Use Case Map Support for TouchCORE

McGill University

Jerry Yu-Chieh Wei

Table of Contents

Table of Figures

Problem/Topic Description

Prior to the project, TouchCORE implemented only one of two supported modelling notations from the User Requirements Notation (URN) – the Goal-oriented Requirement Language (GRL). GRL (Impact and Feature models) provides for the identification and analysis of the actors and system goals, as well as for the modelling of stakeholders and their respective intentions/rationales. With GRL, however, one addresses the non-functioning requirements, but not the functional requirements of a given scenario. One can model a stakeholder's reasoning but not communicate the causal sequences, nor its series of developments. In essence, we answer the "why" but not the "when."

The URN's second notation, Use Case Map (UCM), captures specific responsibilities and components when detailing stakeholder workflows. UCM permits modelling and testing use cases, as well as addresses the non-functional requirements in scenarios. Support for UCMs, therefore, link stakeholder actions and processes to the system's business goal through workflow modelling – thus enhancing TouchCORE with a tool that enables more complete and rounded models.

The workflow of this project is as follows: we started by creating the UCM metamodel using EMF. After this, we developed the element views representing the UCM classes, and associated these views to their classes. Subsequently we built an interface to support user interaction with these elements. We integrated the new extension into the existing TouchCORE application, and then constructed the components necessary for serialization and transitioning to the UCM scene. The first milestone of this extension was to ultimately allow users to create a "basic" UCM path, consisting of a start node, a responsibility, and an end node.

This project references Issue #350 and Issue #371.

Analysis Requirements

- To expand upon CORE's metamodel to include support for the UCM metamodel
- Create views to represent each metamodel element that is displayed visually (path nodes, diagrams, scenes, icons, etc)
- Build controllers to execute commands to modify data in the UCM model
- Construct handlers to process and coordinate user interaction with the application
- Enable saving and loading of a UCM model
- Allow users to select and associate UCM models with features
- Utilize Layout Map and Layout Element to set and save the positions of element views

Pre-Existing Implementation

While there was no pre-existing implementation or support for UCM or workflow modelling prior to the project, the TouchCORE application and its components served as a practical foundation to begin our project on.

The UCM metamodel itself extends the CORE metamodel, allowing access to eClasses from the previous RAM and CORE metamodels. This was useful not only during creation of the UCM metamodel, but also further on in the project as implementation often depended on accessing RAM and CORE's metamodels.

Meanwhile, TouchCORE's implementation of the Multi-Touch for Java (MT4J) framework streamlined much of the work for gestures and UI interaction. This well-founded catalog of various touch actions not only facilitated implementation of view handlers, but also precluded the need for developing additional gestures for the time being.

Moreover, the work done on Aspect models in TouchCORE served as an invaluable example on how realisation models fit into the software as a whole. Having this precedent ultimately proved beneficial during the integration of UCM with TouchCORE.

Design

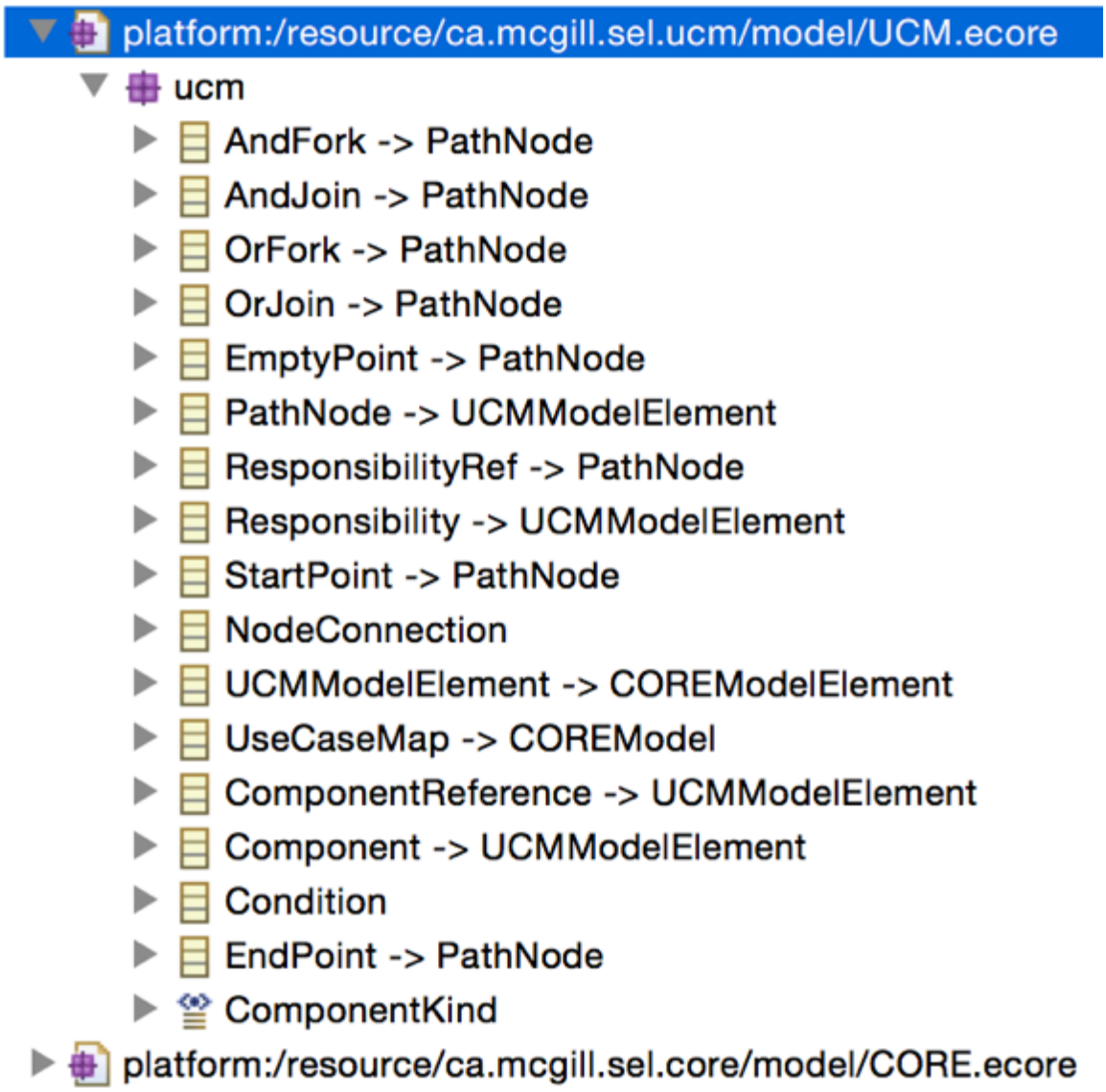

Figure 1: Core metamodel (created in EMF) before layout map

The UCM metamodel is centered around the UCMmap, later renamed to UseCaseMap. This gives UseCaseMap a name (as it inherits from CORENamedElement), and access to the

CoreConcern, the features it realises, its nodes, and countless convenient properties and methods that made development simpler.

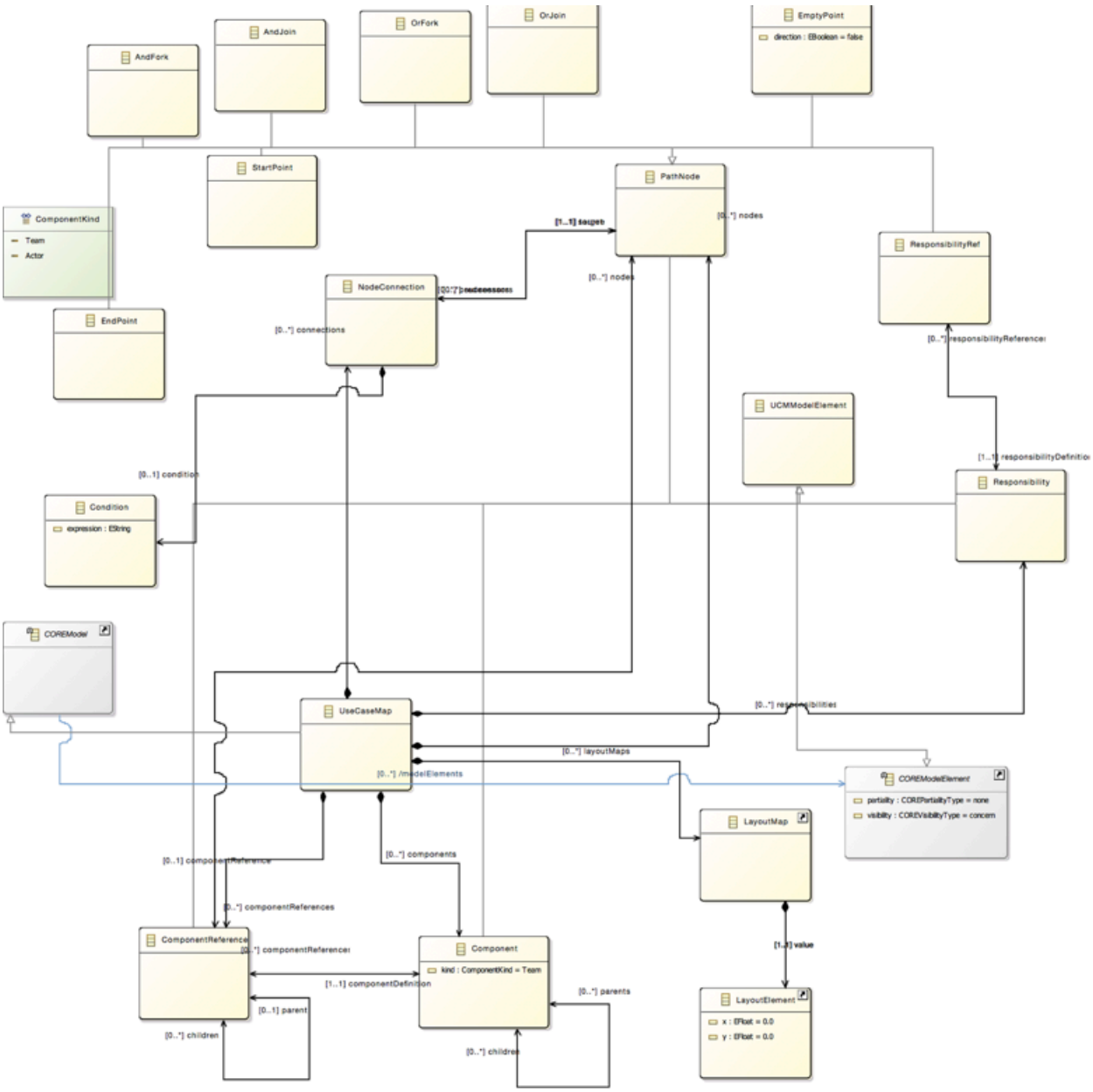

Figure 2: Core metamodel (EMF) after layout map

After integrating this UCM project with TouchCORE, an addition was made to the UCM metamodel to support LayoutMap and LayoutElement classes. This enabled UCMs to save and edit the positions of views on display. Thus, when a UCM file is loaded, its views are reproduced at the same position each time.

One important factor made during high level design decisions in UCM was to maintain consistency with the existing TouchCORE software components. The same or similar touch

gestures were employed in UCM to invoke analogous actions, such as using tap-and-hold to call an instance of a SelectorView menu on a UCM diagram much like in Aspect models.

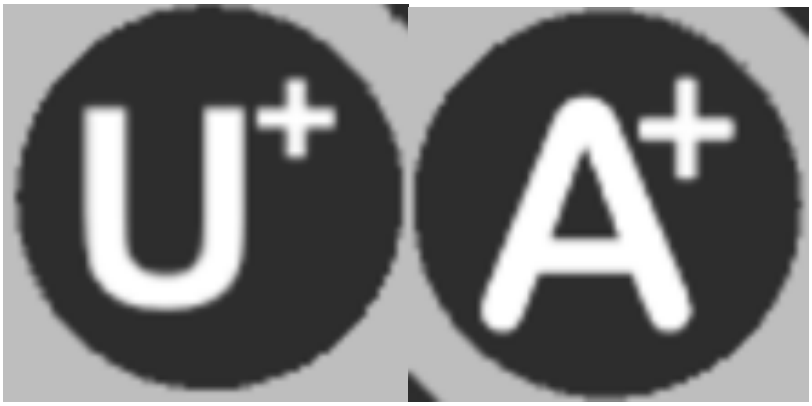

Figure 3: Icon created, next to existing New Aspect icon

By the same virtue, the new icon image in the RamButton, used to initiate a new UCM, was modeled on the New Aspect icon design. The icon image was made using the same graphical editing software, font, and template to maintain uniformity.

UCM views, the visual representation of model elements in TouchCORE, were constructed as children of existing TouchCORE views where possible to maximize code reuse. Every UCM view is a RamRectangleComponent, which itself contains as children the internal elements that together make the view.

Figure 4: StarPointView in TouchCore UCM

Figure 4 shows a StartPointView, the visual representation of a StartPoint in UCM. The StartPointView itself is a ContainerComponent, which is itself a RamRectangleComponent through type hierarchy. Inside this, one ContainerComponent incorporates the TextView to display the Path Node name, whereas a second ContainerComponent contains the RamRoundedRectangleComponent, the black circle of the StartPointView.

This multi-layered approach to constructing views is the same approach used in creating views in TouchCORE. As the application matures, we began considering design decisions revolving around aesthetics and simplifying user interaction. A few examples of such design issues are how to ensure that a NodeConnectionView always connects to the StartPointView in

the center of its circle, or how best to assure that the StartPointView is moveable only when its name label or its circle is dragged.

While following the existing TouchCORE structure prompted a few issues, it was justified by greater cohesion with the existing application and helping create a methodical “procedure” for the UCM project. After completing a milestone in the UCM, one could systematically figure out the next course of action by following what previous developers in TouchCORE did. For example, following development of the path node and diagram views, the next logical step was to fabricate controllers enacting commands to edit the UCM model elements. Following this, the next course was to create handlers that manage gestures and notify the correct listeners, and so forth. This increased cohesion with the application ensures a consistent pattern across all models in TouchCORE, and provides future developers with a familiar structure to work with.

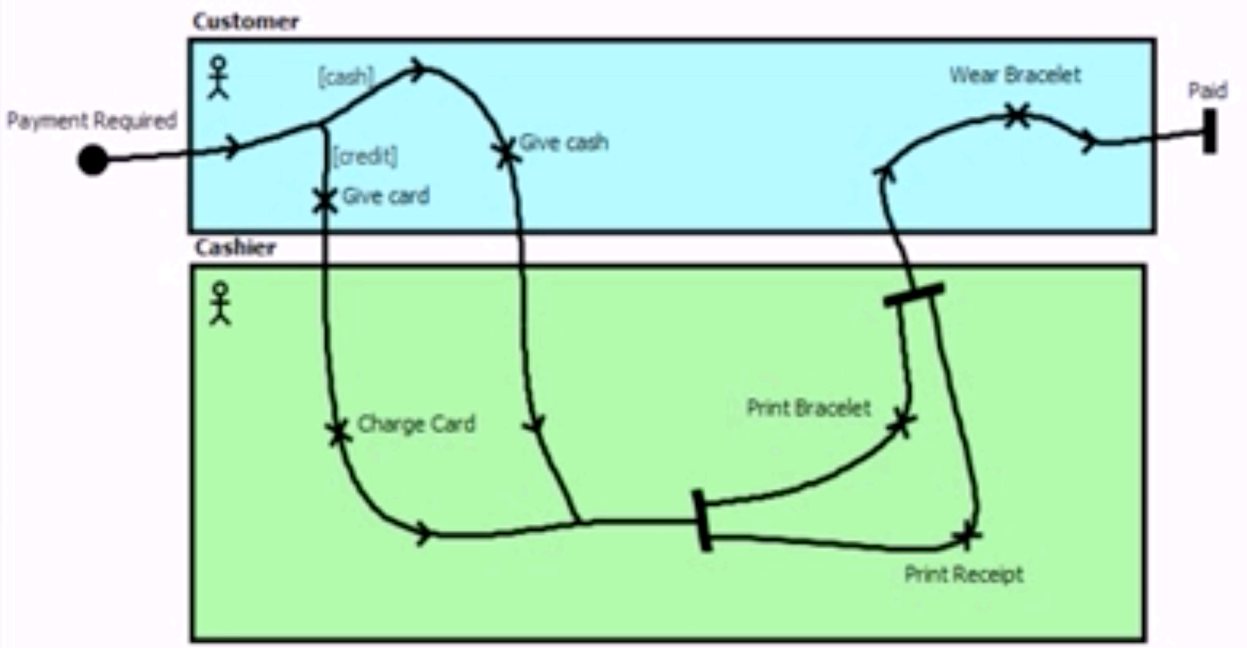

Figure 5: UCM in jUCMNav

An obvious but equally significant influence on design was the jUCMNav plugin and its implementation of UCM. Beyond its basic ideas and visualisation, jUCMNav also had a crucial impact on design decisions regarding how users edit UCMs in TouchCORE. For the operations in jUCMNav that were user-intuitive and worked well, we would consider a similar approach in TouchCORE, while other operations were evaluated and examined for why they did not work or were not feasible, motivating us to find an improvement.

Implementation Details

UCM Metamodel

We began this major augmentation by expanding upon CORE’s metamodel. First, we created the UCM metamodel with EMF, stored in the ecore and genmodel files. This enabled re-generatable code with factories and interfaces, which kept the TouchCORE application separated from implementation classes. Then, using the EMF Editor plugin, we generated a UCM model to validate the metamodel.

PathNode and its Views, Controller, and Handler

Next, we created the three path node views (StartPointView, ResponsibilityView, EndPointView) to represent their respective model objects in the UCM. A PathNodeView class serves as parent to these three views; as such, all generalised features and methods that are applicable to more than one specific path node class (i.e. aggregating a List of NodeConnectionViews of successors and predecessors, updating NodeConnectionView positions, setting a layoutElement, etc.) are incorporated into PathNodeView. So as not to complicate the application, it was decided to only produce classes that were essential to the project. Therefore, UCMModelElementView, the intended parent class of PathNodeView, and various other classes were not produced. While we constructed a new controller for each type of path node to devise and execute commands to edit the UCM model (i.e. StartPointController executes commands to build a StartPoint), all path node views are handled by the PathNodeViewHandler, as there are no differences in how to handle the path node views. The handler updates all the positions of node connections attached to a path node view when dragged, and passes the view's new location to the PathNodeController to be stored. This achieves loading each path node at its saved location on-screen.

UCMDiagramView

The next logical step was to display this UCM model through a diagram view, which contains the UCM path. The UCMDiagramView loads each PathNode from the UCM model and makes the corresponding PathNodeViews with a unique name at its respective positions on-screen. The getUniqueName method serves to differentiate between each path node, and is based off a utility method in CORE. The diagram view's handler, UCMDiagramViewHandler, allows users to instantiate PathNodes at a specific location with a tap-and-hold gesture, displaying options with an OptionSelectorView menu.

NodeConnectionView

Meanwhile, NodeConnectionViews are created if a UnistrokeGesture starts within the boundary of a PathNodeView and ends in another PathNodeView. The NodeConnetionView connects two path nodes while storing both ends' view and object. To improve aesthetics and make it easier for users to connect a NodeConnectionView, EndPointViews rotate to stay perpendicular to their NodeConnectionView. This was done by saving the angle to rotate such that each time the EndPointView or its immediately preceding path node is moved, the EndPointView begins rotation at the last saved angle. An improvement was also made on MathUtils.pointIsInRectangle, the function used by RelationshipView to check if a point is inside a rectangle and a manually-specified acceptance margin. Originally, it did not create NodeConnectionViews if the user started from an EndPointView as the added margin exceeds EndPointView's width. Therefore, the function was modified to pass the minimum of either the margin or the view's width, ensuring that a view would not miss being selected because its dimensions were narrower than the margin.

NodeConnectionView Constraints

At first, NodeConnectionViews were simply constructed in the same direction as the UnistrokeGesture. Further on in the project, this approach caused issues, as it did not account for the constraints inherent to NodeConnectionViews. Firstly, StartPoints, ResponsibilityRefs, and EndPoints cannot have more than one successor or predecessor. More specifically,

StartPoints cannot have predecessors, nor can EndPoints have sucessors, as they denote the extremities in a UCM path. At the same time, a StartPoint or EndPoint cannot be connected directly to another StartPoint or EndPoint, as that would result in an illegal UCM. Finally, a NodeConnectionView is directed, and therefore is produced only when the origin path node has no successors and the destination path node has no predecessors. This also prevents the creation of a duplicate NodeConnectionView if one already exists between two path nodes. To better organize the NodeConnectionViewHandler class, a method orderPathNodes was devised to separate checking of these conditions prior to creating a NodeConnectionView. Early trials found that a notification to the user was imperative, as they may not necessarily understand that an operation was illegal; a PopupType notification now displays to inform users of these conditions.

DisplayUCMScene

Finally, the DisplayUCMScene loads a UCM model with the UCMDiagramView and adds the RamMenu. In the case of a new UCM being produced in the concern, the UseCaseMapController will create a Basic Path and associate the UCM to a feature if necessary. The UCMSceneHandler incorporates handling the functionality of the RamButtons in the RamMenu, as well as confirming and saving the model, destroying the scene, and unloading the UCM model prior to switching back to the concern's feature model. The scene is eventually initialized in one of three ways: tap-and-hold on an existing feature and associating/loading a UCM; through the New UCM RamButton; or from double tapping a loaded UCM in the Realisation Models container.

Integrating UCM with TouchCORE

At the outset, integrating the UCM extension with TouchCORE proved difficult; the UCM started off as a standalone applet, independent of TouchCORE's main application. TouchCORE was designed clearly with Aspect models in mind, without the ability to incorporate new realisation models. An early, reductionist integration strategy was to simply modify affected classes to operate with a COREModel in lieu of an Aspect. This appeared reasonable to try, as Aspect and UseCaseMap both share COREModel as a parent class. Furthermore, the two are both realisation models, designed in the same manner and sharing a similar role of providing users with further details on an associated feature. This simple and rough approach succeeded on a few occasions, such as with the ConcernEditSceneHandler. However, this method proved flawed, as in numerous cases an operation on an Aspect did not apply to a UCM, or vice versa. The two realisation models are fundamentally different – for instance, the back button on the RamMenu transitions back to the COREConcern from a UCM scene immediately, whereas the same button in an Aspect scene cycles through its different views. In the same way, unloading resources in an Aspect requires looping through its instantiations, whereas instantiations do not exist with UCM. In the end, new model-specific methods were ultimately required, although classes were edited to accept both types of models where possible to minimize code duplication and integration issues.

Saving View Positions

An issue faced at this stage was with the layout of the UCM. The original implementation of the UCMDiagramView simply created PathNodeViews on a horizontal line, equidistantly apart. The proposed improvement on this rudimentary method was to utilize LayoutMap and LayoutElement to save the position of each node in the model. This required a second amendment of the UCM metamodel - the UseCaseMap eObject was subsequently altered to

compose LayoutMap, which stores a LayoutElement. More specifically, a LayoutMap is a single mapping from an eObject (the PathNode) to a LayoutElement (the PathNodeView position). The controllers of each path node type were edited to pass and create a LayoutMap and LayoutElement to store its view's position. Likewise, the UCMDiagramView and PathNodeView classes were changed such that a view is constructed at its LayoutElement position. As the LayoutElement is stored with its PathNode, the save implementation does not change. Now, when reloading a UCM diagram, each PathNodeView will display at its last saved location.

Future Work

We continue where we left off by finishing production of the remaining path nodes (i.e. AndFork, AndJoin, OrFork, OrJoin, ComponentRef, etc). While the graphical representation may be straightforward after all the groundwork laid in this project, certain design decisions must be resolved, such as which gesture will be most user-intuitive to create a Fork/Join.

In the same manner, we plan to implement the EmptyPoint, and the functionality to add and save a Condition for a NodeConnection.

Also, a ResponsibilityPanel should be added to display all the Responsibilities in the UCM scene, similar to how Impact models are listed in DisplayConcernEditScene. The same could be constructed for Components as well.

Additionally, many of the classes edited in integrating UCM with TouchCORE can be modified to function with any model. This will not only make it easier to integrate subsequent models added to the application, but will also further future-proof TouchCORE for the time to come.

External Links

- http://jucmnav.softwareengineering.ca/ucm/bin/view/UCM/WebHome
- http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/WebHome