# Reusing Software Design Models with TouchRAM

Jörg Kienzle
School of Computer Science, McGill University
Montreal, QC H3A 0E9, Canada
Joerg.Kienzle@mcgill.ca

## ABSTRACT

TouchRAM is a multitouch-enabled tool for agile software design modelling aimed at developing scalable and reusable software design models. This paper briefly summarizes the main features of the Reusable Aspect Models modelling approach that TouchRAM is based on, and then describes how the tool is used during the design process to incrementally elaborate a complex software design model.

## Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools

## Keywords

model interfaces, model hierarchies, model reuse

## 1. INTRODUCTION

TouchRAM is a multitouch-enabled tool for agile software design modelling aimed at developing scalable and reusable software design models. The tool gives the designer access to a vast library of reusable design models encoding essential recurring design concerns. It exploits model interfaces and aspect-oriented model weaving techniques as defined by the Reusable Aspect Models (RAM) [4] approach to enable the designer to rapidly apply reusable design concerns within the design model of the software under development. The user interface features of the tool are specifically designed for ease of use, reuse and agility (multiple ways of input, tool-assisted reuse, multitouch). A screenshot of TouchRAM v1.0.1 running on a multitouch-enabled 60-inch display is shown in Fig. 1.

This demo paper briefly summarizes the main features of the RAM modelling language that our tool exploits to make design model reuse possible in section 2, and then describes the process of building a complex software design using TouchRAM in section 3.

## 2. ESSENTIAL FEATURES OF RAM

TouchRAM is based on Reusable Aspect Models (RAM) [4], an aspect-oriented multi-view modelling approach that integrates class diagram, sequence diagram and state diagrams. As a result, a RAM model can describe the structure and the behaviour of a software design concern. Currently, however, the TouchRAM tool only provides full support for structural modelling with class diagrams, and partial support for behavioural modelling with sequence diagrams.

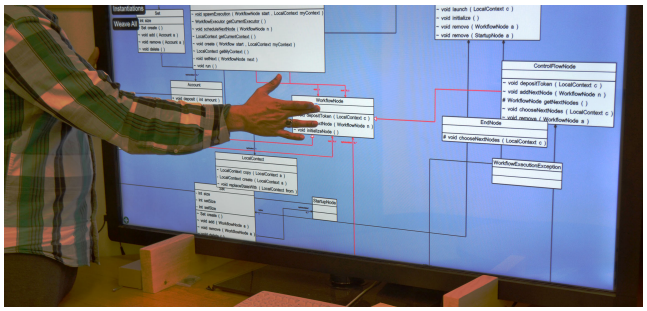

**Figure 1: TouchRAM in Use**

The most important concepts of RAM that make incremental software design modelling possible are *model interfaces* and *model hierarchies*.

### 2.1 Design Model Interfaces

Every RAM model has a well-defined *model interface* [2], which makes it possible to package a design concern in such a way that it is easy to use within other models. The interface has two parts: the *customization interface* and the *usage interface*.

The *customization interface* specifies how a generic design model needs to be adapted to be used within a specific application. To increase reusability of models, a RAM modeller is encouraged to develop models that are as general as possible. As a result, many classes and methods of a RAM model are only partially defined. For example, for classes it is possible to define them without constructors and to only define attributes relevant to the current design concern. Likewise, methods can be defined with empty or only partial behaviour specifications. The idea of the customization interface is to clearly highlight those model elements of the design that need to be completed/composed with application-specific model elements before a generic design can be used for a specific purpose. These model elements are called *mandatory instantiation parameters*.

The *usage interface* is similar to a "classic interface" found in programming languages. It is comprised of all the *public* model elements, i.e., the structural and behavioural properties that the classes within the design model expose to the outside. In other words, the usage interface presents


*AOSD'13 Companion,* March 24-29, 2013, Fukuoka, Japan.

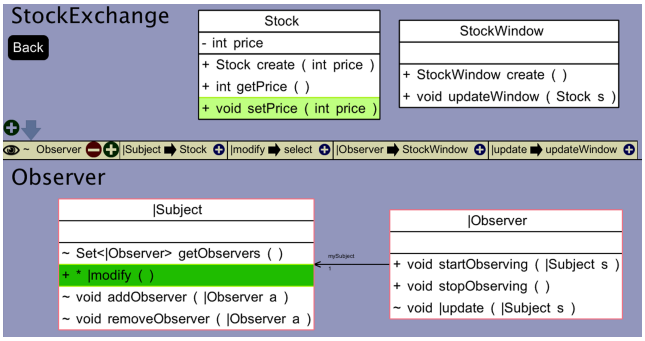


**Figure 2: Reusing the Observer Design Pattern**

an abstraction of the functionality encapsulated within the model to the user of such a model. It describes how the rest of the application can trigger the functionality provided by the model by instantiating classes and invoking operations on them. At the same time, the usage interface hides the internal details of the design model from the rest of the application, which does not need to know how the functionality is decomposed into classes/methods and how objects interact at run-time to achieve the functionality.

The bottom half of Fig. 2 shows the interface of the RAM model for the *Observer* design pattern [3], a software design pattern in which an object, called the *subject*, maintains a list of dependents, called *observers*. The functionality provided by the *Observer* model is to make sure that, whenever the subject's state changes, all observers are notified by calling an *update* operation. The mandatory instantiation parameters that make up the customization interface are highlighted in Fig. 2 by adding a "|" prefix to the name of the class/method/attribute. In case of the *Observer* design model, the mandatory instantiation parameters are the `|Subject` class, which also defines at least one |modify operation, and the `|Observer` class, which must provide an `|update` operation.

The usage interface of the *Observer* model is straightforward. The `|Subject` class must provide at least one public operation that modifies its state, and the `|Observer` class provides two operations, namely `startObserving` and `stopObserving`, that allow an observer instance to register, rsp. deregister, with a subject instance.

## 2.2 Design Model Hierarchies

RAM allows a modeller to build complex models of any size by putting together many interdependent, simple models. This is achieved through *model hierarchies*. To use the functionality provided by a (base) RAM model $B$ within the design of a model $A$, the designer must specify instantiation directives that map all mandatory instantiation parameters of the customization interface of $B$ to model elements of $A$. Using these directives, the TouchRAM tool can compose both models to yield a "woven" model that combines the model elements from both designs.

Fig. 2 depicts a situation where a modeller applies the *Observer* design model (model shown in the bottom half of the figure) to a *StockExchange* application model (model shown in the top half of the figure). The modeller has already mapped the `|Subject` class of the lower-level model to the `Stock` class, and the `|Observer` class to the `StockWindow` class, and is now in the process of mapping the `|modify` operation of `|Subject`. Since `|modify` is part of the class `|Subject` in the lower-level model, and `|Subject` was already mapped to `Stock` in the higher-level aspect, TouchRAM marks only the methods of `Stock` with matching parameters as selectable.

RAM supports two kinds of model dependencies, *model extensions* and *model customizations*. When $A$ *extends* $B$, the modeller's intent is to add additional structural and/or behavioural model elements to $B$ that provide *additional*, *alternative* or *complementary* properties to what already exists in $B$. *The extension model $A$ augments the interface of the base model $B$* with additional structure and behaviour. A *customization* is useful when a modeller's intent is to adapt the structure and behaviour provided by a base model $B$ to be useful in a specific context. Within a customization model $A$, a modeller *alters* or *augments* existing base model properties to render them useful for a new purpose. When using customization, typically most mandatory instantiation parameters of the base $B$ are completed, i.e., mapped to complete model elements in $A$.

In the example shown in Fig. 2, the *StockExchange* application model customizes the generic *Observer* model: it uses the *Observer* design internally to redraw the contents of the `StockWindow` when the state of the corresponding `Stock` instance changes. Customization is shown in TouchRAM by a ~ sign in front of the instantiation directive that specifies the parameter mappings; extension is visualized by a + sign.

In RAM, a model $C$ can depend on one or several models $B_i$, which in turn can depend on models $A_{ij}$, etc., thus creating a model hierarchy. For each pair of models, instantiation directives in the upper model specify how the elements in the lower model are to be combined with the model elements of the higher model. Using these directives, TouchRAM can recursively combine all models of the hierarchy to yield a design model that shows the complete design. How this feature is exploited to create a design of a real-world application model is shown in the following section.

# 3. SOFTWARE DESIGN WITH TOUCHRAM

Software design modelling with TouchRAM integrates well with modern software design processes, e.g., prototyping or iterative methodologies, as the design and implementation of the application is conducted in phases. First, a simple version of the application is developed that only provides core functionality and services. Detailed and additional functionalities are added in subsequent iterations.

## 3.1 Design Modularization Strategies

**Completeness**: The most important criteria for designing with model hierarchies is *coherent modularization*. Each model specifies a logical design step towards the final design model, and therefore needs to contain *all* the structural and/or behavioural elements pertaining to that logical step. This is important for *internal consistency* of the model: it simplifies reasoning about the design concern as well as making coherent changes to the modelled structure and/or behaviour, if needed. An additional advantage of completeness of individual models is that, by construction, any composed model is therefore also complete.

**Size**: Each individual RAM model should be small, as it has been shown in psychological studies that the active *working memory* of a human is limited [5]. Examining or building a model of a system induces a certain mental effort on the modeller. This effort is correlated with the model size,

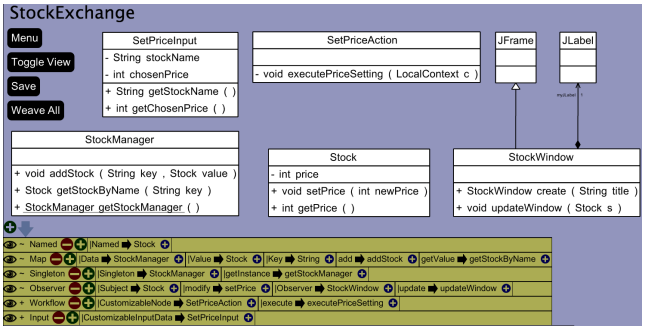


**Figure 3: The Stock Exchange Model**

and influences the amount of working memory the modelling activity utilizes [6]. When an individual undertakes a mental task (e.g. attempting to analyze a model or answer questions about a model) that exceeds their working memory capacity, errors are likely to occur [7].

**Vertical Design Decomposition**: One way of modularizing a complex software design is to follow a top-down and/or bottom-up strategy, depending on whether the focus is to first elaborate high-level abstractions and functionality, or rather to initially flesh out certain important low-level details of parts of the design. For instance, if detailed functional requirements for the software under development have been elaborated, the initial design phase might begin with deciding on a high-level architecture for the system, and how the required functionality is to be decomposed into subfunctionalities and allocated to different components. On the other hand, if a certain subfunctionality is crucial to the functioning of the software under development, or if reusing an existing software artifact such as a middleware is mandatory or highly cost-effective, then low-level details of a specific required functionality might be designed first in order to determine if the design is actually feasible.

To enable such top-down or bottom-up design, *abstraction* and *information hiding* are key to tame the inherent complexity of a system [**?**]. Information hiding is the activity of consciously deciding what parts of a software module should be exposed to the outside, i.e., the "rest" of the software under development, and what parts should be hidden from external use. In RAM this is done by exposing only the structural and behavioural properties that are relevant to use the model in the model's interface. The design details pertaining to *how* this functionality is provided are not relevant to the user. This is why most higher-level models that reuse functionality provided by other models use model customization, since the customized model interfaces of the lower-level models are automatically hidden from the user.

**Horizontal Design Decomposition**: When transitioning from one iteration of the software design to the next, it is typical to consider additional functionality. As a result, the core parts of the existing design are complemented with additional functionality, or new components are introduced that take care of providing the additional functionality and the existing design is adapted to integrate the new components. This form of incremental design is supported in TouchRAM using model extensions, which can *adapt and extend* existing *model interfaces*.

## 3.2 Example Design

Fig. 3 shows a high-level design model of parts of the *StockExchange* application. In addition to the class `Stock`,

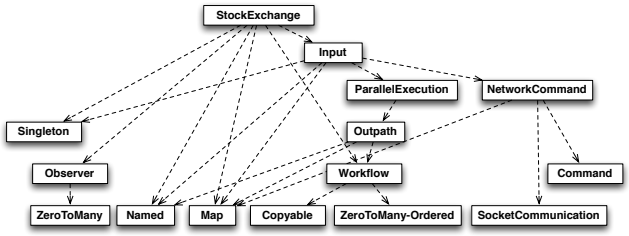


**Figure 4: StockExchange Design Model Hierarchy**

which encapsulates the business data for the application, there is also a class `StockWindow`, which subclasses `JFrame` and uses `JLabel`s to display the current stock name and price. `JFrame` and `JLabel` are classes provided by the Java runtime as part of the Java GUI framework Swing. To maximally exploit reuse of existing implementations, TouchRAM allows any Java class to be imported into a RAM model as a so-called *implementation class*.

The main advantage of using TouchRAM for software design modelling is to reuse existing design models (and their implementations). TouchRAM comes with a Reusable Design Concern Model Library (RDCML), which contains premodelled solutions for recurring design concerns. The library is organized into the following categories:

- The *design patterns* category contains aspects for the basic structural, behavioural and creational design patterns (e.g., *Singleton*, *Observer*, *Command*, etc.)
- The *utility* category contains aspects that provide basic functionalities like copying *(Copyable aspect)* and naming *(Named aspect)*, as well as data structures involving multiple objects, such as *Map*.
- The *networking* category contains aspects relevant to networking, such as *Serializer, SocketCommunication* and *NetworkedCommand*.
- The *workflow* category contains aspects that are useful whenever the application needs to define and execute flexible workflows. For example, the current library supports sequential, conditional, timed, nested and parallel execution of activities.
- The *transactions* category contains aspects that provide support for state checkpointing, state recovery and concurrency control.

The *StockExchange* application model, for example, depends on 6 models from the RDCML as shown by the 6 instantiation directives in the bottom half of Fig. 3. It customizes the model *Named* to add a name attribute as well as getter and setter operations to the `Stock` class. It customizes *Map* to allow the `StockManager` class to find a `Stock` instance based on a name `String`. It customizes the *Singleton* design model to ensure that there is only a single, global instance of the `StockManager` running in the application, and it customizes the *Observer* to refresh the stock window whenever the price of a stock changes.

The above are all examples of quite simple design models. To show off the benefit of reuse, we also designed the *StockExchange* application to be able to act as a server, listening on the network for incoming messages that inform the application of stock price changes. The RDCML contains a workflow execution middleware, which provides abstractions to define and execute complex workflows that can be used, for instance, to specify the interaction protocol of a server. In

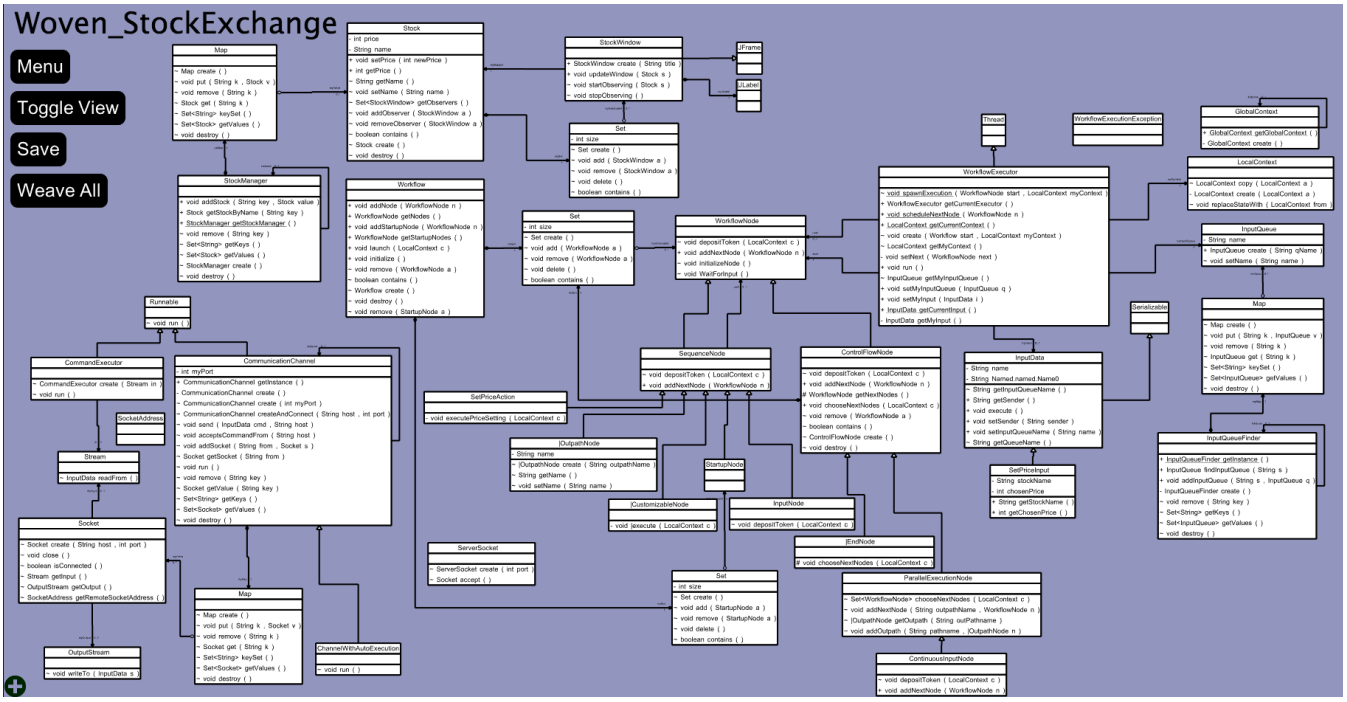


**Figure 5: Woven Stock Exchange Application**

our example, the *StockExchange* application model defines a `SetPriceAction` class, which is mapped to the `|CustomizableSequence` class provided by the *Workflow* model. This makes it possible to change the price of some stock within a workflow that is executed by the middleware. In addition, in order to trigger this action remotely, *StockExchange* defines a `SetPriceInput` class that is used to store the input parameters of the action, i.e., the stock name and new price. This class is mapped to the `|CustomizableInput` class defined by the *Input* model that extends *Workflow*.

A design that supports execution workflows based on remote commands received over the network is inherently complex. It requires, for instance, the definition of many classes that represent workflow concepts, the creation of listener threads that wait on TCP/IP ports, and the definition of serialization protocols for data that need to be transmitted over the network. This is why these functionalities, provided by the RDCML, have not been designed within one model, but again modularized and built using other design models of the RDCML. The entire model hierarchy that comprises the design model of our simple *StockExchange* application is shown in Fig. 4.

To give the reader a sense of the power of the model reuse capabilities offered by TouchRAM, Fig. 5 shows the final design model of the *StockExchange* application, obtained by composing *StockExchange* with the 14 design models from the RDCML it directly or indirectly depends on. The composition took approximately 2 seconds to execute on a standard 2.5GHz machine with a JVM configured to run with 1GB of RAM. The resulting model has 41 classes and 44 relationships between them, and a total of 157 operations.

## 4. CONCLUSION

This demo paper summarizes how TouchRAM, a multitouch-enabled tool for agile software design modelling, can be used to elaborate a complex software design model. For more information on the multitouch user interface and the model transformation technology that TouchRAM is based on, the interested reader is referred to [1]. The current version of TouchRAM and the reusable design concern model library can be downloaded from http://www.cs.mcgill.ca/~joerg/SEL/TouchRAM.html. The tool runs on any reasonably modern Mac/Linux/Windows operating system, provided that Java 1.5 is installed and the graphics card supports Open/GL. Optionally, to use the multitouch features, a TUIO supported multitouch input device must be connected.

## 5. REFERENCES


[1] Al Abed, W., Bonnet, V., Schöttle, M., Alam, O., and Kienzle, J. TouchRAM: A multitouch-enabled tool for aspect-oriented software design. In *SLE 2012* (October 2012), no. 7745 in LNCS, Springer, pp. 275 – 285.

[2] Al Abed, W., and Kienzle, J. Information Hiding and Aspect-Oriented Modeling. In *14th Aspect-Oriented Modeling Workshop* (October 2009), pp. 1–6.

[3] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns*. Addison Wesley, Reading, MA, USA, 1995.

[4] Kienzle, J., Al Abed, W., and Klein, J. Aspect-Oriented Multi-View Modeling. In *AOSD 2009* (March 2009), ACM Press, pp. 87 – 98.

[5] Miller, G. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review 63*, 2 (1956), 81.

[6] Paas, F., Tuovinen, J., Tabbers, H., and Van Gerven, P. Cognitive load measurement as a means to advance cognitive load theory. *Educational psychologist 38*, 1 (2003), 63–71.

[7] Sweller, J. Cognitive load during problem solving: Effects on learning. *Cognitive science 12*, 2 (1988), 257–285.