure 1 is an example of a property function. It computes the set of messages in a mailbox, where messages is an association connecting the Mailbox and Messages classes in the model shown in Figure 3.

A second kind of function, the *script* function, uses Javascript to operate on the domain data accessible through the function's arguments. We use the Mozilla Javascript library, which is tightly integrated with Java. E.g. it is possible to refer to any Java class and getters and setters are accessed using the Javascript property mechanism. To better support Ecore-based data, we have extended the Mozilla Javascript library to support Ecore instances. When a Javascript property of an Ecore instance (EObject) is read or set, we use EMF's dynamic meta-object-based API to access the corresponding attribute or association. This means that we may navigate through any instance graph with both ordinary Java objects and Ecore instances using Javascript's property access syntax.

As mentioned, Ecore supports specifying the implementation of operations using annotations. Unfortunately, since the support relies on code generation and compilation, it is less helpful in our prototyping tool. Therefore, we have added support for annotating the operation with Javascript, thus avoiding code generation and compilation.

In Javascript, a method is a property with an anonymous function as its value. We have augmented the property lookup mechanism to look for our custom annotation. If the property currently has no value and the custom annotation contains source code of a function, an anonymous function object is created on-the-fly, installed as the property value and returned. E.g. the `prettyString` operation defined by the `Message` class and shown in the class diagram in Figure 3, is implemented by the following Javascript: `$1.from + ":" + $1.subject;` (The `$1` symbol refers the implicit object argument, while `$n` for higher n's refers to the explicit arguments.) By supporting a scripting language like Javascript, more of the domain's meaning may be captured in the model and used it in our prototyping tool, without the need for code generation and a costly build operation.

Sometimes there is a need for referencing classes and other model elements in the domain model in Javascript code. This is necessary for testing whether some object is an instance of a specific Ecore class or for creating new instances of a specific Ecore class. To support this, all the Ecore packages associated with the Diamod model are defined as global variables in Javascript's top-level scope. E.g. in the model in Figure 1, Javascript code will be able to refer to the Ecore package named `email` just by using its name. By using the `getEClassifier()`-method from the reflective Ecore-API it's then possible to locate a specific class by name, e.g. `email.getEClassifier("Message")` will return the Ecore class named `Message`. Since looking up contained objects by name is fairly common, we have introduced a shortcut that lets you read it as a property. E.g. the expression `email.Message` will iterate across all objects contained by the email variable's value and return the one with a `name` attribute with the value "`Message`". This mechanism works for all Ecore instances, so any domain model using containment associations and `name`-attributes will support this shortcut.

The behavior of Diamodl is partly dataflow and partly Statecharts. The variables, computations, gates and connections define a network through which data is propagated whenever some change occur. E.g. when a variable changes, the value is propagated through the outgoing connections. Similarly, when one of the input values of a computation changes, a new output value is (re)computed and propagated through the network.

In addition to reacting to changes of values, the Diamodl runtime also support reacting to changes of value *properties*, as this is necessary to ensure that dependent values are correctly updated. E.g. a form field showing some property P of variable O must be updated both when O changes and when O's P property changes. In the Diamodl runtime, this is implemented by means of the JFace Data Binding framework and its *observable* values. Out of the box, this framework supports Java's standard way of listening to property changes. However, we have added support for Ecore's notification mechanism, to allow Ecore instances to also fully participate in dataflow. This makes it possible to have a variable typed with an Ecore class and ensure that property changes are reacted upon. E.g. in the example in Figure 1, this may be used to ensure that the listbox presenting the messages contained by the mailbox variable is correctly updated when a message is added or removed.

The network of connected variables and computations and the various function types represents explicit declarations of relations between data in the user interface, and this is the basis for configuring the data bindings. The Javascript-based functions may, however, hide such relationships. Consider a list of mailboxes, which is populated by running Javascript code that extracts their names, and a separate dialog for renaming the mailboxes. To make sure the list is updated when a mailbox is renamed, we must somehow notice that this change affects the Javascript function. Therefore, the Javascript engine has been extended to record dependencies and report those to the data bindings engine. Hence, when the list of names are extracted from the mailbox objects, the data binding engine knows that subsequent changes to the 'name' property of these mailboxes must trigger re-computation of the Javacript function. In general, after executing a Javascript function, appropriate listeners are attached to Ecore objects based on the dependencies, and when notifications are received, dependent values are re-computed.

Conditions and actions are related to the Statechart "nature" of Diamodl. As mentioned above, they may both be attached to transitions and actions may also be attached to states. Conditions are evaluated in the context of some event, to guard the triggering of a transition after the appropriate event has been received. The event may include an event object which the condition may refer to. E.g. if we want something special to happen when a message with a certain property is selected, the event will be that an object flows out of the appropriate gate, while the condition will check the property. Again, the domain model becomes central, and as for functions, Javascript support has been introduced as a generic solution. I.e. conditions may be defined by means of Javascript expressions. Similarly, actions may be defined by Javascript statements, that are executed in the context of the triggered model element, whether transition or state.

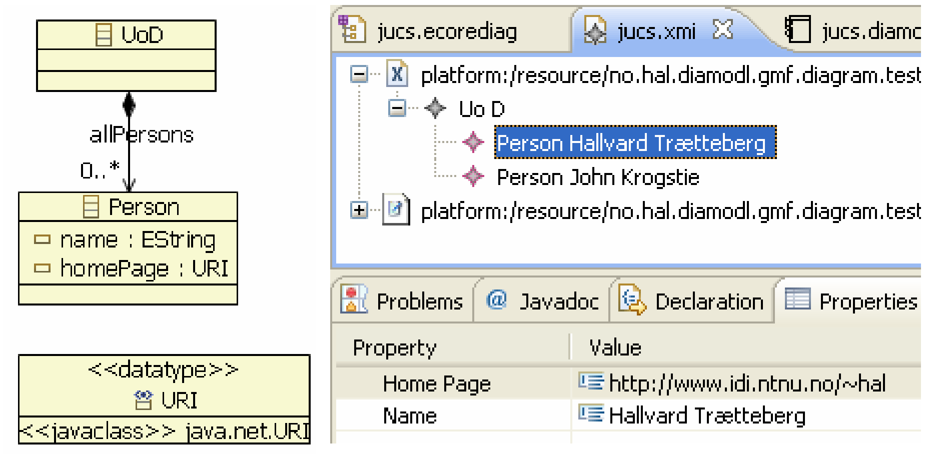
7 Using the EMF and Diamodl tools together

In this section we will outline how the EMF and Diamodl tools are used together in practice, by means of an example.

It is often a matter of taste whether domain modeling is done before, during or after dialog modeling. Fairly often, we model user interfaces for existing systems, and in such cases it is natural to develop the domain model first. EMF provides both tree-based and diagram-based editing of Ecore models, as shown in Figure 3. In the initial development of the model, we prefer the diagram editor, but we find ourselves using the tree-based for details and annotations as it is more responsive and has complete coverage of the Ecore language.

The example domain model is shown left in Fig. 6 and contains the two *classes* `Person` and `UoD` and the `URI` *datatype*. The Person class includes `name` and `homePage` *attributes*, while the `UoD` class has a *containment reference* to the `Person` class named `allPersons`. In Ecore, all objects in a *resource*, e.g. a data file, must be contained in a single root object, and therefore we include the `UoD` (for Universe of Discourse) class. Hence, if we have a reference to the UoD object, we may reach all other objects in our "universe". The `URI` datatype lets us include objects of ordinary Java classes as types in our model. In this example, it is used as the type of the `homePage` attribute.

Once a domain model is available, we can start referring to domain concepts in the Diamodl model, e.g. define variables with type information and computations that access attributes and associations. In the example model shown in Fig. 7, we have three variables named `uod`, `persons` and `person`, with the types `UoD`, `Person[..]` and `Person` (`[..]` indicates a multiplicity of more than one). If a type name is missing from the domain model, the Diamodl editor will insert so-called problem markers, which is an Eclipse mechanism for indicating warnings or errors with small icons in top of the faulty object. The Diamodl editor includes toolbar buttons for opening the domain model in the tree-based or graphical editor, so it is easy to add missing classes to the domain model.



platform:/resource/no.hal.diamodl.gmf.diagram.test
Uo D
Person Hallvard Trætteberg
Person John Krogstie
platform:/resource/no.hal.diamodl.gmf.diagram.test

Property	Value
Home Page	http://www.idi.ntnu.no/~hal
Name	Hallvard Trætteberg

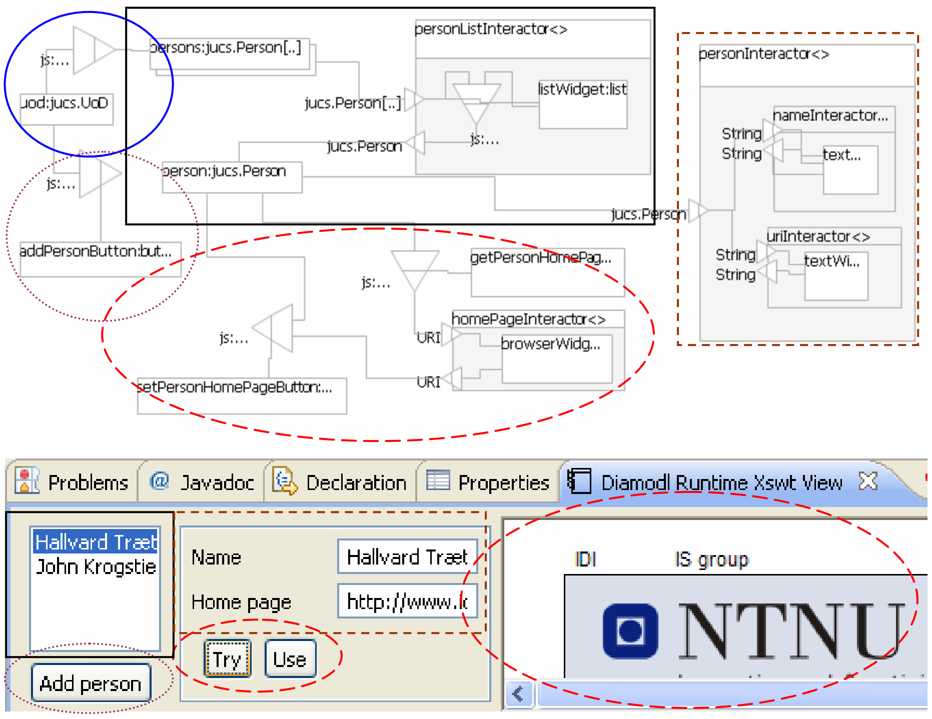
Figure 6: Example domain model and sample data

A qualitative evaluation is possible without actual domain data, but little of the behavior may be tested without actual data to operate on. The Ecore editor includes an action for instantiating the model in a data file. A complete graph of Ecore instances may then be created using the reflective data editor, as shown right in Fig. 7. Contained by the `UoD` object, we find two `Person` objects, and we can see that one of them have 'Hallvard Trætteberg' as its name and 'http://www.idi.ntnu.no/~hal' as the homePage. The same Person instances appear in the list widget at the far left in the prototype in Fig. 7. Since the XML serialization format is based on the open XMI standard, it is straight-forward to export data from existing sources, using appropriate queries and transformations. instead of creating sample data manually.

The example model and corresponding GUI prototype is shown in Figure 7, and contains the following functionality:

- The solid circle contains a variable named `uod` holding the root `UoD` object, in which all other objects are contained (directly or indirectly). All the `Person` objects are extracted by following UoD’s `allPersons` association and sent to the `persons` variable.

- The solid rectangle contains the list widget that is used for viewing the Person objects. It includes Javascript for extracting the `Person` instances’ names. The `person` variable holds the selected `Person`.
- The `person` variable is connected to `personInteractor` inside the dashed rectangle, which models the fields used for viewing and editing the `name` and `homePage` properties.
- The browser widget and “Try” and “Use” buttons are modeled inside the dashed oval. The “Try” button triggers a computation which sends the selected `Person`’s `homePage` property to the browser, while the “Use” button triggers a computation that sets the same `Person`’s `homePage` property to the `URI` currently shown by the browser.
- The “Add person” button is modeled inside the stippled circle and creates a new `Person` instance and adds it to the `UoD` object’s `allPersons` association.



Hallvard Træt
John Krogstie

Name: Hallvard Træt
Home page: http://www.i

Figure 7: The example model(above) and GUI prototype (below) with corresponding parts highlighted

The main dependencies are modeled by the connections, while some hidden dependencies are handled by the integration of Javascript, Ecore and the databinding engine. E.g. when a `Person`’s `name` property is edited, the list widget showing these properties is automatically updated. Similarly, when a new `Person` is created and added to the `UoD` object's `allPersons` association, the list content is updated.

Once the example data are in place, they may be loaded by the runtime and utilized by the model elements. E.g. each variable in the model has an enter action, that is executed when that part of the model is activated as a state. To initialize the variable with data from the data file, you will usually write a simple Javascript expression that navigates from the root instance to the desired data. A special action is provided for the common case where a variable is initialized to all the instances of a certain class.

To test a dialog, all the concrete user interface elements are assembled into an XSWT file and the diamodl runtime renders the GUI and executes the diamodl model. The initialization of the variables will trigger the data binding engine and the GUI prototype will become alive, as the user interface elements are populated with data. The end-user may then start interacting with the prototype and cause values to flow according to the modeled behavior. In the example prototype shown in Fig. 7, the user has selected a person from the list and pressed the "Try" button to view the selected person's home page. If the evaluation indicates that the functionality should be changed, the model may be edited and the GUI prototype rebuilt and run again.

The model may be executed and the GUI rendered in several contexts. In the simplest case, the GUI appears in a view below the diamodl editor. This is useful when running small examples and debugging. The GUI may instead be shown in an application frame, which may be resized and moved around like ordinary applications. A third variant is showing the widgets of the GUI inside the diamodl editor itself. This mode is relevant for visualizing the executing, but since the layout is based on the model's, it is not useful for end-user testing. Finally, the GUI may be executed as part of an editor application utilizing EMF's layered architecture. Although not yet completely implemented, this will allow full support for opening XMI documents, editing the contained data using commands, undoing and redoing these commands and saving the XMI documents.

8 Conclusion and further work

The current version of the Diamodl editor and runtime, is the third in a series of implementations and the first that has been fully integrated with Eclipse and EMF. The EMF integration allows us to utilize a plethora of modeling tools within the Eclipse ecosystem, since most of these also support EMF. Hopefully, this will make dialog modeling easier to combine with model-driven software engineering. The code is open source and available from the recently established diamodl project at sourceforge.net.

We are currently using Diamodl editor and runtime and the EMF tools in our master course TDT4250 on model-driven development of Information Systems. In this course, we combine business process modeling with BPMN, domain modeling with EMF and dialog modeling with Diamodl. Our experiences show that this combination works well from a modeling perspective. The next step is integrating Diamodl and EMF runtime systems with a BPEL engine, to enable Diamodl-based prototypes to participate in modern SOA-based process-oriented architectures. By building on Eclipse and EMF, we believe we have good foundation for this work.

References

[Campos, 04] Campos, P., Nunes, N. CanonSketch: a User-Centered Tool for Canonical Abstract Prototyping. Proceedings of the EHCI/DSV-IS'2004, International Conference on Engineering Human-Computer Interaction / International Workshop on Design, Specification and Verification of Interactive Systems, Hamburg, Germany, 2004.

[EMF] The Eclipse Modeling Framework home page: http://www.eclipse.org/modeling/emf/

[JFaceDataBinding] JFace Data Binding: http://wiki.eclipse.org/index.php/JFace_Data_Binding

[Limbourg, 04] Limbourg, Q., Vanderdonckt, J., UsiXML: A User Interface Description Language Supporting Multiple Levels of Independence, in Matera, M., Comai, S. (Eds.), Engineering Advanced Web Applications, Rinton Press, Paramus, 2004, pp. 325-338.

[Molina, 02] Molina, P.J., Meliá, S., Pastor, O. JUST-UI: A User Interface Specificaiton Model. Ch. Proceedings of the 4th International Conference on Computer-Aided Design of User Interfaces CADUI'2002, Kolski, J. & Vanderdonckt, J. (eds.), Valenciennes, France. Kluwer Academics Publisher, Dordrecht, 2002.

[Mori, 02] Mori, G., Paternò, F., Santoro, C.. CTTE: Support for Developing and Analyzing Task Models for Interactive System Design, IEEE Transactions on Software Engineering (August 2002, pp.797-813).

[MozJS] Mozilla Javascript library: http://www.mozilla.org/rhino/

[Nunes, 01] Nunes, N. J., Cunha, J. F. WISDOM: Whitewater Interactive System Development with Object Models, in Mark van Harmelen (ed.), Object-oriented User Interface Design, Addison-Wesley, Object Technology Series, 2001.

[OMGMOF] Object Management Group MetaObject Facility: http://www.omg.org/mof/

[Paternò, 00] Paternò, F. Model-based Design and Evaluation of Interactive Applications. Series of Applied Computing, Springer-Verlag London, 2000.

[Reichart, 07] Reichart, D. Presented at the Tamodia 2007, Toulouse, France, demo session: http://liihs.irit.fr/tamodia2007/index.php?content=Demos

[Roberts, 98] Roberts, D., Berry, D., Isensee, S., and Mullaly, J., Designing for the User with OVID: Bridging User Interface Design and Software Engineering, MacMillan, 1998.

[SCXML] Apache SCXML engine: http://commons.apache.org/scxml/

[Trætteberg, 03] Trætteberg, H. Dialog modelling with interactors and UML Statecharts - a hybrid approach. Design, Specification and Verification of Interactive Systems. Funchall, Madeira, June 2003.

[Trætteberg, 07] Trætteberg. H. A Hybrid Tool for User Interface Modelling and Prototyping. In Calvary, G., Pribeanu, C., Santucci, G., Vanderdonckt, J. (eds.): Proceedings of the Sixth International Conference on Computer-Aided Design of User Interfaces CADUI'06. (6-8 June 2006, Bucharest, Romania), Chapter 18, Springer-Verlag, Berlin, 2007.

[UsiXML] UsiXML language documentation:
http://www.usixml.org/index.php?mod=pages&id=6

[XSWT] XSWT: http://sourceforge.net/projects/xswt (a bit outdated)