Multi-Language Support in TouchCORE

Maximilian Schiedermeier, Bowen Li, Ryan Languay,
Greta Freitag, Qiutan Wu, Jörg Kienzle
School of Computer Science
McGill University
Montreal, Canada
Maximilian.Schiedermeier,Bo.W.Li@mail.mcgill.ca
Ryan.Languay,Greta.Freitag,Qiutan.Wu@mail.mcgill.ca
Joerg.Kienzle@mcgill.ca

Hyacinth Ali, Ian Gauthier, Gunter Mussbacher
Department of
Electrical and Computer Engineering
McGill University
Montreal, Canada
Hyacinth.Ali@mail.mcgill.ca
Ian.Gauthier@mail.mcgill.ca
Gunter.Mussbacher@mcgill.ca

***Abstract*—TouchCORE is a multi-touch enabled modelling tool aimed at developing scalable and reusable models. Up to recently, TouchCORE was hard-coded to support only design modelling with class, sequence, and state diagrams. This demonstration paper presents the newest release of TouchCORE, TouchCORE 8, that features multi-language support through language plug-ins and perspectives. The paper reviews what a language designer needs to do to integrate a modelling language into the TouchCORE architecture, and how perspectives can be configured to ensure consistency between multiple models. The generic navigation facilities of TouchCORE are reviewed, and the generic split-view GUI support for displaying multiple models side-by-side together with inter-model consistency mappings is presented.**

***Index Terms*—multi-view modelling, model consistency, inter-model navigation, perspective**

I. Introduction

TouchCORE was initially called TouchRAM, and conceived as a multitouch-enabled tool for agile software design modelling aimed at developing scalable and reusable software design models [1]. TouchRAM exploits aspect-oriented model weaving techniques as defined by the *Reusable Aspect Models* (RAM) [2] approach to enable the designer to rapidly apply reusable design concerns within the design models of the software under development. RAM supports composition of class diagrams, sequence diagrams, and state diagrams.

TouchRAM was then upgraded to add support for *Concern-Oriented Reuse* (CORE) [3] (see Section II for more details on CORE) and consequently renamed TouchCORE [4]. To modularise a concern according to features, support for feature diagrams was added. To perform trade-off analysis when choosing between different variants of a concern, an impact model – a variant of goal models – can be defined for a concern that links features with high-level goals and qualities.

This demonstration paper presents the newest release of TouchCORE, TouchCORE 8, that provides support for multi-view modelling through language plug-ins and perspectives. Section II presents a brief background on CORE. Section III reviews what a language designer needs to do to integrate a modelling language into the TouchCORE 8 architecture. Section IV explains how perspectives can be configured to ensure consistency between multiple models. The generic navigation facilities of TouchCORE 8 are reviewed in Section V. The generic split view GUI support for displaying multiple models side by side together with inter-model consistency mappings is presented in Section VI, and the last section draws some conclusions and offers an outlook on next developments.

II. Background and Motivation

CORE [3] combines the ideas of Model-Driven Engineering (MDE), advanced modularization techniques (aspect-orientation), and software product lines (SPL) technology to streamline model reuse.

The main unit of reuse is called a *concern*. A concern groups together software artifacts (models and code) describing properties and behaviour related to any domain of interest to a software engineer, potentially at different levels of abstraction. A concern provides three interfaces [5]: The *variation interface* (VI) describes required design decisions and their impact on high-level system qualities, both explicitly expressed using feature and impact models in the concern specification. The *customization interface* (CI) allows the chosen variation to be adapted to a specific reuse context, while the *usage interface* (UI) defines how the functionality encapsulated by a concern may be used. CORE streamlines the reuse process by allowing a developer to *(a)* choose a desired variant (from the VI), *(b)* adapt the chosen models to the reuse context (with the CI), and then *(c)* use the resulting model (according to the UI).

The TouchCORE tool is implemented in Java on top of the *Eclipse Modelling Framework* (EMF). Java and Maven code generation is implemented using Acceleo. The Graphical User Interface is decoupled from the backend using the Model-View-Controller design. The controllers for manipulating models use command-based editing provided by EMF.edit in order to offer undo/redo functionality. The current model editors use the Multitouch for Java (MT4J) framework for rendering the models using OpenGL and dealing with mouse, keyboard and multitouch input.

Although previous versions of TouchCORE support different modelling languages for design, internally they are encoded in a *single* metamodel that integrates class, sequence, and state diagrams. Evolving such an integrated metamodel was very tedious. Furthermore, the integration of the languages

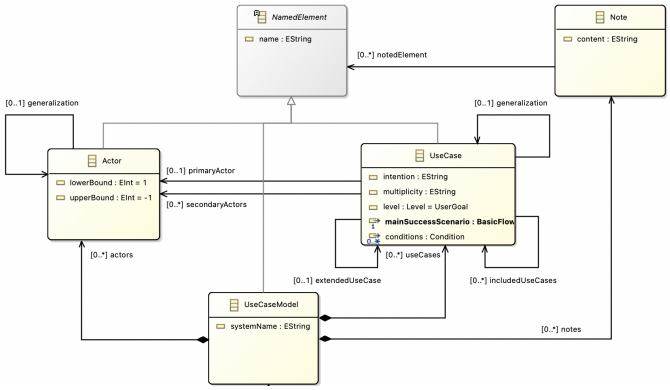

Fig. 1. Use Case Modelling Language Metamodel (Excerpt)

in the metamodel is specifically designed for these three diagrams for modelling software designs. As a result, old versions of TouchCORE are not well suited for creating, for example, domain models using the class diagram editor, where some class diagram elements such as operations are not supposed to be used. This is the main reason why TouchCORE was completely redesigned to support plug-in languages and perspectives as described in the remainder of the paper.

III. Language Plug-ins

This section describes the requirements for integrating a modelling language into TouchCORE 8.

A. Metamodel

Since TouchCORE 8 is based on EMF, modelling languages have to be defined using a metamodel. TouchCORE has no requirements on the language metamodel itself, i.e., any existing modelling language metamodel expressed in Ecore can be used. For example, Fig. 1 shows (parts of) the metamodel of a use case modelling language that now is part of TouchCORE 8, including *Actor* and *UseCase* metaclasses.

In order to support the complete independence of plug-in language metamodels, the additional information needed to support concern-oriented reuse of models is encoded in a separate CORE metamodel. For each model that is part of a concern, the CORE metamodel stores the information on which feature the model belongs to, and what model elements are part of the customization and usage interface of the model. Furthermore, when a model reuses another model, the CORE metamodel stores the concern reuse, the feature selections, and the model element customization mappings.

B. Language Actions

TouchCORE expects that a modelling language defines a set of *language actions* for creating and updating models. Language actions are at a higher level than the typical CRUD operations such as the ones automatically generated by EMF. A language action performs potentially several CRUD operations that together result in one coherent update on a model.

For example, for the use case modelling language, the language action `createActor` instantiates the metaclass *Actor and* inserts the newly created instance into the collection of actors of the *UseCaseModel* root object.

TouchCORE 8 defines a domain-specific language for specifying language plug-ins. Listing 1 shows the definition of the use case modelling language plug-in. Lines 2-7 provide data that TouchCORE needs to correctly register the metamodel of the language with EMF, and to load the Java controller classes implementing the language actions and the model weaver.

Our DSL also provides an optional means to explicitly expose language metaclasses so that they can later be used as part of consistency constraints enforced by a perspective (see Section IV). For the use case modelling language this is illustrated in Lines 8-12, which expose the *UseCaseModel* and *Actor* metaclasses. Lines 13 and onward demonstrate how to define language actions. In our example, a `createActor` language action is defined that creates instances of the metaclass *Actor*. The implementation of the action is provided in the Java controller class *UseCaseController*.

Listing 1. Defining the Use Case Language Plug-In with our Language DSL

```
1  language UseCaseLanguage {
2     nsURI "http://cs.mcgill.ca/sel/uc/1.0";
3     resourceFactory "usecasesResourceFactoryImpl";
4     adapterFactory
       "usecasesItemProviderAdapterFactory";
5     fileExtension uc;
6     rootControllerPackage "usecases.controller";
7     weaver "ca.mcgill.sel.usecases.language
       .weaver.UseCaseWeaver";
8     language elements {
9        languageElement UseCaseModel { }
10       languageElement Actor {
11           nestedElement NamedElement_Name;
12    }  }
13    actions {
14       create createActor(UseCaseModel owner,
          String name, ...) {
15          metaclass Actor;
16          classQualifiedName UseCaseController();
17 }  }  }
```

C. Graphical User Interface

TouchCORE 8 does not provide a generic GUI editor for plug-in languages. However, it offers implementations of a large selection of standard widgets and controls that are useful when implementing model editors, ranging from boxes to connecting lines to selector popups and menus. It also provides support for assisted and automated layout.

In theory it is not mandatory to implement a GUI editor in order to use a language in conjunction with TouchCORE. Existing standalone editors could also be used to create and edit models, and then the models can simply be associated with a feature of a concern to be managed by TouchCORE. However, in order to ensure inter-model consistency, TouchCORE must be able to invoke the language actions through some provided interface. Likewise, in order to support concern-oriented model reuse, a weaver needs to be implemented for the language (see the following subsection).

Figures 2, 3, and 4 show screenshots of GUI editors that are available for some of the language plug-ins.

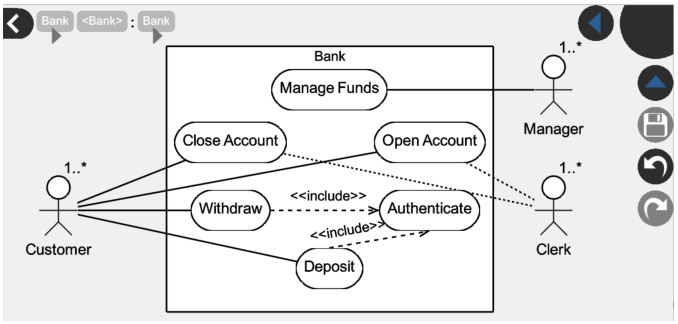

Fig. 2. Use Case Model of the Bank

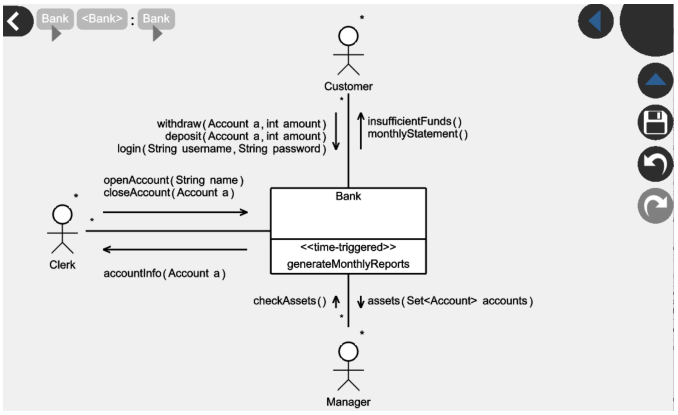

Fig. 3. Environment Model of the Bank

Figure 2 depicts a use case diagram for a *Bank* system with three kinds of actors, i.e., *Managers*, *Clerks*, and *Customers*, that pursue the goals of managing funds, opening and closing bank accounts, as well as withdrawing and depositing money.

Another language plug-in GUI for creating environment models is shown in Fig. 3. An environment model is a kind of UML communication diagram that can be used for specifying the system interface during requirements engineering. Our *Bank* environment model example shows again the three actors, and specifies the detailed signatures of the input and output messages that the Bank system receives and produces.

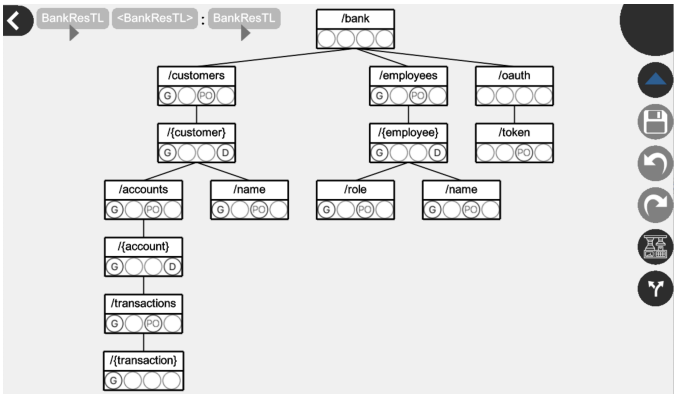

Fig. 4. ResTL Model Defining REST Resources and Methods for the Bank

The last screenshot presents a GUI editor for ResTL, a DSML that was created to model resources and access methods. ResTL assists the design of REST interfaces for distributed systems, notably in the context of the microservice architectural style. RESTful service interfaces consist of hierarchically structured resources with selected CRUD operations [6] (Create, Read, Update, Delete) enabled. Those operations are commonly invoked over HTTP as *Put*, *Get*, *Post*, and *Delete* requests. Figure 4 shows a possible resource layout for the *Bank* system, defining resource based URLs and access methods to trigger account creation and authentication, among others.

D. Weaver

Finally, in order to take advantage of the modularization and reuse capabilities of CORE, a modelling language must also provide a weaver. A weaver is an implementation of a model transformation that takes as input two models of the language, a *source* and a *target* model, as well as a composition specification consisting of a set of mappings that relate model elements from the *source* model with model elements of the *target* model. The weaver composition algorithm must integrate the model elements from the source model with the model elements of the target model according to the composition specification.

TouchCORE currently supports weaving of class diagrams, sequence diagrams, and environment models. The weavers for use case diagrams and ResTL models are under development.

IV. Perspectives

In MDE, a system under development is typically modelled at multiple levels of abstraction and from different points of view. Each model is expressed using the modelling language that provides the most appropriate language concepts for expressing the desired properties. While this promotes modularity and separation of concerns, care must be taken that the views describe the system coherently, i.e., there are no inconsistencies.

TouchCORE 8 supports multi-view modelling with the concept of *perspectives*. A perspective encapsulates one or several plug-in languages intended to be used for a specific modelling purpose. Depending on the purpose, the perspective exposes all or a subset of the language actions of the encapsulated languages as perspective actions that the modeller can use to create and edit models. To ensure consistency between the models, the perspective synchronizes the execution of language actions and keeps track of consistency links between model elements in different models, if needed.

In a perspective, a language plays one or more roles. An example of a single-language perspective is a *domain modelling perspective*, where a class diagram language is used for the purpose of domain modelling, i.e., it plays the domain modelling role. In this case, some class diagram language actions such as `addOperation` are not available to the user, because none of the classes should declare any operations in a domain model.

A more interesting example perspective is a *multi-language perspective* for requirements elicitation that combines the class diagram language and the use case modelling language. The purpose of the perspective is to allow the modeller to capture the concepts of the problem domain and their relationships with a domain model, and specify the goals and services that the system provides as well as the entities with which the system interacts by means of a use case diagram.

Listing 2. Defining the Requirements Elicitation Perspective

```
1  perspective RequirementsElicitation {
2     default Domain_Model;
3     languages {
4        existing language ClassDiagram {
5           roleName Domain_Model;
6        }
7        existing language UseCaseLanguage {
8           roleName Use_Case_Model;
9        }
10    }
11    actions {
12       action redefined Domain_Model
          "ClassDiagramController.createClass";
13       action reexposed Domain_Model
          "ClassDiagramController.createAttribute";
14       ...
15       action redefined Use_Case_Model
          "UseCaseController.createActor";
16       ...
17    }
18    mappings {
19       mapping DMClass_UCActor {
20          fromCardinality 1;
21          toCardinality 0..1;
22          fromElement Class from Domain_Model;
23          toElement Actor from UseCase_Model;
24          nested mappings {
25             nested mapping ClassName_ActorName {
26                matchMaker true;
27                fromElement "name" from
                   Domain_Model;
28                toElement "name" from
                   UseCase_Model;
29 }  }  }  }  }
```

Listing 2 shows how the *RequirementsElicitation* perspective is defined in TouchCORE using the provided perspective DSL. Lines 2-10 declare the languages that are used in the perspective, and the roles they play. Lines 11 to 17 illustrate how the perspective engineer specifies which language actions to expose as perspective actions. For example, since during requirements elicitation classes in the domain model can have attributes, line 13 specifies that the `createAttribute` language action provided by the class diagram language should be available *as is* in the domain model view. In other words, the `createAttribute` language action is *reexposed* as a perspective action.

Lines 12 and 15 instruct TouchCORE to expose the `createClass` language action of the class diagram language and the `createActor` language action of the use case modelling language as perspective actions, but to instrument their execution to ensure consistency between the two models. In fact, lines 18 to 29 specify a consistency link between classes in the domain model and actors in the use case model: every actor in the use case model must have a corresponding class in the domain model. Conversely, classes in the domain model may be linked to actors in the use case model, but do not have to be. Line 26 states that the *name* attribute is used to determine matches between classes and actors.

Based on this specification, TouchCORE automatically ensures that whenever an actor is created in the use case model, the domain model is searched for a class that has the same name. If such a class is found, a consistency link is established connecting the actor and the found class. If no such class is found, a new class is created with the same name, and a consistency link is established. Once linked, TouchCORE will make sure that whenever the name of one of the entities is changed, the other one is updated as well.

V. Generic Navigation

The GUI of TouchCORE 7 already displays a navigation bar at the top of the screen that shows a modeller which model is currently being displayed, and which feature that model is associated with. Furthermore, the navigation bar allows the modeller to navigate from model element to model element *within a design model*. For example, it is possible to navigate from a design class to its superclasses.

Since TouchCORE 8 now supports language plug-ins and multi-view modelling, the navigation bar has been rewritten to be completely generic. The CORE metamodel has been extended to support the definition of intra-language and inter-language navigation [7]. As a result, a language designer can now specify in the language plug-in definition which links in the language metamodel should be navigable from the navigation bar. Likewise, a perspective designer can specify which consistency links should be navigable. Whenever a model is being displayed, the generic navigation bar now updates the provided navigation facilities automatically based on these definitions.

Figure 5 showcases the navigation possibilities offered by the navigation bar when opening the *Bank* domain model in the requirements elicitation perspective. From the class *Clerk*, it is possible to navigate to its superclass *Employee* (an intra-model navigation), but also to the actor *Clerk* in the linked use case model (an inter-model navigation).

VI. Generic Split View

MDE and especially multi-view modelling promotes separation of concerns by allowing a modeller to focus on one view of the system under development in isolation. It is nevertheless useful to be able to visualize several models simultaneously, e.g., to gain an understanding of how the different views work together and to verify that they are consistent.

To this aim, TouchCORE 8 now provides a generic split view that can simultaneously display two GUI editors on the screen, separated by a horizontal or vertical line. Each editor remains fully functional, as the split view simply passes the user's interactions to the appropriate editor. As a result, the displayed models can be edited in the same way than in the single model display mode.

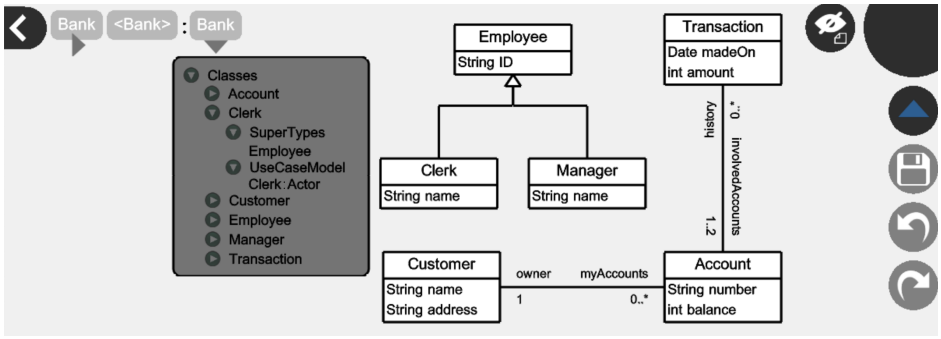

Fig. 5. Navigation Possibilities in the Domain Model View of the Requirements Elicitation Perspective

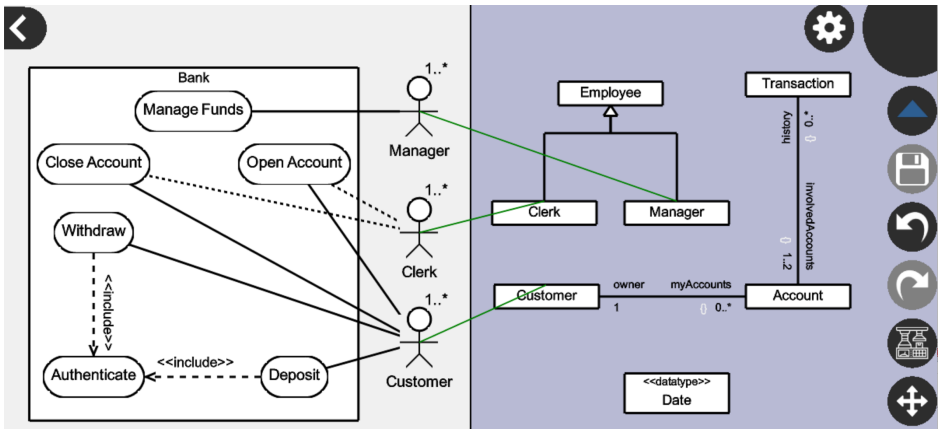

Fig. 6. Use Case Model and Domain Model Visualized Side-By-Side Using the Generic Split View

Additionally, the split view can also display the consistency links established between the models used in a perspective. For example, in the *Requirements Elicitation* perspective, the split view visualizes the consistency links between *Actors* in the *Use Case Model* and *Classes* in the *Domain Model* as illustrated in the screenshot shown in Fig. 6.

The split view also supports creation of consistency links. For example REST resources and CRUD methods of the Bank can be displayed simultaneously using ResTL (see Fig. 4), and a design model implementing the offered services. Then the split view can be used to establish links between the CRUD endpoints in ResTL and the operations of the design classes that implement the offered functionality. Support for consistency links is completely generic, as in the CORE metamodel mappings between models relate EObjects to EObjects.

VII. Conclusion and Outlook

This tools paper presents the new multi-view modelling support of TouchCORE 8. We described the four prerequisites – metamodel, language actions, GUI, and weaver – that a language needs to provide to integrate fully with TouchCORE. We show how to define language plug-ins and perspectives with the provided DSL. We describe the generic navigation support offered by the navigation bar, and showcase the generic split view that displays two models of the same perspective side-by-side.

The next major development effort on TouchCORE migrates the tool to a distributed client-server architecture that will allow us to support collaborative modelling.

References

[1] W. Al Abed, V. Bonnet, M. Schöttle, O. Alam, and J. Kienzle, "TouchRAM: A multitouch-enabled tool for aspect-oriented software design," in *5th International Conference on Software Language Engineering*, ser. LNCS, no. 7745. Springer, October 2012, pp. 275 – 285.

[2] J. Kienzle, W. Al Abed, and J. Klein, "Aspect-Oriented Multi-View Modeling," in *Proceedings of the 8th International Conference on Aspect-Oriented Software Development*. ACM Press, March 2009, pp. 87 – 98.

[3] O. Alam, J. Kienzle, and G. Mussbacher, "Concern-oriented software design," in *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, vol. 8107. Springer, 2013, pp. 604–621.

[4] M. Schöttle, N. Thimmegowda, O. Alam, J. Kienzle, and G. Mussbacher, "Feature modelling and traceability for concern-driven software development with TouchCORE," in *Companion Proceedings of the 14th International Conference on Modularity, MODULARITY 2015, Fort Collins, CO, USA, March 16 - 19, 2015*, March 2015, pp. 11–14.

[5] J. Kienzle, G. Mussbacher, O. Alam, M. Schöttle, N. Belloir, P. Collet, B. Combemale, J. DeAntoni, J. Klein, and B. Rumpe, "VCU: The three dimensions of reuse," in *International Conference on Software Reuse, ICSR 2016, Limassol, Cyprus, June 5-7, 2016*, ser. LNCS. Springer, June 2016, no. 9679, pp. 122–137.

[6] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000, vol. 7.

[7] H. Ali, G. Mussbacher, and J. Kienzle, "Generic graphical navigation for modelling tools," in *Proceedings of the 11th System Analysis and Modelling Conference*, ser. LNCS 11853. Springer, 2019, pp. 44–60.

APPENDIX

In this appendix, we briefly outline the TouchCORE 8 demonstration steps. There are two parts of this tool demonstration: single-language perspective and multi-language perspective.

Single-Language Perspective Demonstration: A single language perspective reuses only one language for a modelling purpose. This first part of the tool demonstration shows how a perspective designer defines and then uses a single-language perspective. The outline is as follows:

- First, we show how a perspective designer defines a single-language perspective which reuses a class diagram language for requirement elicitation. Hence, the class diagram language plays a domain modelling role in the single-language perspective. The presenter, playing the role of a perspective designer, will first show how to register the class diagram language as a language plug-in using the DSL of the TouchCORE tool (see Fig. 7). Then, the designer defines a *domain modelling perspective* which reuses the class diagram for the domain modelling purpose, as shown in Fig. 8.
- Second, the perspective designer launches the TouchCORE tool to show the functionalities supported by the single-language perspective (*domain modelling perspective*). In this perspective, the *create operation* language action is hidden from the user, while other language actions are re-exposed (see sample model in Fig. 9).
- And finally, the perspective designer demonstrates other existing single-language perspectives in the tool, including the environment model perspective, the use case modelling perspective, and the ResTL modelling perspective.

Multi-Language Perspective Demonstration: A multi-language perspective combines more than one language for a modelling purpose. This second part of the tool demonstration shows how a perspective designer defines and then uses a multi-language perspective. The outline is as follows:

- First, we show how a perspective designer defines a multi-language perspective that reuses the *class diagram* language (playing a domain modelling role) and the *use case modelling* language for requirement elicitation purpose. At this step, both the *class diagram* language and *use case modelling* language are registered in the tool. Hence, the designer defines a multi-language perspective (*requirement elicitation perspective*) which reuses the two languages, as shown in Listing 2. At this step, we show how the perspective establishes a correspondence between the `Class` metaclass from the class diagram language and the `Actor` metaclass from the use case modelling language. Additionally, we demonstrate how the perspective redefines some language actions from both languages to maintain consistency between the two models using this established relationship.
- Second, the perspective designer launches the TouchCORE tool to show the functionalities supported by the newly created *requirement elicitation perspective*.

```
external language ClassDiagramLanguage {
  rootPackage "classsdiagram.package";
  rootControllerPackage "classsdiagram.controller";
  packageClassName ClasssdiagramPackage;
  nsURI "http://classdiagram/1.0";
  resourceFactory "ClassDiagramResourceFactoryImpl";
  adapterFactory "ClassdiagramItemProviderAdapterFactory";
  fileExtension  "cdm";
    language elements {
            languageElement ClassDiagram {
            }
            languageElement Class {
                    nestedElement Class_Name elementName name;
            }
            languageElement Attribute {
            }
            ...
    }
    actions {
            create createClass(ClassDiagram owner, String name, ...){
                    metaclass: Class;
                    classQualifiedName : ClassDiagramController();
            }
            create createAttribute(Class owner, String name, ...){
                    metaclass: Attribute;
                    classQualifiedName : ClassDiagramController();
            }
            ...
    }
}
```

Fig. 7. Defining the Class Diagram Language Plug-In with our DSL

```
perspective DomainMOdelling {
    default Domain_Model;
    languages {
            existing language ClassDiagram {
                    rootPackage "classdiagram.package";
                    packageClassName ClassDiagramPackage;
                    roleName Domain_Model;
            }
    }
    actions {
            action reexposed Domain_Model "ClassDiagramController.createClass";
            action reexposed main_Model "ClassDiagramController.createAttribute";
            ...
    }
}
```

Fig. 8. Defining the Domain Model Perspective with our DSL

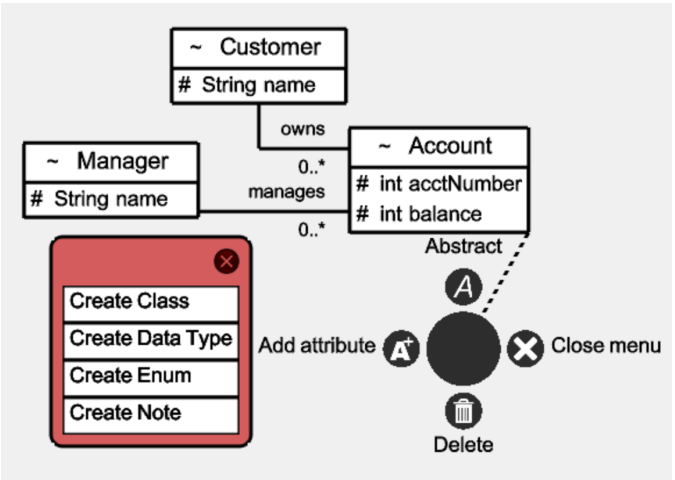

Fig. 9. Domain Model

Again, the *create operation* language action from the class diagram language action is hidden from the user in the domain modelling view. In addition, creating an actor (*redefined action*) in the use case model automatically creates a corresponding class in the domain model.

- Third, we show how a user can navigate between model elements within a single model as well as across different models from a different languages.
- And finally, we demonstrate the TouchCORE split view, which shows two models side-by-side (see Fig. 6), each using a different modelling language. Also, we use the split view to create a mapping between two elements and then visualize the mapping to the user. In this demonstration, we show several mappings, each between a pair of a class and an actor from the class diagram language and use case modelling language, respectively.