Enhancing Binding-based User Interfaces with Transaction Support

Nicolás Passerini
Universidad Nacional de Quilmes
Universidad Nacional de San Martín
Universidad Tecnológica Nacional
npasserini@gmail.com

Javier Fernandes
Universidad Nacional de Quilmes
Universidad Nacional de San Martín
Universidad Nacional del Oeste
javier.fernandes@gmail.com

Ronny De Jesus
Universidad Nacional de Quilmes
nnydejesus@gmail.com

Pablo Tesone
Universidad Nacional del Oeste
Universidad Nacional de Quilmes
Universidad Nacional de San Martín
tesonep@gmail.com

Leonardo Gassman
Universidad Nacional de Quilmes
lgassman@gmail.com

Categories and Subject Descriptors D-2.2 [*Design Tools and Techniques*]: User interfaces; D-2.3 [*Coding Tools and Techniques*]: Object-oriented programming

Keywords MVC, MVVM, MVB, Transactions, User Interface, Aspect Oriented Programming, AOP, Observers, Patterns

Abstract

In the construction of object-oriented user interfaces, a significant amount of time is usually spent in a few rutinary tasks related to the transference of data between domain objects and user interface components. Multiple repetitions of these simple tasks are a common source of programming errors. Binding-based MVC architectures, such as Direct Manipulation, MVVM and MVB, reduce these tasks but introduce some non-trivial issues like cancelling the current operation and rolling back changes. This work proposes an aspect-oriented solution to these problems combining well-known aspect-oriented techniques, such as observable objects and software transactions, to provide a simple and transparent MVC implementation. Our solution focuses on providing warranties of *atomicity*, *consistency* and *isolation* of objects environments and it demonstrated to be useful in a big range of applications.

1. Introduction

An usual pattern in the development of an interactive software application is to separate the *user interface* (UI) from the *domain logic* [3]. In this way an interactive application contains specific components for handling the UI, separated from those which handle domain logic, with the objective of improving the flexibility and maintainability of the system.

Since the objective of the UI is to allow the user to interact with the domain logic [5] a new problem arises from this separation, which is the transfer of information between the UI and the domain model. A big amount of the information entered by the user in the UI is stored in the domain model, after some validation and transformation. Also, most of the information shown in the UI comes from the domain model. It is not uncommon to see industrial projects which handle this information transfer manually, or UI frameworks which do not provide a simple way to avoid multiple repetitions of the transformation and validation logic. Although these tasks are often not too complex, repeating the same logic multiple times is considered a *code smell* [13] because it represents a possible source of inconsistency. Also, multiple repetitions of the same logic undermines the desired flexibility.

In an MVC architecture [35] the task of communicating the components of the UI (i.e. the *view*) and the components which implement the domain logic (i.e. the *model*) is handled by a third kind of component, named *controller*.

Among the model-view-controller triad, the controller is the component with more diverging shapes in the different interpretations of the MVC pattern [16, 24, 27, 32–34, 37, 39]. Some of these strategies have achieved a step forward in reducing the work needed for handling the transfer of information between view and model, by using small, fine grained, reusable controllers to build a *binding* between any property from the domain model to an input field in the UI (c.f. Sect. 2.1). However, simple binding strategies change the model while the user is entering data, providing no way to handle the cancellation of the current operation and rolling back changes in case of failure.

We propose a solution to these problems based on *aspect-oriented programming* [22], which uses two aspects to intercept modifications to the desired domain objects. In the first place, the *Observable Aspect* has the responsibility of producing events each time a domain object is modified. These events are consumed by the data-binding mechanism of a MVB [27] framework in order to keep view and model synchronized with each other. On the other hand, the *Transactional Aspect* implements an STM solution [38]. Therefore, it records all changes made to domain objects and, in case of an error or cancellation, provides an automatic mechanism to restore the domain objects to their state at the beginning of the cancelled operation.

Our main contribution is a new way of integrating these two aspects to build an MVC implementation, which both reduces the *boilerplate code* [26] needed and provides some tools to warranty the principles of *atomicity*, *consistency* and *isolation* at the object level[1] [36]. We will also describe the details of the framework

[1] The fourth characteristic of a transactional context, *durability*, is not important for this work.

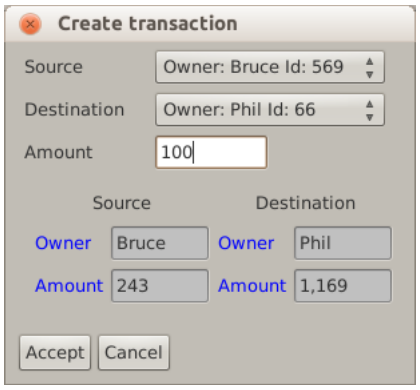
we built to support this approach and our experience developing applications in this way.

This work is organized as follows. Sect. 2 describes the problem we are trying to solve and shows the motivation to our approach. Sect. 3 depicts the main objectives of this work and how we propose to solve the problems we describe before, while Sect. 4 delineates a sample implementation of these ideas. Sect. 5 shows a example application. Sect. 6 discusses our approach, analysing some possible variants and 7 compares it to other ideas with similar objectives. Finally, Sect. 8 shows our conclusions and the next steps of our research.

2. Motivation

This section describes the problem we are trying to solve, how it is usually tackled in industrial developments and the drawbacks we have found in those solutions, which in turn motivate us to look for a new approach.

We illustrate the different approaches with an example which manages bank transfers. The system allows the user to select two accounts, enter an amount and transfer it. Fig. 1 shows transfer dialog. The transfer process implies two steps: first we withdraw money from one account, and then we deposit the same amount of money into the other one. Fig. 2 shows the `Transfer` and `Account` classes.


Create transaction

Source: Owner: Bruce Id: 569
Destination: Owner: Phil Id: 66
Amount: 100

Source		Destination	
Owner	Bruce	Owner	Phil
Amount	243	Amount	1,169

Figure 1. Single Transference Screen

The domain model is responsible for validating that only valid transfers are executed. If an invalid transfer is attempted, it is rejected by throwing an `InvalidAmountException`. Since the exception thrown is a subclass of `UserException` we expect that de UI catches it and informs the situation to the user. Still, the domain model (i.e. the `Transfer` class) is responsible for keeping itself consistent. It is worth noticing that the code in Fig. 2 would handle correctly any exception thrown by the `withdraw` method, because the `deposit` would not be executed.

2.1 Constructing Applications Using the MVC Pattern

One of the main objectives of the MVC pattern is to de-couple the view from the domain model. To achieve this, the controller *mediates* [15] between view and model. When the user inputs information to the application through the view, the controller is responsible for updating the model with this new information. Previously, the controller validates the new information and transforms it according to the needs of the model. Similar transformations need to be done in order to show the user the information coming from the domain model. The logic of these validations and transformations is often fairly simple. The main problem is that these rules have to be enforced in every user interaction, i.e. every piece of

```
class Account {
  private var balance

  def withdraw(amount) = {
    if (balance < amount)
      throw new InsufficientFundsException
    balance -= amount
  }

  def deposit(amount) =
    balance += amount
}

class Transfer {
  private var source
  private var destination

  private var _amount
  def amount = _amount
  def amount_=(newAmount) = {
    if (newAmount < 0)
      throw new InvalidAmountException
    else
      _amount = newAmount
  }

  def execute {
    source.withdraw(this.amount)
    destination.deposit(this.amount)
  }
}

class InvalidAmountException extends UserException
class InsufficientFundsException extends UserException
```

Figure 2. Fragments of the domain model

information entered by the user has to be validated and possibly transformed. Without an high level tool to describe the validation and transformation logic, we could end up producing large amount of repetitive, error prone and hard to maintain *boilerplate* code.

Since Reenskaug first proposed the model view controller pattern in 1979 [35], his idea seems to be into a *de facto* standard in the software industry. However, there has been a large quantity of *diverging* interpretations of this pattern: almost every UI framework claims to be an *MVC framework*, yet having big differences among them. The different interpretations of the MVC pattern can be divided in two large groups, depending on the shape of the controller.

In the first group, we consider MVC interpretations which choose to use big, coarse grained controllers which act as a *facade* [15] to the domain logic. This is the most simple version of the MVC, and the controller has the responsibility of coordinating the whole application flow, it receives *requests* from the view, extracts the data contained in each request, validates and transforms the data, decides which action to execute, invokes the necessary domain logic (in some cases it even executes some domain logic by itself), interprets the results of the domain operation and chooses the next view to show to the user. In many cases, the consequence of trying to de-couple view and model using this strategy is to create a third participant (the controller) that is coupled to both of them. In such situations, the controller becomes a complex and error prone component, and is the main cause of the UI consuming more development effort than any other part of the system [28].

In the second group, we consider *event-based* MVC interpretations. The *Observer Pattern* [15] is a typical approach to handle event-driven programming in object-oriented systems. The Observer Pattern defines two main actors: a source or *observable* subject that produces the events; and an observer or *listener* which registers itself to be notified when a specific type of event occurs. A typical implementation uses an intermediary named *event handler*, which keeps a global *registry* of the listeners and is responsible for receiving all events and dispatching them to the corresponding listeners.

In this case, instead of a big controller coordinating application flow, we have multiple small, fine grained controllers which listen to the events produced by both view and model. There is no need for a global coordination of the application flow, which is instead guided by the events which are produced by the view on each user action. Besides, any change in a domain object can be modelled as an event, which a controller can listen to and therefore update the view accordingly [11]. The rest of this work focuses on this kind of user interfaces.

Events allow a very low-coupled communication between the source and the listeners, since the source only has to *fire* the events without needing to know which components are going to handle them (or even *if* they are going to be handled). Consequently, events conform a very useful mechanism for implementing the MVC pattern.

2.2 Bindings and Properties

An event-driven implementation of MVC allows the usage of *bindings* between view and model. A binding is a declaration that establishes a connection between two *properties*[2]. A *binding manager* keeps a registry of these declarations and ensures that the values of bound properties remain synchronized with each other. In order to be able to bind properties, they have to fire events when its value changes. The event will reach the binding manager, which updates all bound properties according to the change.

A *property* is a named characteristic of an object that has a value which is exhibited through its public interface. In some cases, the value of the property can also be updated through the public interface of the object. It should not be confused with an instance variable, which is an implementation issue. Instead, a property is a feature that an object exhibits in order to let other objects interact with it. Also, properties are allowed to have some associated behavior, for example when an update of the value is requested the object can validate the new value and *veto* the change, or register it in a change log. Moreover, properties can be calculated; normally clients may not know if the property they are using is calculated or not. Some languages, such as C# [19] or Scala [30] provide specific constructs to define properties. In older languages they are implemented as methods that follow a name convention. For example the Java language defines a standard, named *JavaBeans* [12], which specifies that a property named `height` should be implemented using the methods `getHeight` and (optionally) `setHeight`. These conventions allow that frameworks identify properties automatically.

Bindings can also be configured to perform *transformations* and *validations*. Transformations are necessary when the two properties are not of the same type, for example a String-valued property can be bound to a Number-valued property, provided that the binding manager can convert the values when needed. The binding manager has to provide a mechanism to handle failures, which can occurr on transformation, validation or as a *veto* from the property itself.

In UI construction bindings can be used to synchronize a property from a domain model object. with a UI component that shows a value or allows the user to enter a value (such as a TextBox or a ComboBox). Different authors have proposed *binding-based* MVC architectures, such as Direct Manipulation [32], MVVM [16, 39] and MVB [27] These MVC implementations have achieved a big step improvement in reducing the boilerplate code in controller objects, and therefore the portion development effort which has to invested in the user interface.

Binding-based frameworks provide a strategy to *structure* the controller code. Each kind of validation is defined in a small, reusable object called a *validator*. All validators implement the same interface, so they are polymorphically interchangeable. The same idea is applied to create small, reusable polymorphic transformers. The binding-mechanism puts all these objects together. Frequently it provides a *declarative* way to define a binding between two properties, and at the same time associate it with the needed validators and transformers. In many cases, these frameworks can even infer which transformed is required using *reflection* techniques. In this way, binding provides a very high-level strategy to transfer information between view and model.

Fig. 3 shows a simplified version of the code needed to define a binding-based view for the `TransferDialog`. The exaple is coded using a UI framework named Arena[3]. The Arena framework is based on SWT [29] and JFace [18], which in turn use JavaBean events [25] in order to update the visual components when the underlying domain objects change.

Both ComboBoxes and TextBoxes define `bindValue` methods, which bind the main value of the components to the property referred by the received *block expression* [30]. Also, ComboBoxes allow us to bind the list of possible options, through the `bindItems` method.

```
class TransferDialog extends Dialog {
  val model = new TransferViewModel

  def createContents(mainPanel) {
    new ComboBox
      .bindItems { model.possibleOrigins }
      .bindValue { model.transfer.source }
    new ComboBox
      .bindItems { model.possibleDestinations }
      .bindValue { model.transfer.destination }
    new TextBox
      .bindValue { model.transfer.amount }

    new Button
      .caption = "Accept"
      .onClick { model.transfer.execute }
  }
}

class TransferViewModel {
  val transfer = new Transfer
  def possibleOrigins: List[Account] = ...
  def possibleDestinations: List[Account] = ...
}
```

Figure 3. Simplified code of the `TransferDialog` class. Layout information has been removed.

Binding UI components to domain objects allows domain objects themselves to validate the operation in course, for example if the transfer amount is negative or if the account has not enough

[2] *Binding* and *property* are terms that can have different meanings in other contexts, here we use them with their classic interpretation in UI construction.

[3] Arena is an educational UI framework developed by teachers of the universities of Quilmes (UNQ, Argentina), Tecnológica (UTN, Argentina) and San Martín (UNSAM, Argentina). It has been used to teach topics about UI construction since 2010. More details about the UI framework can be found in `https://sites.google.com/site/programacionui/material/herramientas/arena` (in Spanish)

money to do the transfer. This simplifies the process of ensuring that those rules are fullfilled after every operation, and also it avoids the repetition of such validations in the UI code, which would be a possible source of errors and inconsistencies.

2.3 Firing Events from the Domain Model

We will focus on bidirectional bindings, i.e. in which any of the two properties can fire events that update the other one. This allows that UI to automatically show a change in the underlying domain object. For example if we are showing an Invoice to the user and allow him to add a new Item, it is a OOP best practice to have the Invoice object calculate its new total by itself. A bidirectional binding will allow us to show this total in the screen by modelling it as a calculated property and binding any kind of read-only UI component to it. This strategy avoids the need to calculate the total in the UI, which would represent a duplication of the logic already contained in the Invoice object.

To allow a bidirectional binding between view and model, it is necessary for both ends of the binding to be able to fire events when they change. This is not a problem for the UI end, given that there are many UI component libraries that fire events on different user actions. However, firing events from the domain model poses a problem, since we do not normally want to pollute our domain model with event-firing code. Therefore, there are few UI frameworks that support this kind of bindings, and it is difficult to implement this feature for web applications. To make our domain objects fire events, we need to detect each code segment in which the value of a property can be changed, and add some code to notify the event handler about the new event.

Fig. 4 shows the changes to be made to the `Account` and `Transfer` classes in order to have them fire a event every time one of their properties changes. Similar changes could be needed for the `TransferViewModel` class. This reduces the cohesion of our domain classes, because we are adding code to handle another concern different to their main purpose. Also, we have to *ensure* that an event is fired each time a property is changed; a manual approach to this is always subject to human errors.

Multiple UI frameworks solve this problem by having only *view-to-model* bindings, i.e. when the view changes the model is updated automatically by the binding framework, but not the other way around. On the other side, bidirectional bindings allow to automatically update the screen when the model changes. This is not only useful in contexts where a model can be modified from different sources, but also when calculated properties of a model are shown in the screen, because the calculated properties can change as result of an action performed by the same user. This, in turn, favors the usage of the logic contained in the domain model, instead of repeating part of it in the view itself.

If we are using *view-to-model* bindings, we already constructed a way to update models from view events performing validations and transformations. Being able to throw events also from the domain model would allow to reuse the same binding definitions, validation and transformations to implement *model-to-view* communication. Therefore, a mechanism that automatically fires events when the domain is modified would be of big help by eliminating much of the manual work in UI construction.

2.4 Consequences of Binding

A binding-based MVC implementation provides a useful strategy to manage the data collected by the UI, transformating and validating it, and transferring the results to the domain model. Strategies which impose intermediate structures to transfer data between UI and domain model usually require more bureaucratic code.

Moreover, binding encourages the programmer to empower domain objects and exploitation the logic contained in it. Even when a

```
class ObservableModel {
  val pcs = new PropertyChangeSupport(this)
}

class Account extends ObservableModel {
  private var _balance
  def balance = _balance
  def balance_=(newBalance) = {
    val old = balance
    balance = newBalance
    pcs.firePropertyChange("balance", old, _balance)
  }

  ...
}

class Transfer extends ObservableModel {
  def amount_=(newAmount) = {
    if (newAmount < 0)
      throw new InvalidAmountException
    else
      val old = _amount
      _amount = newAmount
      pcs.firePropertyChange("amount", old, _amount)
  }

  ...
}
```

Figure 4. Modifications to the `Account` and `Transfer` classes (only the changed methods are shown).

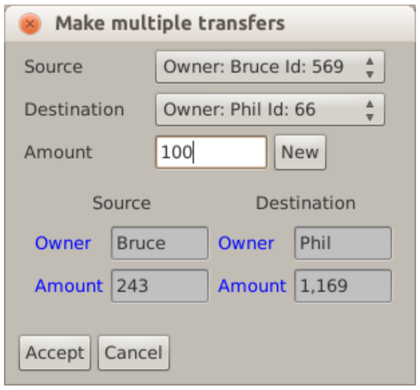
domain rule affects navigation, the UI code should not contain domain logic. Questions like "should this action be allowed?", "is this property mandatory?", "which values are valid for this property?" should be answered by the domain model. The only responsibility of the UI is to communicate the rules to the user, not to decide if a rule is being accomplished. Binding encourages this division, for example by allowing a domain object to validate itself *while* it is being edited. Besides, the possibility of automatically updating the UI when a domain object changes, allows to take advantage of the logic contained in the domain. Opposite to that, strategies with intermediate *data transfer* structures (usually named DTOs, i.e. *Data Transfer Objects* [14]), produce *anemic domain models* [8], among other *code smells* [13].

On the downside, a straightforward implementation of binding would modify domain objects while they are being edited. This can be undesired in many contexts, for example when the edition can still be cancelled by a user action, rejected by a program validation or because other users could be seeing the changes before they are confirmed. Several techniques have been created to overcome this problem, but all of them have their drawbacks.

In DTO-based techniques, intermediate data structures are used to hold the data that has to be transferred between view and model. In this approach, the DTO can be bound to the view and store the data while it is edited. Also, after confirmation of the operation another set of bindings could be used to automatically transfer the data to the domain model. The main problem is that DTO-based techniques inhibit the exploit of domain logic from the UI, i.e. the logic in domain objects can only be used after the operation is confirmed; if we wanted to use some of these logic in the view to aid the user, it probably had to be repeated outside the domain model.

Another common technique is to *copy the domain objects before edition*, i.e. before starting the edition domain objects are copied and after the edition either the original domain object is replaced by the copy, or the original object's internal state is replaced by the copy's internal state. Contrasted to DTOs based techniques, this one has the advantage of allowing the exploit of domain object behavior for the UI. Nevertheless, none of these processes is easy when we have objects with complex internal states or a graph of related objects to be edited. If we have to edit a set of related objects, we will need to know exactly which objects to copy, and replicate that part of the object graph.

To illustrate this problem we extend our transfer example to have a unique dialog that allows to perform multiple transfers atomically. Fig. 5 shows this new dialog. To implement the logic behind this dialog we would like to update the balance of the user accounts each time he adds a new transfer. In this way we are able to reuse the validations implemented in the Transfer and Account classes. Also we require to be able to cancel all the selected transactions together. Fig. 6 shows a base implementation of the `MultipleTransfersDialog` class and its view model, which does not address these problems. Neither DTO-based nor copy-based techniques can provide an adequate solution to this example.


Make multiple transfers

Source: Owner: Bruce Id: 569
Destination: Owner: Phil Id: 66
Amount: 100 [New]

Source		Destination	
Owner	Bruce	Owner	Phil
Amount	243	Amount	1,169

[Accept] [Cancel]

Figure 5. Multiple Transfer Dialog

A variant to this is to *copy on demand*, i.e. use aspect-oriented programming to intercept object modification and create a copy before allowing it. This approach would require a machinery similar to the ideas described in this work. In either case, replacing an object requires knowing all the references to it, which is not a common feature in many languages.

2.5 Transactions

Nowadays many applications have the need to implement a transactional behaviour. The applications have to perform their use cases as transactions, enforcing the concepts of ACID [17] .

A common strategy is to rely on the use of database transactions, because many database management systems implement this feature. In this approach the application performs all the changes to the model and the persistent engine is the responsible party for tracking all the changes to the database inside the transaction and perform the commit or the rollback of the operation. Every use case is performed inside a Database Transaction, so in every use case the transaction has to be opened and after its execution it has to be commited or aborted. This way of working is very simple because the domain model does not know anything about the handling of the transaction, only it has to clearly establish when the transaction starts and when the transaction ends. This information can be performed in a declarative way so the programmer does not need to implement it by hand, as in architectures based on *Enterprise JavaBeans* [40].

This approach has a major drawback: all the objects in a failed transaction have to be discarded, because the changes are rolled

```
class MultipleTransfersViewModel {
  val transfers = List[Transfer]
  val currentTransfer = new Transfer
  def possibleOrigins: List[Account] = ...
  def possibleDestinations: List[Account] = ...

  def addTransfer = {
    currentTransfer.execute
    currentTransfer = new Transfer
  }

  def confirm = ...
  def cancel = ...
}

class MultipleTransfersDialog extends Dialog {
  val model = new MultipleTransfersViewModel

  def createContents(mainPanel) {
    new ComboBox
      .bindItems { model.possibleOrigins }
      .bindValue { model.currentTransfer.source }
    new ComboBox
      .bindItems { model.possibleDestinations }
      .bindValue { model.currentTransfer.destination }
    new TextBox
      .bindValue { model.currentTransfer.amount }

    new Button
      .caption = "New"
      .onClick { model.addTransfer }
    new Button
      .caption = "Accept"
      .onClick { model.confirm }
    new Button
      .caption = "Cancel"
      .onClick { model.cancel }
  }
}
```

Figure 6. Base implementation of the `MultipleTransfersDialog`

back in the database but not in the domain objects. This problem can even get worse if the application has a rich interface in which domain objects are bound to UI components. Another problem this solution presents, is the time and the resources the database needs to handle the transaction. Many times failed use cases would not even have the necessity of reaching the database, but they reach it only to exploit its transaction support.

Other possible implementations, such as memory transactions, do not provide a clear integration with the rest of the application architecture and therefore keep most of the problems of the database solution. The integration between the UI and the transactional behaviour is not well solved by any of the solutions stated above. They demand the programmer to perform the integration by hand and numerous tasks need to be done before the desired integration can be achieved, generating error prone and repetitive code.

3. Proposed Solution

The main objective of this work is to enhance MVVM / MVB architectures by combining it with a transparent object-level transaction mechanism. To achieve this objective, we require two aspect-oriented tools, namely the *Observable Aspect* and the *Transactional Aspect*. These are two well-known uses of aspect-oriented programming, which have been implemented in many different technologies. The following two sections respectively describe the requirements we pose on each of these tools. Afterwards, Sect. 3.3

describes the key part of our approach, which is the integration of both aspects into a UI framework.

3.1 Observable Aspect

The *observable aspect* has the responsibility for firing events each time an object is modified. The objects which changes are monitored by the observable aspect are known as *observable objects*. Combined with a MVB framework, the observable aspect helps keeping the model and the view synchronized.

The observable aspect associates a set of Listeners [15] to each property of each observable object. These associations are kept in a global *listener registry*. The listeners forward the received events to the binding mechanism of the MVB framework, allowing to bind the value of the listened property to a UI component. In other implementations of MVB, the programmer has the responsibility of manually notifying the binding mechanism about the changes in the properties of domain objects. The purpose of implementing an observable aspect is to release the programmer from the need to write this *boilerplate* code.

To achieve its objective, the observable aspect has to intercept *in a transparent way* all changes to the internal state of an observable object and generate the corresponding events. By this means the integration between the model and the view is achieved without imposing modifications to the the domain model code.

3.2 Transactional Aspect

The *transactional aspect* allows to control the modifications to the internal state of domain objects and their visibility. An object whose internal state is controlled by the transactional aspect is named a *transactional object*.

We also define a *transactional context*, which is a conceptual unit of work [14]. Every access to a transactional object has to be associated to a transational context. The transactional aspect intercept all accesses to a property of any transactional object and gives control to the transactional context. In this way, we are able to (a) take notice of the modifications done in a transactional context, in order to roll them back if needed, and (b) isolate the transactional contexts, i.e. operations in a transactional context are unable to see the modifications done concurrently in other transactional context.

There are three operations that manipulate the transactional context itself: *beginTransaction*, *commit* and *rollback*. The *beginTransaction* operation creates a new transactional context. The *commit* operation confirms the modifications done in the context and publishes them, so that they are visible to other contexts. The *rollback operation* automatically discards all modifications. As expected, both commit and rollback finish the current transactional context.

The transactional context allows multiple operations in the same application to manipulate the same object concurrently with a *read commited* isolation level.

Our approach also requires *nested transactional contexts*, i.e. we can start a new transactional context inside an existing one. We say that the first transactional context is the *parent context* of the second one. This allows the division of a transaction into separated parts that can be commited or rolled back individually. When a child transaction is commited, the modifications done in the context of that transaction are published to its parent context but remain invisible to other, unrelated contexts.

3.3 Integrating both aspects together and with the UI

Using both aspects together raises subtle issues that have to be addressed properly. It is also necesary to study how the aspects will interact with the UI components. Our strategy to solve this problems is defined by the following three actions.

First, every time that the UI starts an operation that might later be cancelled by the user, we associate the execution of that operation with a transactional context. In our solution, we achieve this by adding specific UI components that interleave the operations of *beginTransaction*, *commit* and *rollback* with the code specific to the operation requested by the user.

Second, we associate bindings and events to their corresponding transactional contexts. When a binding between the view and the model is established, the model observer holds a link to the transactional context associated to the view. Then, when an event is fired by the observable aspect, it is associated to the current transactional context. These *transaction-aware* bindings will ignore any event coming from a different transactional context other than its own.

Finally, we need to pay attention to the commit process. When a transaction is commited all the changes made during the transaction are copied to the parent context. For each of these changes, an event has been fired, but these events were confined to the child transactional context. If there are views associated to the parent context, they have *missed* the events in the child context. When we propagate a change from a context to another, we have to fire the event again, in the new context. So, in the commit process, besides propagating the changes from the child context to the parent one, we have to reproduce their corresponding events.

4. Sample Implementation

The framework is implemented in a mix between Java and Scala [31] and can be used for applications written in either both languages. We assume that it can be used in applications programmed in any JVM language if it complies with the JavaBeans convention [12]; we have tested our assumption succesfully with XTend [9] and Groovy [2].

We decided to implement our own versions of the observable and transactional aspects, in order to simplify integrating them together. These tools are named *Pure Observable Objects* (POO) and *Pure Object Transactions* (POT), which respectively implement the ideas depicted in Sect(s). 3.1 and 3.2. Aspect-Oriented Programming is achieved in a self-developed lightweight AOP-framework called *Aspects for Pure Objects* (APO).

It is not the objective of this paper to provide a full description of the implementation of these tools, since they are well-known examples of AOP applications. Still, an overview is needed in order to fully understand the subtleties of the integration (c.f. Sect(s). 4.1 and 4.2 respectively). Afterwards, Sect. 4.3 describes the central part of the implementation, i.e. how we integrate all these tools together. For a detailed discussion about the selection of the AOP framework see Sect. 6.

All our tools are released under the terms of the MIT License and available at `http://xp-dev.com/svn/uqbar/projects`. A full description of the implementation can be found in [7].

4.1 Observable Aspect Implementation

The observable aspect uses the APO framework to produce an event each time a domain object is modified. More specifically, the aspect affects the classes which have the `Observable` Java Annotation. To achieve this, each field write is intercepted, i.e. every expression of the form `this.<fieldName>= <expr>` is transformed by the weaving process in order to notify the `EventManager`. Fig. 7 shows the code template used to transform field writes.

The `EventManager` coordinates the event handling mechanism. It keeps the registry of the listeners for each field of each observable object, and has the responsibility of creating the event objects and dispatching them to the interested listeners. Fig. 8 shows a simplified version of the `EventManager` class.

The method `EventManager.default` provides access to the unique instance of this class. The implementation allows to change

```
    val oldValue = this.<fieldName>
    this.<fieldName> = <expr>
    EventManager.default.valueChanged(
        this, "<fieldName>",
        oldValue, this.<fieldName>)
```

Figure 7. Code template used by the observable aspect to transform field writes in observable objects

```
object EventManager {
  var default = new EventManager
}

class EventManager {
  var listeners =
    new HashMap[(Object, String), Set[Listener]]

  def listenersFor(obj, fldName) =
    listeners.getOrElseUpdate((obj, fldName),
                              Set())

  def addListener(obj, fldName, listener) =
    listenersFor(obj, fldName).add(listener)

  def valueChanged(obj, fldName, oldval, newval) =
    if (oldval != newval) {
      val event = createEvent(...)
      for (l <- this.listenersFor(obj, fldName)) {
        l.valueChanged(event)
      }
    }

  def createEvent = ValueChangeEvent
}
```

Figure 8. EventManager code (simplified)

the event manager in order to provide a customized implementation. This feature will be used in order to integrate it with the transactional aspect (c.f. Sect. 4.3).

The code inserted by the observable aspect sends the `valueChanged` message to the event manager, indicating the modified object, the name of the modified field and the old and new values of the field. The `valueChanged` method creates an event and notifies all the registered listeners. In order to be notified when a property of an object changes, the interested component has to implement the `Listener` interface and register itself through the `addListener` method.

4.2 Transactional Aspect Implementation

The transactional aspect also uses the APO framework to intercept accesses to the internal state of objects. It affects the objects marked with the `Transactional` Java Annotation. In this case, both field reads and field writes are intercepted, allowing the intervention of the TransactionalContext. Field writes (i.e. expressions of the form `this.<fieldName>= <expr>`) are replaced by using the following template:

```
  ctx.fieldWrite(this, "<fieldName>", <expr>)
```

`ctx` is a method injected by the aspect to transactional objects, which retrieves the current `TransactionalContext`:

```
  def ctx = TransactionalContext.current
```

Also, field reads are transformed, i.e. `this.<fieldName>` is translated to:

```
  ctx.fieldRead(this, "<fieldName>")
```

Fig 9 shows the code of the `TransactionalContext` class and its `TransactionalContext` *companion object*[4]. The transactional context object keeps a record of the changes done during the transaction. Its `fieldWrite` method registers the new value of the field in this record. When the `fieldRead` method is invoked, it first looks for the value in the record and if its not found it delegates to the parent context.

The parent chain always ends in a `NullContext`. We achieve this by associating an instance of this class to each new thread; when a transaction is created in the thread, it will have the `NullContext` as parent. The `fieldWrite` and `fieldRead` methods of the null context access the transactional object's fields by reflection. This provides the same behavior as if we had no transactional aspect.

Unlike the `EventManager`, there are multiple instances of the `TransactionalContext` class. To have a global point of access to the current transactional context, we associate it to the executing thread, using a `ThreadLocal` object, which is provided by the Java platform. The expresssion `TransactionalContext.current` gives access to the transactional context associated to the current thread. The companion object also provides global methods for handling transactional contexts. The `beginTransaction` method simply creates a new context and stores it as the current context. The `commit` and `rollback` methods remove the current context and restore its parent as current context. The commit operation delegates in the context itself to propagate its changes to its parent. On the other hand, the rollback operation simply discards the transactional context and all the changes contained in it.

4.3 Integration

This section provides the implementation details for the strategy defined in Sect. 3.3. First we cover the integration of each aspect into the UI framework; then we describe the problems which arise when using both aspects at the same time, and the solutions we have found for these problems.

The integration with the observable aspect is fairly simple, we only have to provide a `Listener` implementation that listens to POO events and adapts them to the needs of the UI framework. By implementing similar connectors we can integrate the POO tool with other UI frameworks.

The integration with the transactional aspect leads to more interesting problems, since we have to decide how to delimitate transactions. We decided to create a new subclass of Arena `Dialog`, creating a `TransactionalDialog`, which automatically handles the transactional context. In other UI frameworks with more primitive UI components, transactional contexts might have to be handled manually (c.f. Sect. 6). Fig. 10 shows a simplified version of the `TransactionalDialog` class.

When a transactional dialog is opened, it creates a new transactional context. Since Arena Dialogs already have *Accept* and *Cancel* buttons, we only need to have them invoke the respective POT actions (*commit* and *rollback*). The default `accept` method method of a `Dialog` also invokes the domain action associated to the dialog, so it is a good place to catch domain exception and rollback the transaction. We defined a special type of exception (`UserException`), which the domain code shall use to notify problems that have to be informed to the end user.

[4] Companion Objects are the way to have *class methods* in Scala [30].

```
 class TransactionalContext(parent) {
  var values = new Map[(Object, Field), Object]

  def fieldRead(obj, fldName) =
    values(obj, fldName) match {
      case Some[value] => value
      case None => parent.fieldRead(obj, fldName)
    }

  def fieldWrite(obj, fldName, value) =
    values(obj, fldName) = value

  def commit =
    for (((obj, fldName), value) <- values)
      publish(obj, fldName, value)

  def publish(obj, fldName, value) =
    parent.fieldWrite(obj, fldName, value)
}

class NullContext {
  def fieldRead(obj, fldName) =
    obj.getClass.getField(fldName).getValue(obj)

  def fieldWrite(obj, fldName, value) =
    obj.getClass.getField(fldName)
              .setValue(obj, value)
}

object TransactionalContext {
  val _current = new ThreadLocal

  def current = {
    if (_current.get == null)
      _current.set(new NullContext)
    _current.get
  }

  def current_=(newContext) =
    _current.set(newContext)

  def beginTransaction =
    current = new TransactionalContext(current)

  def commit = {
    val tx = current
    current = current.parent
    tx.commit
  }

  def rollback =
    current = current.parent
}
```

Figure 9. TransactionalContext class, simplified

```
class TransactionalDialog extends Dialog {
  override def open = {
    TransactionManager.beginTransaction
    super.open
  }

  override def accept = {
    try {
      super.accept
      TransactionManager.commit
    } catch {
      case e: UserException =>
        TransactionManager.rollback
        this.display(e)
      case e: Exception =>
        TransactionManager.rollback
        throw e
    }
  }

  override def cancel = {
    TransactionManager.rollback
    super.cancel
  }
}
```

Figure 10. `TransactionalDialog` class (simplified)

```
  class TransactionalEventManager {
    extends EventManager {

    def createEvent = TransactionalValueChangeEvent

    def addListener(obj, fldName, listener) =
      super.addListener(obj, fldName,
        new TransactionalListener(listener))
  }

  class TransactionalListener(delegate: Listener) {
    extends Listener {

    val ctx = TransactionalContext.current

    def valueChanged(evt) =
      if (this.ctx == evt.ctx)
          delegate.valueChanged(evt)
  }
```

Figure 11. TransactionalEventManager code (simplified)

As explained in Sect. 3.3, integrating both aspects together requires that we restrict the events produced by POO, so that they only reach the listeners in the same transactional context. This is done by extending the event manager of Arena and by adding a filter that discards the events from other contexts. Fig. 11 show a simplified version of the `TransactionalEventManager` class.

The new event manager introduces two modifications to the original one. First, we change the type of events that we fire; these new *transactional events* keep track of the transactional context in which they have been produced. Then, the `addListener` method is *overridden* in order to decorate [15] each listener with a `TransactionalListener`. The transactional listener also keeps track of the transactional context in which it was created. When the listener receives an event, we compare the context of the event with the context of the listener and only if they both share the same context we propagate the event to the *decorated* listener. All other events are just ignored as they occurred outside the transactional context of the listener[5].

Last, we need to pay attention to the commit process according to the ideas described in Sec. 3.3. To do so, we extend the `TransactionalContext` object in order to take care of the events produced during the transaction. This is achieved by overriding the method `publish` on the transactional context, as shown in Fig. 12. The pattern used to notify the `EventManager` is very similar to that one used in the field writes of observable objects (c.f. Fig. 7).

[5] There could be other strategies to select the events, but a comprehensive comparison of these strategies goes beyond the objectives of this work.

```
class ObservableTransactionalContext {
  override def publish(obj, fldName, value) =
    val oldValue = parent.readField(obj, fldName)
    super.publish(obj, fldName, value)
    EventManager.default.valueChanged(
        obj, fldName, oldValue, value)
}
```

Figure 12. Extensions to the TransactionalContext

It is important to notice that the `commit` operation *first* changes the transactional context and then asks the old context to commit its changes (c.f Fig. 9). In this way, the `publish` method is invoked in the new context (parent of the old one) and therefore the events will be taken into account by the listeners of this new context.

Another important remark is that the observable aspect has to be weaved *before* the transactional aspect because it introduces a field read (c.f. Fig. 7). If we would apply the aspects in the wrong order, this field read would not be affected by the transactional aspect, leading to an unexpected behavior.

5. Example Application

Using our approach simplifies both the domain logic and the UI. The `Transfer` and `TransferViewModel` classes remain unchanged from the original version, except for the annotations added (c.f. Fig. 13). Since these objects are transactional, we are free to modify them, because changes will not be visible outside the transaction; also, in case of an error, the changes will be automatically rolled back. In case of an error, the framework will trap it and *rollback* the transaction, leaving all of the domain objects in their original state.

```
@Observable @Transactional
class Transfer { ... }

@Observable @Transactional
class TransferViewModel { ... }
```

Figure 13. Code fragment of the Transfer class

To take advantage of the integrated transactional mechanism, we should have our dialog extend the `TransactionalDialog` class, as shown in Fig. 14. The `acceptAction` method is a hook provided by the `TransactionalDialog` class to define the domain code to be invoked when the *Accept* button is pressed.

```
class TransferDialog extends TransactionalDialog {
  def acceptAction = model.transfer.execute

  ...
}
```

Figure 14. Modifications to the `TransferDialog` class.

6. Discussion

One important goal of our approach is to minimize impact in the application code. Regarding this objective, the two most visible points of impact of our approach in the rest of the system are *transaction delimitation* and the *selection of observable and/or transactional objects*. We consider that the responsibility of a transactional framework is to implement transactional behaviour, but it is not possible to decide when a transaction starts or ends. Therefore, transactions has to be delimited either by the application code itself or by other architectural components. In our sample applications, transactions have been delimited by the components that integrate our aspects with the UI framework.

The selection of the objects to which the aspects are applied to is done using *Java Annotations* [1]. Annotations provide a simple configuration, yet they are invasive because they force us to modify the domain classes. The selection of observable/transactional objects is independent from the rest of the approach, and could be replaced by a less invasive configuration if desired.

With regard to implementation, one important question is the selection of an AOP technology. We considered two AOP frameworks: *Javassist* [4] and *AspectJ* [23]. Despite the fact that AspectJ is a very powerful tool for AOP programming, we decided that its *compiler-based weaver* is not suitable for the level of transparency we aimed to achieve. Using AspectJ would require the programmers using our tools to replace its favorite developing environment for one supporting AspectJ. On the other hand, Javassist uses a *classloader-based instrumentation* which is less invasive to client programmers. Still, Javassist is a very low-level framework which makes more difficult to implement complex aspects. Therefore, we decided to implement a light AOP framework on top of it, to simplify the design of our two aspects. This framework is called Aspect for Pure Objects (APO) and has been released as an independent tool.

Another point to take into account is bringing this kind of ideas closer to industrial applications. Particularly, STM is a well-known technique in the research field, but it has little diffusion in the software industry. We think that many industrial applications could take advantage of STM-related techniques.

Whenever the idea of intercepting field accesses in industrial applications appears, it frequently arises a concern about *efficiency*. Our current implementation provides us with very promising results, showing only a 5% to 6% of performance penalty for big transactions. Systems with a heavy load of very small transactions can have a bigger performance penalty. Still, we have already succesfully applied a previous version of our ideas in a big financial system, with almost 20000 classes, one milion lines of code, and thousands of requests per second. Although variable interception is not penalty-free, the memory-based in-process nature of our framework makes it (by far) more efficient than any database-based transaction implementation.

Another industrial concern is the availability of adecquate tools for testing and debugging aspect-oriented code. To address this problem, we developed a *transaction monitor* integrated to Arena, which helps debugging of applications by allowing the programmer to see currently open transactions, which domain objects are affected by transactions and which properties have been modified in each context. We think that this kind of tools is crucial to allow new technologies to be adopted in industrial developments.

7. Related Work

This paper is built upon the idea of building application integrating the transactional and the observable aspects. Implementations for both aspects are well known in the industry. However, to our knowledge the integration of both aspects has not been studied yet.

For the implementation of the *Transactional Aspect* most of the work is directed to the use of database transactions and the use of STM [38]. The use of database transactions mostly follows the definition of transactions present in SQL [10], and all the implementation is left to the underlying database. There is a great number of studies about STM implentation. The DSTM2 [21], HybridTM [6] and the Software Transactional Memory for Dynamic-Sized Data

Structures of Herlihy et. al. [20] present the means to perform the required transactional aspect.

The implementation for the *Observable Aspect* has been explored by Massey [27], Gossman [16] and Smith [39], by showing different ideas of how to keep the view and the model synchronized. These works have expanded the MVC idea including bidirectional binding, as required by our solution for the observable aspect.

8. Conclusions and Future Work

Nowadays, there is a great number of tools for UI development, but still there exist some problems which are frequent in industrial developments and which are not addressed by these tools. When these problems arise, the programmers have to implement *ad-hoc* solutions. These solutions imply facing common problems in a manual way, writing many lines of repetitive code, which are error prone.

In this work, we are facing two routine problems with the UI: the synchronization between the view and the model, and the possibility to model atomic operations. Binding-based variations of MVC, like MVVM and MVB provide an interesting solution to the synchronization problem. Combining these ideas with STM greatly broadens their applicability and allows to solve both problems in a *transparent* way, i.e without requiring changes in the domain code.

Our approach has shown to be applicable to a big set of possible domains. We have tested our approach against multiple domains, many of which are by far more complex than the examples shown here. All these examples can be found online at `http://xp-dev.com/svn/uqbar/examples/ui/arena/` and prove our approach to be a *generic solution*, with industrial applicability.

The integration of the transactional and observable aspects the Arena UI framework, have improved its utility as a tool for the teaching of UI construction. In the early versions, the students needed to raise events in a manual way, which forced us to explain complex concepts and distracted the attention from the topics of the subject.

As a future work we plan to integrate different strategies to resolve the conflicts between transactions. The current version does not detect the possible conflict which arises when two or more transactions commit changes on the same object. Also, we are looking forward to adding different isolation levels, like in SQL-99 [10], and support for collections in the observable aspect.

Another path of development is a web version of the Arena framework. Our intention is that the same view description will work for both the web and standalone versions of the framework. The most difficult issue is to implement the transactional and observable aspects across the network. Finally, we are working on improving view definitions by using an XText-based DSL.

Acknowledgments

We want to thank Carlos Lombardi, Débora Fortini, Ana Larramendy and Flavia Fortini for their collaboration discussing these ideas and reviewing preliminary versions of this work.

References

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java programming language*. Addison Wesley Publishing Company, 2006.

[2] K. Barclay and J. Savage. *Groovy Programming; An Introduction for the Java Programer*. Morgan Kaufmann, 2006.

[3] S. Burbeck. Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc), 1987. URL `http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html`.

[4] S. Chiba. Javassist—a reflection-based programming wizard for java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming*, pages 92–115, 1998.

[5] D. B. Computing and D. Benyon. Domain models for user interface design. In *In Benyon*, 1996.

[6] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *12TH International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOSXII)*, 2006.

[7] R. De Jesus. Objetos puros observables y transaccionales. `http://xp-dev.com/svn/uqbar/research/papers/TIPRonnyDeJesus12-12-12.pdf`, 2012.

[8] B. Dudney, S. Asbury, J. K. Krozak, and K. Wittkopf. *J2EE antipatterns*. Wiley, 2003.

[9] S. Efftinge. Xtend language reference, 4.1, 2006.

[10] A. Eisenberg and J. Melton. Sql: 1999, formerly known as sql3. *ACM Sigmod record*, 28(1):131–138, 1999.

[11] C. Elliott. Declarative event-oriented programming. In *Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '00, pages 56–67, New York, NY, USA, 2000. ACM. ISBN 1-58113-265-4. . URL `http://doi.acm.org/10.1145/351268.351276`.

[12] R. Englander. *Developing Java Beans*. O'reilly, 1997.

[13] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[14] M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, and R. Stafford. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[16] J. Gossman. Introduction to model/view/viewmodel pattern for building wpf apps, 2005.

[17] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15:287–317, 1983.

[18] R. Harris and R. Warner. *The definitive guide to SWT and JFace*. Apress, 2004.

[19] A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[20] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III. Software transactional memory for dynamic-sized data structures. In *22nd ACM Symposium on Principles of Distributed Computing*, 2003.

[21] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997. URL `http://dblp.uni-trier.de/db/conf/ecoop/ecoop97.html#KiczalesLMMLLI97`.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with aspectj. *Commun. ACM*, 44(10):59–65, 2001. URL `http://dblp.uni-trier.de/db/journals/cacm/cacm44.html#KiczalesHHKPG01`.

[24] G. E. Krasner, S. T. Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[25] Y.-y. LIU and Z.-l. LIU. Pojo and lightweight frameworks overview. *Journal of Wuhan Institute of Shipbuilding Technology*, 2:018, 2008.

[26] R. Lämmel and S. P. Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003*, pages 26–37. ACM Press, 2003.

[27] S. Massey. Design patterns in zk: Java mvvm as model-view-binder, 2012.

[28] B. A. Myers and M. B. Rosson. Survey on user interface programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 195–202. ACM, 1992.

[29] S. Northover and M. Wilson. *Swt: the standard widget toolkit, volume 1*. Addison-Wesley Professional, 2004.

[30] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. The scala language specification, 2004.

[31] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, Mountain View, CA, 2008. ISBN 978-0-981-53160-1.

[32] R. Pawson. Naked objects. *Software, IEEE*, 19(4):81–83, 2002. ISSN 0740-7459. .

[33] M. Potel. Mvp: Model-view-presenter the taligent programming model for c++ and java. *Taligent Inc*, 1996.

[34] H. C. Qiuhui. Study on mvc model2 and struts framework. *Computer Engineering*, 6:109, 2002.

[35] T. Reenskaug. Models - views - controllers. Technical report, Technical Note, Xerox Parc, 1979. URL `http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html`.

[36] T. Schummer. Acid transaction. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 21–26. Springer US, 2009. ISBN 978-0-387-39940-9. URL `http://dblp.uni-trier.de/db/reference/db/a.html#X09o`.

[37] G. Seshadri. Understanding javaserver pages model 2 architecture. *JavaWorld. com*, 12:29–99, 1999.

[38] N. Shavit and D. Touitou. Software transactional memory, 1995.

[39] J. Smith. Wpf apps with the model-view-viewmodel design pattern. *MSDN magazine*, (2009), 2009.

[40] J. P. Sousa and D. Garlan. Formal modeling of the EJB component integration framework. Technical Report CMU-CS-00-162, Carnegie Mellon university, Sept. 2000.