A Framework for Command Processing in Java/Swing Programs Based on the MVC Pattern

A. Naderlinger, J. Templ

ACM International Conference on Principles and Practice of Programming In Java
(PPPJ 2008),
Modena, Italy,
Sep. 9 - 11, 2008

C. Doppler Laboratory
Embedded Software Systems
University of Salzburg
Austria

A Framework for Command Processing in Java/Swing Programs Based on the MVC Pattern

Andreas Naderlinger
C. Doppler Laboratory Embedded Software Systems
University of Salzburg
Jakob-Haringer-Str. 2
5020 Salzburg
Austria
andreas.naderlinger@cs.uni-salzburg.at

Josef Templ
C. Doppler Laboratory Embedded Software Systems
University of Salzburg
Jakob-Haringer-Str. 2
5020 Salzburg
Austria
josef.templ@cs.uni-salzburg.at

ABSTRACT
We present a framework for command processing in Java/Swing programs based on the model-view-controller (MVC) pattern. In addition to standard approaches our framework supports (1) centralized exception handling, (2) premature command termination, (3) pre- and postprocessing of commands, (4) undo/redo based on event objects and model listeners, and (5) generic undo/redo commands. The framework has been applied successfully in a number of graphical editors as part of a tool chain for real time programming. It proved to increase the quality of the software by eliminating local exception handlers and by confining the impact of undo/redo to a small add-on to the model part of the application.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *Frameworks, Patterns.* D.2.5 [**Software Engineering**]: Testing and debugging – *Error handling and recovery.* D.2.13 [**Software Engineering**]: Reusable Software – *Reusable libraries.*

General Terms
Design, Reliability, Languages.

Keywords
Framework, MVC, Command, Pattern, Exception, Undo, Redo, Java, Swing

1. INTRODUCTION
As a result of the broad adoption of programming languages with guaranteed initialization of variables, static and dynamic type checking, array bounds checking, and automatic garbage collection (e.g. Java, C#), one might expect that nondeterministic program behavior is a matter of the past.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPPJ 2008, September 9-11, 2008, Modena, Italy.
Copyright 2008 ACM 978-1-60558-223-8/08/0009...\$5.00

However, we still observe that in many large Java programs with a graphical user interface there are situations where the user is surprised by the program behavior. After some desperate mouse clicks, the only solution is to quit and restart the application. The program behavior appears to be as unpredictable as the behavior of C or C++ programs with dangling pointers or other memory errors. Unless race conditions are involved, we suspect that most cases of unpredictable behavior can be traced down to the proliferation of exception handlers throughout the program code. If exceptions are handled locally or even dropped silently by a deeply nested method call, follow up problems and exceptions might arise at an unrelated part of the program. Using exceptions in a disciplined way is absolutely crucial for a smooth user experience and also helps structuring the program code and avoiding code replication.

Some may argue that a user should never be confronted with an exception thrown by a program and that it is better to catch exceptions than to display them to the user. We argue that in the regular case there should not be any exception at all, hence the user is not bothered with exception handling. But if an exception occurs, we prefer to notify the user about the problem rather than silently dropping the error message or logging it to some invisible output stream.

We encountered the above mentioned problems in the course of developing Java-based graphical editors as part of a tool chain for hard real-time programming. As we neither found appropriate support in the Java library nor applicable standard design patterns, we decided to design our own command processing framework that supports centralized exception handling at its core. Another problem we encountered concerns the support for undo/redo functionality in an application based on a graphical user interface (GUI). This feature has the potential of introducing large portions of replicated program code and/or undesired dependencies between components if not done properly. It turned out that both problems are related to some degree, which led to the command processing framework described in the subsequent sections.

2. MVC AND COMMAND PATTERN
The separation of data and its representation is one of the essential characteristics of GUI-based applications. The model-view-controller (MVC) pattern, an architectural pattern widely used in software engineering, reaches back to Smalltalk [1]. MVC splits an application into different parts, namely the application's data model (M), the views (V) on this data model, and the controllers

(C) that affect the model, typically as a response to user interaction.

Figure 1 shows a refined representation of this trisection. The model is the central element displayed by an arbitrary number of unknown views and affected by arbitrarily many, in principle completely independent, controllers. A close 1:1 relationship between views and controllers as in the original Smalltalk framework is unnecessarily restrictive although it exists in many UI frameworks [2].

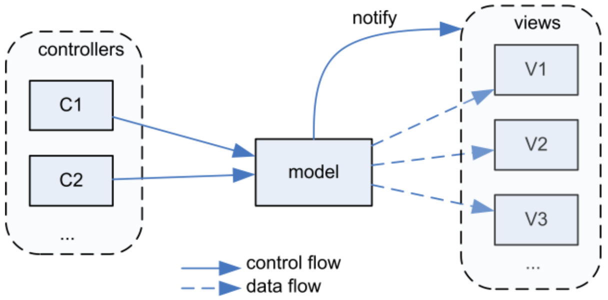

Figure 1. MVC pattern

The model comprises the domain specific data together with operations for reading and writing model data. Views are representations of the model. They are informed about model changes by some notification mechanism (notify) that allows for decoupling the model and its views (e.g. Observer pattern [3, page 293]). Controllers perform write operations on the model that, in turn, notifies all views about the state change. Typically, a controller is associated with a GUI control, such as a button or menu item.

For applications with a graphical user interface, the *Command pattern* as proposed by [3, page 233] is a popular approach for implementing the controllers. Each user request is encapsulated in a `Command` class that implements a base interface with a single method `execute()`. This architecture obviously decouples the GUI framework, which hosts the command, from the implementation of the request. The traditional command pattern does not address problems related to exception handling. The command pattern as found in the Java library also lacks an exception handling strategy as shown in more detail in the subsequent chapters.

2.1 Command Pattern in Java

Java provides support for the command pattern by means of the AWT delegation event model. A user control such as a button provides a registry for listeners on the *action performed* event, which is fired whenever the user clicks the control or activates it by a keyboard input. The action listener, which must implement the interface `java.awt.ActionListener`, plays the role of the command class. The method `actionPerformed()` in the ActionListener interface is the analog to the method `execute()` in the command pattern. Swing defines an extended interface named `Action`, which introduces a set of command attributes such as icon, label, keyboard shortcut, etc. The abstract base class `AbstractAction` serves to maintain the Action attributes. Figure 2 shows the relation between the pattern (a) and its realization in AWT (b) and Swing (c). The extensions introduced in (d) will be explained below.

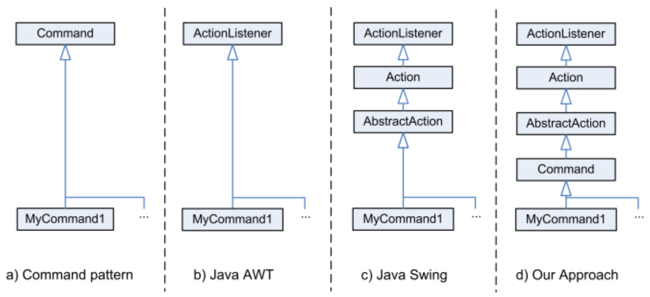

Figure 2. Command patterns

Since it is recommended and common practice to use the Swing library for building GUI-based applications in Java, we designed our command pattern for Swing only. However, the principal ideas could also be applied to AWT.

2.2 Exception Handling Problems in Swing Actions

It seems obvious and in fact it is common practice that every concrete command (e.g. `MyCommand1`) is a direct subclass of `AbstractAction`. However, this approach ignores an important aspect, namely exceptions. Exceptions are neither explicitly considered in the command pattern nor are exceptions part of the interface declaration of the Java delegation event model. The ActionListener interface defines a method `actionPerformed()` to be implemented by every command class.

```
public void actionPerformed(ActionEvent e);
```

This method signature does not allow the command implementation to throw a checked exception and thus forces any command to provide an exception handler to catch at least all checked exceptions. This leads to code replication as no central exception handling mechanism exists. Additionally, it tempts one to handle exceptions locally in the model itself. Indeed, this is regularly done although the model cannot know how to properly deal with exceptions.

Unchecked exceptions are even trickier as one might easily forget about them completely. The standard implementation logs unchecked exceptions to the standard error output, which might not be visible to the user. Since Java 5 there has been a documented way of changing the default reaction to exceptions and errors by setting a thread’s exception handler. However, this is a quite far-reaching intervention and might be inappropriate or even forbidden by a security manager if the application (or component) shares its Java VM with others.

2.3 Adding Exception Handling in Actions

In our framework, we introduce an abstract class `Command` as a subclass of the `AbstractAction` class. The implementation of the method `actionPerformed()` delegates to a new abstract method `onExecute()` and handles any kind of exception. This allows for centralizing the exception handling and thereby avoids code replication. The class `Command` is used as common base class for any concrete command class. In addition, we removed the event parameter from the signature. In the rare cases where the event parameter is required, we make it available by means of a getter function.

```
public abstract class Command extends AbstractAction {
  public abstract void onExecute() throws Exception;
  public EventObject getEvent() { … }
  …
}
```

Any exception thrown during command processing will terminate the command and be handled at the very top of the call chain in the class `Command`. There should not be any local exception handlers inside the model, view, or controller implementations (unless forced by the Java library, which occurs in some rare cases). In addition, we encountered situations where command processing should be terminated in a controlled way, which can be achieved by throwing a special kind of exception. We identified the following three categories of exceptions:

- An exception may be thrown in order to terminate a command execution in a controlled way and the user should be notified. Therefore, we introduce a `TerminateException` that is typically used to report user input errors for which we know in advance that they may happen. The default handling of this type of exception is to display the exception's title and message in a modal dialog box.
- An exception may be thrown to terminate a command execution silently. Therefore we introduce a `CancelException`. It is used to stop the command processing when the user has explicitly cancelled the current command. By default, the handler for this kind of exception is empty.
- We regard any other kind of `Throwable` (`Exception` or `Error`) as unintended and their occurrence as an error in the program code or runtime system. In the default exception handler implementation these abnormal command terminations are displayed in a modal dialog box that can optionally display the complete stack trace.

2.4 Adding Pre- and Postprocessing to Actions

In order to allow for application specific actions related to command execution and exception handling we introduced a simple mechanism for command pre- and postprocessing, called a command `Processor`, which can be registered with the command framework. Command processors have to implement the interface `Processor` with the two methods `preProcess` and `postProcess`.

```
interface Processor {
  void preProcess(Command cmd)  throws Exception;
  void postProcess(Command cmd) throws Exception;
}
```

Every registered command processor is executed each time before (`preProcess`) and after (`postProcess`) the actual command is executed. Processors are maintained as weak references in order to prevent memory leaks. They can be used, for example, to log command execution, as additional exception handlers (according to [4], exception logging is a major concern of many applications), for collection of statistical command execution time, or for checking licensing aspects.

2.5 Sharing of Commands

As a Swing action represents a Java bean with bound decoration properties [5], a single command instance can be shared between multiple GUI objects such as a push button, a menu item, or a corresponding toolbar button. If a command is disabled, for example, all GUI elements that share this command will be disabled as well. Sharing of commands also means that no matter how a command is invoked by the user it executes exactly the same shared code.

2.6 Adding more Listeners

Unfortunately, not all activities initiated by the GUI user can be handled by an `ActionListener`. Selecting a node in a tree or in a list, for example, may also need some code to be triggered but it is not possible to register an action listener for this type of activity. The class `JTree`, for example, expects a `Tree-SelectionListener` and the class `JList` expects a `List-SelectionListener`. In order to extend the applicability of commands, we define class `Command` to additionally implement the interfaces `ChangeListener`, `PropertyChange-Listener`, `TreeSelectionListener`, `List-SelectionListener`, and `Runnable`. The corresponding event object is made available by means of a getter function. For any remaining cases where a local exception handler is still required, class `Command` provides a static method that implements the default exception handling strategy. A `WindowListener`, for example, cannot be mapped directly to the `onExecute` method because it requires multiple handler methods. Please refer to chapter 4 for a discussion on the number and type of remaining local exception handlers that we have encountered in our approach. A possible solution to the problem with multi-method listeners is described in the Future Work section.

3. UNDO/REDO

In a mature interactive application, the user expects to be able to *undo* and possibly later *redo* the effect of executing a command. Depending on the application's context and on the support for multiple users, different approaches are preferable. The so-called *restricted linear model* [6] has become the de facto standard for single user applications. All executed commands are put on an undo stack. The user can only undo the last command on the stack (*linear*). Undoing a command removes it from the undo stack and places it on the redo stack. Redoing a command removes it from the redo stack and pushes it again on the undo stack. When a new command is added to the undo stack, the redo stack is cleared (*restricted*).

The implementation of the restricted linear model is simple in principle. Nevertheless, there are different ways of implementing it in detail and it has the potential for affecting the whole application structure and for introducing a lot of code replication, which we tried to avoid in our framework.

3.1 Conventional Undo mechanism based on Commands

The widely cited standard implementation for undo/redo is based on the Command pattern [3, page 233] mentioned in section 2. Command classes are extended by an undo- (and possibly a redo-) method which reverses the model modification performed in the method `execute()`. Additionally, in order to reverse the performed operations, every nontrivial concrete command implementation needs access to the model's current state information. The tempting and straightforward approach for realizing undo is to store the state information directly in the command objects. In many cases this will expose parts of the internal structure of the model and violate the principle of encapsulation. Furthermore, model specific data structures are likely to be replicated in the command classes. Consequently, source code changes in some model operation that involve adjustments in the model's state have to be precisely followed in each command class that makes use of this state information. Typically, the relations between model operations and user commands are not obvious, and thus the software is difficult to maintain. Attempts to make state information transparent to commands, such as applying the Memento pattern [3, page 283], relax the coupling between model and commands. However, in many cases it only shifts the problem of code duplication to the model. Additionally, the model complexity increases, as supplementary (memento) classes have to be introduced.

We followed an approach that does not implement undo/redo on the level of commands, but on finer-grained model operations. Commands simply execute a sequence of model operations. If these model operations are invertible, commands are invertible as well without any further coding.

3.2 Undo/Redo Mechanism based on MVC

The proposed undo/redo mechanism is based on an event notification architecture and consequently is expected to integrate well with applications based on the model-view-controller (MVC) architecture. The core idea of MVC is to decouple model and views, which is established by means of a notification mechanism (e.g. Observer pattern [3, page 293] aka Publish/Subscribe). In case some model operation causes the model to change, all registered views are notified in order to adjust their state. The information on the particular kind of model change is encapsulated in a so-called event object. This object contains any required information needed to ensure consistency between the model and its views.

Figure 3 shows the relation between commands and events. A single command may produce an arbitrary sequence of events. Undoing a command may be considered as undoing the effect of every single event produced by a command in reverse order. Thereby the size of the code required for undo is roughly proportional to the number of different events, not to the number of different commands, which may be a significant difference. The same calculation holds for redo. In addition, if we regard events as the unit of undo/redo, we can implement undo/redo support as part of the model, not as part of the controllers, which allows us to keep the model together with undo/redo functionality in its own package without the needs of publishing internal details.

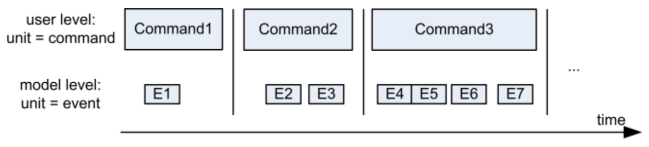

Figure 3. Commands and Events

In our approach, events form the basic unit for undo and redo. Each event represents a model change, and each event *knows* how to undo resp. redo itself.

```
public abstract class UndoableEvent extends EventObject {

  public abstract void undo() throws Exception;
  public abstract void redo() throws Exception;

}
```

All model events must subclass `UndoableEvent` and they must implement `undo` and `redo`. Therefore in addition to information about the new model state, undoable events have to store information on previous values, too. In most cases it suffices for the `undo` and `redo` implementation to invoke a single model operation because this approach operates on fine-grained model operations.

3.3 A Reusable UndoManager Component

The generic undo/redo functionality is provided by the framework's `UndoManager` component. It provides two managed commands (`undoCmd` and `redoCmd`) that can be used directly, for example, as an application's menu items for undo/redo. The commands are managed in the sense that they adapt their *label* and *enabled* state to the current situation. In addition the `UndoManager` provides methods for adding an undoable event, starting and ending a named event sequence for grouping of multiple events to a single entry in the undo/redo mechanism, and for resetting the undo/redo state.

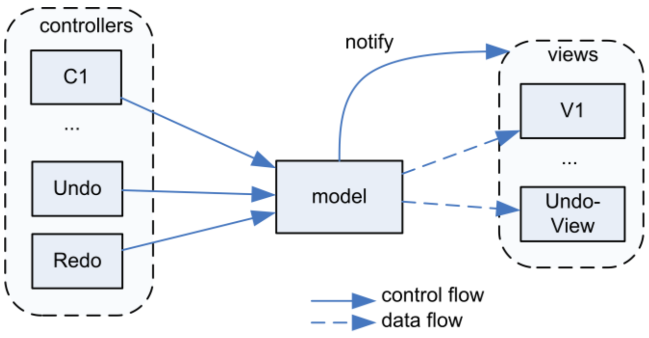

Figure 4. MVC based Undo/Redo Support

For connecting the UndoManager component to a particular application model, a subclass of the UndoManager must be created which implements the event listener interface of the application model by simply delegating all relevant events to its base class. Conceptually, this subclass acts as just another view which is registered as a model listener. Instead of providing a visual representation of the model, this view keeps track of all fired events and manages the undo and redo commands. shows the resulting MVC model including controllers for undo and redo and a view that serves to maintain the undo/redo state of the model.

The following code listing is a complete example for a subclass of the UndoManager component:

```
class UndoView extends UndoManager
  implements MyModelListener {

  UndoView(MyModel model, Component owner) {
    super(owner);
    model.addMyModelListener(this);
  }

  public void handleEvent(MyEvent e) {
    if (e.id == MyEvent.SEQ_BEGIN) {
      beginSequence(event.caption);
    } else if (e.id == MyEvent.SEQ_END) {
      endSequence();
    } else {
      addEvent(e);
  }
}
```

Figure 5 exemplifies the interaction between the actors starting from the application's user. Events that are fired as a consequence of model operations and which indicate a change in the model's state are observed by all registered views including the `UndoView`. As mentioned above, invoking a single model operation may lead to several events. To achieve a correspondence between user actions (executed as commands) and undo operations, events can be grouped to labeled sequences, which can be nested arbitrarily. When the user performs an undo (redo), the events in the top of the undo (redo) stack are executed in the appropriate order and the undo and redo stacks are adjusted accordingly. While performing undo (redo), the undo manager ignores any events received from the model (disableEvents, enableEvents). This allows us to *reuse the same model operations* for normal command processing and for undo/redo and thereby avoids code duplication.

4. STATISTICS

We have evaluated two applications which are part of a tool-chain for real-time application development and which use the presented framework for command processing, exception handling and undo/redo. The evaluation gives information about the number of exception handlers that remained for our applications and identifies the reasons for their existence. Additionally, we measured the total code size and the code portion required for undo and redo in terms of lines of code.

Both applications are graphical front-ends for the development of real-time applications based on the Timing Definition Language (TDL). The first application, called TDL:VisualCreator [16], is a graphical editor for specifying the timing behavior of TDL components in a platform independent way. This tool is seamlessly integrated in and two-way synchronized with the simulation environment MATLAB®/Simulink® from The MathWorks [17] with which it shares the same Java VM. The second tool, called TDL:VisualDistributor, allows one to configure the target system's topology and to map TDL

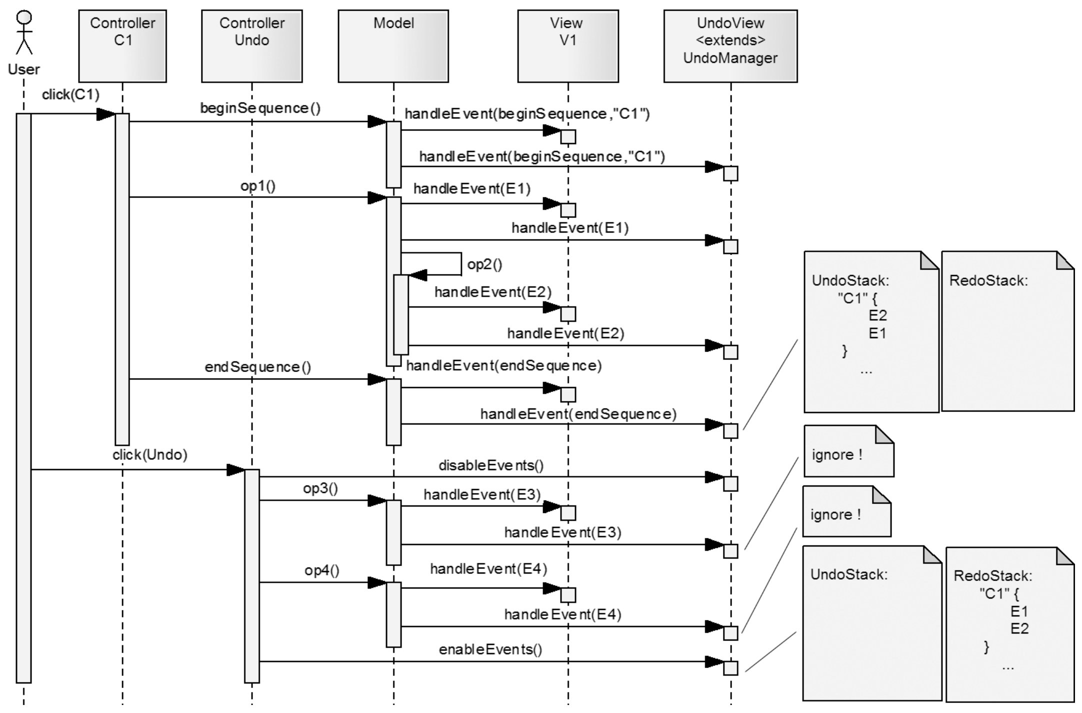

Figure 5. Sequence Diagram

components to target nodes. It is a graphical configuration front-end for code generators.

4.1 Exception Handlers

We ran a text search for *catch* blocks on the source code of the mentioned applications and manually classified the exception handler code. The following categories have been identified:

Catch & throw. Exceptions are caught and immediately re-thrown, for example in order to map checked to unchecked exceptions or to provide user-friendly error messages by mapping a `NumberFormatException` to a `TerminateException`.

Top-level exception handlers. We consider exception handlers to be at top-level of an application if there is no possibility for the developer to catch exceptions at a higher level in the call hierarchy. This is trivially the case for try/catch blocks surrounding the main method of the application or the run method of any thread. Furthermore, we rank exception handlers within AWT event listener methods among this category, as such event listeners act as callbacks of the AWT event dispatcher thread, which is not under control of the application developer. Top-level exception handlers are required as a consequence of the AWT delegation event model, whenever it is not feasible to use class Command. The `awt`, `awt.dnd` and `swing` packages in the Java library contain 40 different interfaces that extend the `EventListener` interface. The ratio of those which only declare a single method for handling an event is 40%. These interfaces can theoretically be covered by the `Command` class and its `onExecute` method. When implementing any of the other interfaces (e.g. `MouseListener`, `TreeModelListener`, `WindowListener`) an exception handler is required that forwards the exception to the centralized exception handler provided by class Command.

Local exception handlers. This third category comprises catch blocks that are at a lower level of the call hierarchy. Exceptions are handled locally without being re-thrown and thus are not propagated to a top-level handler. In certain cases, the Java library forces developers to rely on exceptions as it provides no facilities to check, for example, the validity of method arguments. There is nothing like `Integer.isInteger(String)`, for instance, that could obviate the possibility for a `NumberFormat-Exception` when trying to convert a string to an integer.

Table 1 gives an overview of the individual exception handling categories by listing the number of exception handlers for the evaluated tools. Compared to the lines of code (LOC) (see Table 2), we consider the number of exception handlers, in particular local exception handlers, to be pleasantly small. Almost every user request is implemented as a command which performs the centralized top-level exception handling for free. The rather increased number of top-level exception handlers in TDL:VisualCreator is caused by the Java interfacing mechanism provided by MATLAB®. Three local exception handlers in TDL:VisualCreator are forced by Java's threading mechanism, which requires one to catch the `InterruptedException` after a `wait()` or `sleep()` call. Four local exception handlers in TDL:VisualDistributor are due to a plug-in mechanism that deals with extension classes by using `Class.forName()`, which throws an exception if a class cannot be loaded. One is due to the optional support of `System.getenv()` for configuring extension classes and one is due to ignoring failed look-and-feel selection during Swing startup.

Table 1. Number of exception handlers

Application	catch & throw	top-level	local
TDL:VisualCreator	8	30	5
TDL:VisualDistributor	15	9	6

4.2 Code size

Table 2 gives a summary of both tools in terms of the code size expressed by lines of code (LOC – without comments or blank lines). For the evaluation we used the eclipse plug-in Metrics[1].

Table 2. Code size in LOC

Application	Total	Model	Undo/Redo	UndoView
TDL:VisualCreator	20393	4865	457	44
TDL:VisualDistributor	10530	1673	246	32
Framework	442		214	

The undo and redo related code accounts for about 2.5% of the application in total for the TDL:VisualCreator and about 2.6% for the TDL:VisualDistributor. Measured by the application model, the ratio is about 10% and 17% respectively. Apart from the UndoView, a subclass of UndoManager that represents a view of the model data, the entire code that is used for undo/redo is located in the applications' model packages.

Unfortunately we cannot provide a comparison such as LOC of the individual applications before and after applying the proposed framework, which would be especially interesting for the undo/redo related code. We directly applied the presented mechanism when we extended the TDL:VisualCreator with undo support. The TDL:VisualDistributor supported undoable commands from the beginning. Table 3 lists the number of implemented user commands and gives an estimation of the number of LOC we saved simply because of centralizing the exception handler. This avoids at least 4 LOCs for the try/catch clause for each command.

Table 3. Number of commands and LOC saved

Application	Commands	Saved LOC for exception handling
TDL:VisualCreator	52	≥ 208
TDL:VisualDistributor	45	≥ 180

The savings easily compensate for the part of the framework required for command processing (442 – 214 = 228 LOC). Note that we did not count the code required for displaying exceptions in dialog boxes because this code would also be required without the framework.

[1] http://metrics.sourceforge.net

5. RELATED WORK

Since the pioneering work of Goodenough [11] numerous results have been published in the field of exception handling. Cabral et al. [4] have performed a comprehensive study on how exception handling mechanisms are used in practice by analyzing more than thirty applications, 16 of them were developed in Java. Their results show that exception handlers are typically not tailored to specific errors but perform generic operations like logging or user notification. Garcia et al. [12] evaluate and compare various mechanisms for exception handling implemented in different languages and propose an *ideal* exception handling model. Robillard et al. refine previously published concepts for Ada and present guidelines for *designing robust java programs with exceptions* [13]. Approaches for handling exceptions with aspect oriented programming are presented in [14, 15].

There are a few projects, for example [7], that provide improved command processing for Java applications. However, to our knowledge, there is no framework for Swing-based applications that covers command processing in combination with undo support.

In [3], Gamma et al. describe an undo mechanism based on coarse-grained commands, which is described in section 3.1.

Wang and Green [8] elaborate on the difficulties at applying recovery mechanisms to object-oriented software architectures and propose a framework that shifts a large part of history management to the individual objects.

In the Java Foundation Classes (JFC), we encountered a set of classes that are intended to ease the implementation of an undo mechanism based on event listeners (`javax.swing.undo`), which looks similar to our approach at a first glance. However, besides being much more complex in the number of interfaces and classes involved, the undo notification mechanism is orthogonal to model change notification, which results in code duplication. This can be seen by studying the implementation of the Swing text editor component, which makes use of this undo mechanism. In addition, the Swing undo mechanism does not support centralized exception handling and it does not provide generic undo and redo commands. Also JGraph [18], a graph visualization Java library, uses the Swing undo package. Again, undo and model change notification is implemented orthogonally. In addition, the JGraph core model gets quite complex as it strongly depends on this set of classes.

The work described in [9] contains a description of a fine-grained approach for adding undo support to JavaBeans without modifying existing code by using automatic code generators.

6. OUTLOOK AND FUTURE WORK

Besides exception handling problems, we observe inconsistencies in the handling of enabled/disabled state changes of GUI controls (grey-logic) in many large Java program. If a program contains a large number of controls and the grey state depends on a number of parameters, it becomes difficult to update all controls at the right times and it spreads the code for implementing the grey-logic across the whole program. As a solution to this problem we envision an extension of the Command concept by a boolean function called a *guard*. If the guard evaluates to true, the control is enabled, otherwise it is disabled. In the ideal case, calling these guard functions is done by the framework and not in the program code.

The problem of supporting WindowListeners or, in general, listeners with more than one handler method as mentioned in Section 2.6, might be solved by providing special subclasses of Command, e.g. WindowCommand. These Command classes act as adapters for the corresponding listeners, i.e. they provide an empty implementation of the implemented interface. Of course, we could apply the principle of subclassing also to all interfaces which are currently implemented by class Command. This would result in an ActionCommand, a ListSelectionCommand etc., and it would simplify the base class Command at the expense of introducing a number of subclasses.

7. CONCLUSION

We presented a framework for Java/Swing applications to overcome shortcomings in the current AWT/Swing library with respect to exception handling in command processing. Furthermore, we presented the integration of a light-weight mechanism for undoing and redoing commands based on undoable event objects that serve to undo/redo elementary model operations as opposed to undoing the effect of a command as a whole. The approach fits particularly well with applications based on the MVC pattern and has been applied successfully in a number of graphical editors for the TDL tool chain. It helped us to eliminate almost all local exception handlers, which increased the software quality and at the same time reduced the code size. Undo/redo support has been added to our editors with only a very small fraction of the code required for the rest of the applications and without any unintended structural changes or violations of abstraction boundaries.

The source code of our Command framework is available on request via e-mail.

8. ACKNOWLEDGEMENTS

The command framework presented in this paper has been inspired partially by utility classes developed as part of an earlier project of one of the authors [19]. This subproject, however, is unpublished and it did not include support for undo/redo. The grey-logic support as proposed in the Future Work section has been prototyped in the mentioned subproject and was originally inspired by the BlackBox Component Builder framework [20].

We would like to thank Johannes Pletzer, Wolfgang Pree and Gerald Stieglbauer for carefully proofreading this paper. We would also like to thank the anonymous reviewers for their comments and suggestions. They helped us to improve the presentation considerably.

9. REFERENCES

[1] S. Burbeck. *Applications Programming in Smalltalk-80: How to use Model-View-Controller*, 1987.

[2] A. Fowler. *A Swing Architecture Overview*, http://java.sun.com/products/jfc/tsc/articles/architecture

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* ISBN 0-201-63361-2, Addison-Wesley, 2002.

[4] B. Cabral and P. Marques. Exception Handling: A Field Study in Java and .NET. *In ECOOP'07: Proceedings of the 21st European Conference on Object-Oriented Programming*, pages 151-175, Berlin, Springer, 2007.

[5] Sun Microsystems: *The JavaBeans Specification 1.01*. http://java.sun.com/products/javabeans/docs/spec.html

[6] R. Mancini, A. Dix and S. Levialdi. Reflections on Undo. Technical Report RR9611, University of Huddersfield, 1996, http://www.comp.lancs.ac.uk/~dixa/papers/undo-techrep-96/tech9611.pdf

[7] Brico, http://sourceforge.net/projects/brico

[8] H. Wang and M. Green. An Event-Object Recovery Model for Object-Oriented User Interfaces. *In UIST'91: Proceedings of the 4th annual ACM Symposium on User Interface Software and Technology*, pages 107-115, New York, 1991. ACM.

[9] H. Washizaki and Y. Fukazawa. Dynamic Hierarchical Undo Facility in a Fine-Grained Component Environment. *In CRPIT'02: Proceedings of the 40th International Conference on Tools Pacific*, pages 191-199, Darlinghurst, Australia, 2002.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The JAVA Language Specification*. Sun Microsystems, Inc, Mountain View, California, USA, 2000.

[11] J. B. Goodenough. Exception Handling: Issues and a Proposed Notation, *Commun. ACM,* 18(12):683-696, 1975.

[12] A. F. Garcia, C. M. F. Rubira, A. B. Romanovsky, and J. Xu. A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. *Journal of Systems and Software*, 59(2): 197–222, 2001.

[13] M. P. Robillard and G. C. Murphy. Designing Robust Java Programs with Exceptions. *In FSE'00: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2000. ACM.

[14] F. C. Filho, A. Garcia, and C. M. F. Rubira. Error Handling as an Aspect. *In BPAOSD'07: Proceedings of the 2nd Workshop on Best Practices in Applying Aspect-Oriented Software Development*, New York, 2007. ACM.

[15] M. Lippert and C. V. Lopes. A Study on Exception Detection and Handling using Aspect-Oriented Programming. *In ICSE'00: Proceedings of the 22nd International Conference on Software Engineering*, pages 418-427, New York, 2000. ACM.

[16] TDL tool suite, http://www.preeTEC.com

[17] The MathWorks, http://www.mathworks.com

[18] JGraph, http://www.jgraph.com

[19] J. Templ. The KITE Application Server Architecture. *In Lecture Notes in Computer Science*, Volume 2789/2003, ISBN 978-3-540-40796-6, pages 37-48, 2003. Springer.

[20] BlackBox, http://www.oberon.ch/blackbox.html