Kaj Björklund

A Serialization Library with Undo Support

Master's Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology

Espoo, 10th April 2005

Teknillinen korkeakoulu	**Helsinki University of Technology**
Tietotekniikan osasto	**Department of Computer Science and Engineering**

HELSINKI UNIVERSITY OF TECHNOLOGY	ABSTRACT OF THE MASTER'S THESIS
Author:	Kaj Björklund
Title of thesis:	A Serialization Library with Undo Support
Date:	10th April 2005
Number of pages:	75
Department:	Department of Computer Science and Engineering
Professorship:	T-106 (Software Technology)
Field of study:	Software Systems
Supervisor:	Prof. Eljas Soisalon-Soininen
Instructor:	Prof. Eljas Soisalon-Soininen

Agile development and usability are frequently appearing concepts in contemporary software development. While focusing on these areas results in major gains for the customers and end-users, they place the application developer in front of an increased set of challenges. Designing an architecture which embraces usability and continuous change requires considering the effort of the application developer as well.

Serialization is an architecturally sensitive feature, which is useful for implementing saving and loading, among other things. The ability to undo is a major usability requirement. Undo allows the users to recover from their mistakes and explore the application without being afraid of breaking something.

In this thesis, the means available for implementing object serialization and undo in interactive graphical user interface applications are studied. Commonly acknowledged requirements, models and techniques for both serialization and undo are investigated. Furthermore, existing implementations are reviewed.

A library for implementing serialization in the C++ programming language is presented. Then extensions for semi-automatic single-user linear undo support are given. The design is based on a generic serializable memento, which allows capturing and restoring the state of an object and serializing the captured state.

The presented library is analyzed against the background theory and it is shown to reduce programmer effort particularly in the case of undo and be useful for agile projects. Also experiences of deploying the library to two commercial project management applications are discussed and several directions for improvements and further studies are given. Finally, the thesis is concluded with a summary of the presented topics.

Keywords: serialization, undo, memento, persistence, agile development

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Kaj Björklund
Työn nimi:	A Serialization Library with Undo Support
Työn nimi suomeksi:	Kumoamista tukeva sarjallistamiskirjasto
Päivämäärä:	10. huhtikuuta 2005
Sivuja:	75
Osasto:	Tietotekniikan osasto
Professuuri:	T-106 (Ohjelmistotekniikka)
Pääaine:	Ohjelmistojärjestelmät
Valvoja:	Prof. Eljas Soisalon-Soininen
Ohjaaja:	Prof. Eljas Soisalon-Soininen

Ketterä kehitys ja käytettävyys ovat usein esiintulevia käsitteitä nykyisessä ohjelmistokehityksessä. Näihin tekijöihin keskittyminen tuottaa suurta hyötyä asiakkaille ja loppukäyttäjille, mutta samanaikaisesti ne tuovat sovelluskehittäjille uusia haasteita. Käytettävyyden ja jatkuvan muutoksen huomioivan arkkitehtuurin suunnittelu vaatii myös kehittäjien työmäärän puntarointia.

Sarjallistaminen on arkkitehtuurisesti herkkä ominaisuus, joka on hyödyllinen muun muassa tallentamisen ja lataamisen toteuttamisessa. Mahdollisuus kumota tehty toiminto on merkittävä ominaisuus käytettävyyden kannalta. Kumoaminen tarjoaa käyttäjille keinon selvitä virheistään sekä tutkia sovellusta tarvisematta pelätä rikkovansa jotakin.

Tässä työssä tutkitaan menetelmiä olioiden sarjallistamisen sekä toimintojen kumoamisen toteuttamiseen interaktiivisissa sovelluksissa, joissa on graafinen käyttöliittymä. Erityisesti selvitetään yleisesti hyväksyttyjä vaatimuksia, malleja ja tekniikoita sekä sarjallistamiseen että kumoamiseen, mutta myös olemassaolevia toteutuksia tarkastellaan.

Työssä esitellään kirjasto sarjallistamisen toteuttamiseen C++-kielellä. Lisäksi kirjastoon esitetään laajennuksia, jotka lisäävät siihen puoliautomaattisen yhden käyttäjän lineaarisen kumoamistuen. Kirjaston suunnitelma perustuu yleiseen sarjallistuvaan muistoon (*memento*), joka mahdollistaa olioiden tilan vangitsemisen ja palauttamisen, sekä vangitun tilan sarjallistamisen.

Esiteltyä kirjastoa verrataan taustatyössä esiteltyihin menetelmiin, jolloin käy ilmi että kirjaston käyttö vähentää sovelluskehittäjän työtaakkaa erityisesti kumoamisen tapauksessa. Osoittautuu myös, että kirjasto soveltuu ketteriin projekteihin. Myös käyttökokemuksia kirjaston soveltamisesta kahteen kaupalliseen projektinhallintasovellukseen esitellään. Jatkotutkimuksille annetaan useita suuntaviivoja. Diplomityö päätetään yhteenvetoon läpikäydyistä aiheista.

Avainsanat: sarjallistaminen, kumoaminen, muisto, pysyvyys, ketterä kehitys

Acknowledgements

I had the opportunity to select the area for this Master's thesis according to my own interests. Those interests have been motivated by my work at Dynamic System Solutions (DSS) Ltd[1] since the year 2000 to the present date. I also want to thank DSS for the study leave and financial support they have provided me.

The chance to spend time on studying the research conducted on the areas that I encounter during my daily work has been rewarding. It has also been a pleasure to observe that my university studies have helped me understand and have opinions on the topics discussed in academic literature.

This thesis has been produced using the fpTeX distribution of the LaTeX document preparation system. Notepad++ was used to edit the LaTeX source files. The figures have been drawn using the Dia[2] [1] diagram creation program. The Concurrent Versions System (CVS) repository for the LaTeX source files was manipulated using the TortoiseCVS utility through a PuTTY powered Secure Shell (SSH) connection. The Portable Document Format (PDF) file was procuded using GhostScript. All of the tools used are freely available online. I wish to thank the authors of these wonderful programs for their efforts. This thesis is also available online at `http://www.iki.fi/kbjorklu/mthesis/`.

I want to thank my friends Barış Boyvat, Markus Jakobsson, Sampo Nurmentaus, Ilkka Pelkonen and Markku Rontu for beneficial discussions about various topics related to this work, and for reviewing draft versions of this thesis. I would also like to thank my professor, Eljas Soisalon-Soininen, for his valuable assistance during the writing process, and Kenneth Oksanen for his helpful feedback on this work.

Finally, I would like to thank my family and all of my friends. Hopefully we will have the chance to celebrate my graduation soon.

Otaniemi, 10th April 2005,

Kaj Björklund

[1] `http://www.dss.fi/`

[2] See also Section 3.3 for an analysis of the Dia undo.

Contents

Terminology

Agile Software Development An approach to software development, which emphasizes individuals and interaction over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation and responding to change instead of following a plan [3].

Cohesion In a high cohesion design, classes have approximately equal a number of responsibilities which are highly related to each other, and the number of responsibilities per class is relatively low.

Coupling Coupling is the degree of interaction between elements [51], that is, it measures how strongly one element is connected to, has knowledge of, or relies on other elements [36].

Design Pattern Description of a solution to a general design problem with analysis of its consequences.

Domain Model Layer In a typical layered architecture in GUI applications, the application logic is implemented in the domain model layer and visualized to the user in the user interface layer.

Encapsulation Separation of interface and implementation so that interface users do not need to know the details of the implementation. Encapsulation is a *language construct* which facilitates implementing the *design principle* of *information hiding* [10].

GUI Graphical User Interface is a way for a person to interact with a computer program through directly manipulating graphical elements as well as text.

Layered Architecture An architecture in which the logical structure of a software is organized into separate layers with distinct responsibilities so that the dependencies between different layers are restricted to avoid excessive coupling.

Linear Undo An undo model in which only the action whose execution put the application to its current state can be undone.

Memento A design pattern which allows capturing and externalizing the internal state of an object without violating encapsulation so that the object can be restored to the state later [24].

Non-linear Undo An undo model in which any action in the history list can be undone without undoing other actions.

OOP Object-Oriented Programming, a responsibility guided programming paradigm in which software systems are modeled as objects communicating with each other.

Persistence Preserving the state of a data structure between program runs.

Redo Undoing the effects of undo.

Reflection The ability for a program to inspect and modify aspects of itself at run-time. For instance, the Java programming language allows, among other things, looking up and invoking the constructor and the member functions of classes and reading and writing member variables of objects at run-time.

Schema Data schema describes how data structures are composed, ultimately from language primitives such as integers and characters.

Serialization Encoding the memory representation of a data structure to another representation, such as a flat sequence of bytes or database records. Serialization is also referred to as *marshalling*, often in the context of remote proceduce calls, and as *pickling* in certain programming language communities, such as the community of Python. In object-relational literature serializing objects is often called *dematerialization* or *passivation*.

Serialization Mechanism Specifies the output representation of serialization.

Software Architecture Architecture describes the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution [27, 34].

Undo To reverse the effects of an action made in a system.

Usability The effectiveness, efficiency and satisfaction with which users can achieve their goals [21].

XP Extreme Programming, an agile software development process [9].

Chapter 1

Introduction

Modern software development has several non-trivial objectives. This is particularly emphasized by the adoption of *agile* process models, such as Extreme Programming (XP) [9]. Software developers are expected to produce lots of new features within short time frames. As design decisions or customer requirements change, it should be possible to rapidly make even major changes to existing software, often without breaking backward-compatibility.

Usability has become increasingly important. Usability can roughly be described as the effectiveness, efficiency and satisfaction with which users can achieve their goals [21]. Indeed, it is not difficult to see why usability is considered to be such an important part of a software product by all project stakeholders. An important factor in the usability of an interactive program with a graphical user interface (GUI) is the ability to *undo* actions. This encourages users to explore without being afraid of "breaking the system".

Software architecture describes the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution [27, 34]. Designing and *maintaining* a good architecture is difficult, especially with the pressure of providing business value at full speed. It is however acknowledged that a sound architecture will pay off in the long run, both from the technical point-of-view and in terms of total cost of ownership [39].

From points-of-views of both the developers and the customers it would be convenient if the development effort could be focused on the essential, that is, the problem domain related program features which provide business value to the customer. Extreme Programming lets the customers pick the most valuable features to be done next [9], so it can be said that XP elevates the idea. So on one hand, developers should not ignore architectural issues, but on the other hand, they should not spend too much time on them.

Often when writing program code related to features which provide business value, the developers need to write a lot of mandatory code related to cross-cutting fragments of the program architecture. Such code includes support for undo, *serialization* and *persistence*. This kind of code is typically quite tedious and error-prone to write [70], is architecturally sensitive [22], affects a large portion of the code base [34] and tends to deteriorate gradually, especially if it is written in an ad-hoc manner.[1]

Object-oriented (OO) programming is a software development paradigm very popular today. It is supported by the majority of modern programming languages. The key concept in the paradigm is an *object*. Software systems are modeled as groups of objects communicating with each other. Objects are instances of *classes*, which describe the attributes and behaviors of the objects.

Object serialization is the process of encoding the memory representation of an object to another representation, such as a flat sequence of bytes or database records. Serialization can be used for persistence, that is, storing the object state across application runs, but it also has other uses.

In this work, we will study how undo and serialization can be implemented in the object-oriented paradigm. The implementation should be such that it reduces the need for everyday attention paid to these issues. Also when the application developer does need to deal with undo and serialization, the required effort should remain moderate. With these two requirements for the system, we hope that developers will not be too tempted to skip implementing undo and serialization properly even under schedule pressure. We also assume that these requirements will allow the development team to produce business value at full speed. Our objective is to present a design of a library for serialization and undo, and to show that it can be used to achieve these goals.

We will deliberately not study the details of serialization *mechanisms*, that is, how and what is done with the data after it has been extracted from the objects, apart from using the data to implement undo. Other common uses would include sending the data over a network connection to achieve remote procedure calls (RPC)[2], and persisting or exporting the data to binary flat files, Extensible Markup Language (XML) [71] files or databases.

Our goal for undo is not to design a state of the art research system, but to present an undo design which is similar in features to most modern commercial programs, and which keeps the developer effort moderate. This means that we will narrow our studies on multi-user and non-linear undo models.

This thesis is organized as follows. In Chapters 2 and 3, we will introduce background theory on both serialization and undo, respectively, introduce existing designs and analyze their features. Then, we will present our library which provides convenient serialization for objects. After that, we extend the library to support undoing actions performed on the objects. Furthermore, we analyze our library in the context of the background theory presented in earlier chapters, and discuss how the design could be improved. In Chapter 7 we present experiences of real-life case studies of two commercial project management systems. The experiences originate from the use of the library at Dynamic System Solutions Ltd[3]. Finally, we conclude the thesis with a summary of our findings.

[1]The required effort and care are further elaborated later. We also show that similar opinions are typical in literature.
[2]Object-oriented RPC is also known as remote method invocation (RMI).
[3]`http://www.dss.fi/`

Chapter 2

Object Serialization

The traditional definition for serialization is the process of turning data structures in memory and their states into sequences of bytes. These flat sequences of bytes can then be used in a variety of ways, such as storing them in permanent storage to achieve persistence across program runs, or sending them over a network connection to make remote procedure calls. In the context of modern serialization facilities this may be somewhat inaccurate, as most libraries support arbitrary *serialization mechanisms* for the extracted data. For instance, the *s11n* [8] serialization library supports text, XML, binary and MySQL[1] database mechanisms, among others. We will give our slightly modified definition of serialization in the beginning of Section 2.3.

The details of serialization mechanisms are an important topic on their own, as they affect the performance, portability and other qualities of both the application and the serialized data they produce. While we do not study the details of serialization mechanisms in this thesis, the design of a serialization library should not prevent implementing several types of mechanisms. The design should neither dictate a certain type of mechanism to be used. In particular, it is not uncommon to make the assumption that a (relational) database management system will be running under the application. However, in this thesis we will try to be neutral about the underlying mechanism, but do take into account that mechanisms will set certain specific requirements, such as unique identifiers, to the serialization system.

Some programming languages, such as *Java*[2], provide built-in support for serialization. In these languages, the application developers have to do virtually nothing to make their data structures serializable. However, these sorts of approaches typically do require developer intervention at the latest when the data *schema* changes. Data schema describes how the data structures are composed, ultimately from language primitives such as integers or characters. In object-oriented programming the class definition usually describes the schema of the object. Examples of changes in a schema are the addition and removal of data structures or their member variables and changing the types of the variables. In object-oriented programming schema changes also include changes in the inheritance relationships and moving member variables between base and derived classes. Serialization system support for schema changes is called *versioning*, and will be further explored in Section 2.3.3.

[1]`http://www.mysql.com/`
[2]`http://java.sun.com/`

Implementing serialization or persistence can be quite demanding in practice. Serialization often becomes a key player in the application architecture, which places several major requirements that need to be considered carefully. This is highly emphasized in the literature as well. Wilson and Kakkad [70] consider serialization *tedious and error-prone* to implement. Willink [67, p. 242] uses the same two words to describe implementing marshalling and unmarshalling code, and proposes the code to be generated automatically by a meta-program. Rontu [49, p. 47] observes that his serialization code is overly complex, and suggests using suitable frameworks for persistence and undo. Brown and Whitenack [13] warn that without care persistence may hurt object-orientedness and cause the project to suffer, and estimate that 25-50% of application code may be persistence related. Atkinson *et al* [6] estimate persistence to typically be about 30% of code size. Atkinson and Morrison [7] observe that building and maintaining reliable persistent applications in a timely fashion frequently proves to be much harder and expensive than expected.

In many applications, a large portion of the classes need to be serializable. In most cases, code related to serialization needs to be written for each of the serializable classes. However, the serialization code is not related to the actual responsibilities of the classes. For instance, it is realistic that a `Car` class is responsible for containing wheels, because the class models the problem domain. In the problem domain, there is nothing related to serialization, that is, serialization is an unrelated responsibility. The reason for this is that any way of partitioning the software cannot divide all logically related entities into separate modules [34]. A logical entity that cuts across the main partitioning of the software is called an *architecture fragment*. *Aspects* are a popular way to implement fragments non-intrusively.

In the next section we present approaches to implementing serialization. Then we move on to defining a set of requirements for a serialization library, and briefly discuss the topic of third party data access. Finally, we introduce existing implementations and analyze their properties.

2.1 Ad-Hoc Approaches

It is possible to directly implement serialization in an ad-hoc manner, but this tends to be error-prone and make maintenance harder. Extending the serialization code, for instance to support new serialization mechanisms can also be difficult. Ad-hoc approaches typically also have the downside of forcing all developers to become acquainted with the details of the mechanism, such as the Structured Query Language (SQL) statements involved [43].

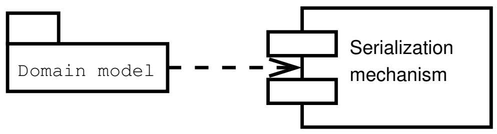

Figure 2.1: The Brute-Force Approach. The domain model classes are directly coupled to the serialization mechanism.

The most straight-forward way to implement serialization is to directly place it in the *domain model* classes. This is illustrated in Figure 2.1. This approach may be appealing when starting the implementation, since it requires very little initial effort to get serialization going and to some, may seem to promote *encapsulation*. However, this brute-force implementation has major disadvantages. The approach will couple the domain classes to the serialization mechanism, making mechanism modifications difficult. It also adds challenging unrelated responsibilities to the domain model classes, which will make them harder to understand for developers and reduces the cohesion of the code.

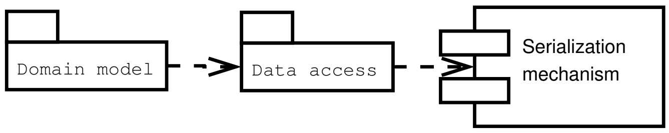

Figure 2.2: The Data-Access Objects Approach. The data access classes are directly coupled to the serialization mechanism, but reuse of domain classes may be possible.

Another approach, shown in Figure 2.2, is to separate the serialization code to other classes, known as data access objects. In this scenario, each class in the domain model will be related to another class with serialization capabilities. While this may make it possible to reuse the domain classes, and make them more readable, it will cause the number of classes to increase and still couple the implementation of the data access classes to the serialization mechanism.

2.2 The Library Approach

Serialization is most conveniently implemented using a serialization library. The benefits of the library approach include clean separation of concerns from other parts of the system, which means less coupling. This also promotes code reuse, since the serialization library can be implemented in an application-independent manner. Also, as mentioned previously, serialization libraries often support several mechanisms and abstract the details of them. This makes it easier to add new mechanisms, since the application can be written in a mechanism-independent manner.

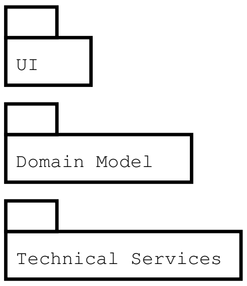

Figure 2.3: Typical Architectural Layers. Serialization should be placed in the technical services layer.

Figure 2.3 depicts a typical layered software architecture [34, 36]. The user interface layer is responsible for displaying reports and views of the domain model. The domain model implements the business rules and the logic of the application. The technical services layer provides the necessary tools, such as logging, for implementing the higher levels.

Serialization is placed in the technical services layer [4, 36, 43]. The objects in the domain model layer are made serializable using the library. The user interface layer triggers the serialization to take place, for instance, when the program user wishes to save or load, or when it is time to do an auto-save.

The three layers in Figure 2.3 are the key elements in most layered architectures. However, it is not uncommon to further split the layers at least conceptually [34, 36]. For instance, the UI layer could be divided into the presentation layer, which would handle reports and views, and to the application layer, which would control the window transitions and session state. Reusable business infrastructure could be separated from the domain model so that it could be used in several related projects. Low level technical services, such as threads, files and database connectivity are sometimes seen as another layer. However, the three layer separation is adequate for the purposes of this thesis.

2.3 Serialization Library Requirements

The key requirement for a serialization library is the ability to serialize and deserialize data structures. For the purpose of this work, we define serialization as the act of transforming data structure instances, in our case object, into a non-object representation of data, such as a sequence of bytes, intermediate structured representations or database records. Deserialization is the opposite activity [36]. In order to achieve serialization and deserialization in an extensible and usable way, there are a number of requirements which the serialization library needs to address. We discuss these requirements in the following sections.

2.3.1 Low Coupling to Domain Model

In object-oriented programming, class models should be designed to be cohesive [36]. High cohesion means that classes in the software have approximately equal a number of responsibilities which are highly related to each other, and the number of responsibilities per class is relatively low. For serialization purposes, it is commonly acknowledged that the domain model should know as little as possible of the serialization facilities. This is called *full encapsulation of persistence mechanism* [4, 43], or in the case of completely decoupling the domain class from serialization, *orthogonal persistence* [7] and *non-intrusive serialization* [47].

As explained in Section 2.1, these objectives may be difficult to achieve using ad-hoc approaches. A library designed for the purpose of serialization should enforce the policy of reducing coupling between the domain model and the serialization facilities. Implementing serialization without the serializable object being aware of it is sometimes impossible, but doable in certain cases, and the only feasible way in others. For instance, if the classes being made serializable cannot be altered, the non-intrusive approach is the only way to go. Either way, serialization will be dependent on the serializable classes.

However, it is often enough to support non-intrusive serialization optionally, and to also provide a model where the domain classes need to implement a light-weight interface dictated by the serialization library. Due to practical reasons, this will make serialization support easier to implement. It will also make the library easier for the domain application developers to use, as they do not need to consider tricky issues, such as serialization of referenced objects through base-class pointers or providing enough visibility of member variables to the serialization code. These issues are further explained in Sections 2.3.4 and 2.3.7.

Another argument against full orthogonality is that the developers of the domain model classes should have some control over the serialization process. For instance, in Figure 2.4, the `Car` class has two member variables. The value of `numberOfWheels` is calculated from the length of the linked list `wheels` every time the latter one changes. The reason for doing this might be, for instance, that the calculation takes so much computing time that it needs to be cached[3]. When designing the serialization support for `Car`, the developer might skip serializing the `numberOfWheels` variable, and recalculate it at deserialization-time. In the Java programming language this could be expressed using the `transient` keyword.

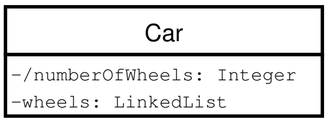

Figure 2.4: Simplified `Car` Class. The `numberOfWheels` is a derived variable which is calculated from the length of the `wheels` list.

The decision of what to serialize can either be put to the domain model class or to somewhere else, for instance to a domain model description file. This information will be dependent on the domain model in any case, because changes in the domain model will require updating the serialization descriptions. It is also questionable whether the descriptions can be written without explicit information about the internal structure and behavior of the domain class. For instance, updating the `numberOfWheels` in the previous example requires knowledge of the behavior. If and when this information is not completely available through the public interface of the domain class, the describer will have to gain additional access and insight to the descriptee class by some other means. This may either violate encapsulation or tightly couple the two classes. The same argument can be applied to other external serialization approaches. Furthermore, it can be argued that the developer of the domain class often has the best knowledge of how to write the descriptions.

As a conclusion, the design should strive towards easing the burden of serialization for the domain model, but it is not necessarily worth trying to pursue perfect orthogonalization. It can be useful to accept that serialization is a separate, cross-cutting fragment. In the next section we will explore decoupling the serialization mechanism from the domain model, which plays a major role in isolating the domain model from serialization.

2.3.2 Mechanism Decoupled from Domain Model

When the data has been extracted from the objects, it will be further processed by a component called the serialization mechanism. Mechanism is also known as *archive* [41, 47], *connection* [43] or *storage mechanism* [36]. We have chosen to refer to it as mechanism since the term better communicates that the output of the serialization process can be used for other things than archiving, storage or persistence, for example to make remote procedure calls.

[3]For instance, the complexity of walking the linked list of the example is $O(n)$.

Coupling is the degree of interaction between elements [51], that is, according to Larman [36], it measures how strongly one element is connected to, has knowledge of, or relies on other elements. Larman also gives problems that too highly coupled classes may experience:

- Changes in related classes cause local changes.
- Harder to understand in isolation.
- Harder to reuse, because their use requires the additional presence of the classes on which they are dependent.

For serialization, it would be convenient for the serializable classes in the domain model to be decoupled from the serialization mechanism [7]. This means that the serialization mechanism should not be directly implemented in the domain model classes. Then it will be possible to avoid the typical problems with too much coupling, which in the context of a serialization library can be rephrased as follows:

- Changes in serialization mechanism cause changes in the domain model classes.
- All application developers need to be familiar with the details of the mechanism.
- Using the domain model classes without the need for the implemented serialization mechanism is inconvenient, and the same applies to using them with a different mechanism.

Decoupling the domain model from the serialization mechanism can be achieved by creating an intermediate element between the domain model and the mechanism. The intermediate element should then provide interfaces for both the mechanism and the domain model classes. This setting is illustrated in Figure 2.5. The library user has written three classes, `Car`, `Wheel` and `Garage`, and made them serializable using the serialization library core. The serialization library provides three mechanisms, namely `Binary flat file`, `XML` and `Database`. In this setting, the serialization core acts as a middle-man, hiding the details of the mechanism from the domain classes, and respectively hiding the details of the domain classes from the mechanism.

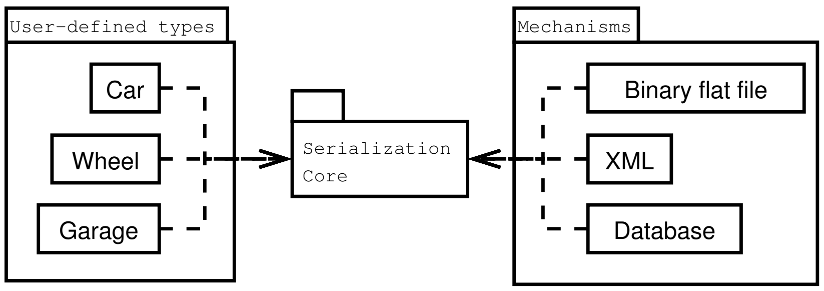

Figure 2.5: Decoupling the Serialization Mechanism. The mechanisms are separated from the domain objects by the serialization library.

As a side-note, it should be possible for the users to create their own mechanisms as well, although it is not emphasized in the figure. The interface provided by the library core can be used by the library itself to provide the default built-in mechanisms, but it should also be exposed to the library users.

With the serialization core standing between the domain model and the serialization mechanism, the logical process of serialization becomes the following:

1. Extract data from data structures to an intermediate form.
2. Apply a mechanism to the intermediate form.

Observe that there is no need for the intermediate form to be physically implemented. As explained in Section 2.5, several existing implementations support directly writing the contents of the data structures to the storage, without any intermediate in-memory representation. This can yield marginally better performance, because no intermediate data-copying is needed. It is still useful to think about the *logical* intermediate form, given that it assists in understanding the coupling issue.

Although it is not a good idea to expose the details of the serialization process to the domain model classes [7], it may sometimes still be needed for practical reasons. This will further be discussed in Section 2.3.9.

2.3.3 Versioning Support

In traditional data-oriented design techniques, the structure of the data model is first determined, and then the operations on the data are designed to fit the agreed model [51]. Such an approach is typical in projects which make the assumption that the application will have loads of data, and will operate on top of a database. Designing the data model first makes it possible to fix the database model in a very early phase of development. This is convenient since changes to the database structure tend to be challenging, in particular if existing data in the database needs to be accessible using the new system. It also makes it possible to immediately start converting data from legacy systems.

However, it is easy to predict that version 1.1 of a software will have added new features and improved previous ones, which usually means having more data or a changed data model. Also modern agile software development processes embrace rapid change [9] and argue that knowing all details of the software in the very beginning of the project is unrealistic. These issues typically make changes in the data model quite frequent.

We gave examples of data model (schema) changes in the beginning of this chapter. Supporting these schema changes can be implemented in various ways. In the context of databases, the ability to support change can be categorized as follows [48]:

Schema Modification Schema modifications to populated databases are allowed.

Schema Evolution Schema modification without the loss of existing data is supported.

Partial Schema Versioning All data can be viewed both retrospectively and prospectively through user-definable version interfaces. Updates are allowed to one designated schema version only.

Full Schema Versioning All data can be viewed and updated both retrospectively and prospectively through user-definable version interfaces.

Of the four, full schema versioning is the strongest requirement as it requires both read and write access to new versions of the data schema using old versions of the software. This can be quite difficult to support in the general case, but feasible in the case of simple data model changes, such adding a new member variable to a class. In the general case, the system may have changed so much in a later version that supporting full versioning would require explicitly writing code to maintain legacy data structures. This is not only tedious to program, but also sets new requirements to the testing process. In practice it is usually enough to support schema evolution, in which new versions of the system are able to correctly read and convert old versions of the data. We will restrict further discussion on the subject to schema evolution. Schema versioning is further discussed in [23, 48], among others.

The basic mechanism for data type evolution support is to have the library provide means for the serializable classes to detect the version of the object data at deserialization-time. Then the serializable objects can set default values to variables that did not exist in previous versions, and skip reading variables that have been dropped in newer versions. The latter can be trivially accomplished by using member variable identifiers as described in Section 2.3.5. The former may be more demanding, as there may be arbitrary dependencies between objects in a system. The version number can also be convenient for fixing invalid data caused by bugs in the old versions of the software. For instance, if the value of a member variable has been calculated incorrectly in a previous version, the deserialization function can recalculate it for old schema versions.

Sometimes setting default values to new fields requires reading values from other objects. In general, there can be no guarantee that the other objects are already in a valid state, as they may not have been deserialized yet, or even if they are, they may not have set the default values for themselves yet. Because of the possibility of cyclic data structures it is difficult to overcome this problem without the serializable classes intervention of the developer.

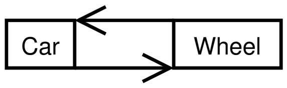

Figure 2.6: Simplified Class Diagram for `Car` and `Wheel`

For instance, there is a cyclic dependency in the class diagram shown in Figure 2.6. A `Car` may have wheels, and the programmer has decided that if a `Wheel` is in a car, the wheel knows it. These kinds of two-way associations are typical in many implementations, as they make look-ups perform better. Given this class diagram, suppose that the programmer adds a new derived member variable to the `Car` class, which represents the time it takes to accelerate the car from $0\frac{km}{h}$ to $100\frac{km}{h}$. The value naturally depends on the properties of the wheels, so their state needs to be investigated during the calculation. If it turns out that the `Wheel` class does not need any information from the `Car` class during deserialization-time, the situation can be cleanly handled by deserializing and setting default values to the `Wheel` objects before starting to set the default values to the related `Car` objects.

To clarify the previous paragraphs, the steps to deserialize objects can be summarized as follows:

1. Read the data from a mechanism to an intermediate form.

2. Instantiate data structures.
3. Restore the states of the data structures from the intermediate form.
4. Transform the restored data structures to the latest schema.

As in Section 2.3.2, these steps are to be interpreted as logical steps only. In the actual implementations steps may be combined for performance or memory consumption reasons, but it is again useful to acknowledge the abstract steps of the process. We already discussed that the order in which these steps are to be taken is important. It still remains unclear where the code for each activity should be placed in the architectural sense, and when the whole process of data model conversion should take place.

Placing the evolution code to the domain model classes is straight-forward, and justified because the knowledge of performing the conversion is best available in those classes. The drawback is that it may eventually make the domain classes bloated with evolution-related code. Externalizing parts of the code to a service of the serialization library has been proposed [23, 32, 43], but in the general case the domain model has to put in some effort as well, for instance to calculate the derived member variables. Also the encapsulation discussion in Section 2.3.1 is valid here. That is, external changes in the data of an object move the responsibility of maintaining data integrity away from the class of the object thus violating encapsulation.

In our example about `Car` and `Wheel`, we did not state when the existing data should be converted to the new format. In the context of database persistence, Clamen [17] gives three alternatives for the decision:

Emulation All interaction with old instances is done via a set of filters, which support all the new-style operations on the old-style data format. An additional feature of emulation is that old program versions can still run on the old data.

Eager Conversion A special program, which converts the old data to the new data, is executed once. The system needs to be able to access all instances of old data. This approach may also require downtime on the data instances.

Lazy Conversion Whenever the program encounters an old data instance, it converts it to the new data format. This may make accessing old instances more expensive.

As we will see in Section 2.5, lazy conversion is by far the most common approach in serialization libraries. In addition to the factors given by Clamen, this is probably motivated by the fact that with support for file-based serialization mechanisms, such as XML files, it may be difficult to locate all existing data instances. Also as we explained earlier in this section, implementing evolution-support in the deserialization code is quite straight-forward in most cases. This makes it natural to do conversion lazily.

2.3.4 Proper Handling of Pointers

In object-oriented programming objects may refer to each other by means called associations, pointers or references. The pointer graph may be cyclic and contain multiple pointers to the same object.

In the latter case the object is said to be a shared object. Also pointers do not necessarily point to objects of the compile-time types of the variables, but instead to derived class objects. That is, both pointers and the values they point to have types, and the types are not necessarily the same. The serialization library should be able to handle all these cases properly.

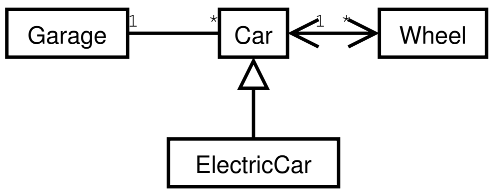

Figure 2.7: Simplified Class Diagram for `Garage`, `Car`, `ElectricCar` and `Wheel`. `ElectricCar` is derived from `Car`.

As an example, suppose that the developers are designing an application for managing garages. Cars can be parked in garages. Cars can have any number of wheels, even zero. The developers have observed that only electric cars can be recharged with electricity, so they have derived a class `ElectricCar` from `Car`, and given it the additional functionality of being rechargeable. This setting is illustrated in Figure 2.7.

If the application is running in the state shown in Figure 2.8, and the serialization library is asked to serialize the instance of `Garage`, in which there is one `Car` with two `Wheels`, the library should support automatically finding the `Car` and the `Wheels` because they can be reached from the `Garage` object by following pointers. This is called serialization by reachability. This corresponds to the transitive closure of `Garage` as defined in [25], and is known as the *transitive state* [53] of `Garage`.

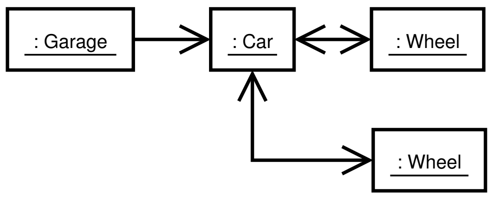

Figure 2.8: Instances of `Garage`, `Car`, and `Wheel`. There is one `Car` in the `Garage`, and the `Car` has two `Wheels`.

In Figures 2.7 and 2.8 the associations between `Car` and `Wheel` are bidirectional. This means that the serialization library will encounter the same `Car` object several times during the serialization process, since it is referenced from three different objects: one `Garage` and two `Wheels`. The library should understand that the `Car` object is shared between the other three objects and serialize it only once. After deserialization, the object graph should be exactly as in Figure 2.8 again, that is, the `Car` object should remain shared.

The bidirectional associations between `Car` and `Wheel` also mean that the object graph will be cyclic. This means that if the serialization library is careless, it will first find the `Wheel` object from the `Car` object, then it will find the same `Car` object from the `Wheel` object, then the same `Wheel` object again, and eventually crash due to stack overflow. The library must be written in a way which properly detects cycles, and is able to deserialize them correctly as well.

Assuming that `Garage` references the `Cars` it contains using pointers to the compile-time type `Car`,

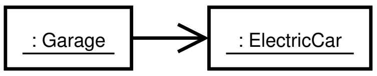

Figure 2.9: Instances of `Garage` and `ElectricCar`. An `ElectricCar` is parked in a `Garage`, but the `Garage` handles it as if it were a `Car`.

Figure 2.9 contains another pitfall, namely derived class serialization through base class pointers. The object in the `Garage` is actually an instance of `ElectricCar`, not `Car`. This should be detected during serialization so that that all the member variables of the `ElectricCar` are serialized too. Also the deserialization routine needs information about which type to instantiate. Typical way of implementing this is forcing an intrusive approach and making the serialization function virtual, that is, dynamically bound.

2.3.5 Identifiers

Objects can be described by three key properties: their classes, the values of their member variables and their identities. Serialization and deserialization should preserve these properties. As explained in the previous section, this may be somewhat tricky in the case of pointers as member variables. Cyclic pointer graphs, shared objects and serialization through base class pointers all need to be correctly handled. Tracking the identities of the objects that have already been processed and not reprocessing if they have already been encountered before is a feasible way of dealing with shared objects and cyclic graphs.

However, there is an issue about serializing pointers. When an application is running, pointers are typically implemented using memory addresses of the referenced objects. Memory addresses do not make much sense in the serialized form, because there is no telling what the memory addresses will be after deserialization[4]. This means that pointers need to be implemented by other means, namely by creating additional object identifiers, which are independent of the address space of the program. An identifier should be created for each object that is serialized[5]. The identifier can be as simple as an increasing integer value, although more complex schemes are used as well [43, 57]. Serializing the pointers with this approach becomes equal to serializing the object identifier of the referred object.

Objects are not the only elements which benefit from identifiers in serialization. It is also useful to have identifiers for member variables and classes. Having identifiers for member variables makes it easier to drop them during the course of type evolution, as explained in Section 2.3.3. Also some serialization mechanisms, such as XML and databases, require that type information is present. Generating the type information becomes easier when the member variables have explicit identifiers.

Class identifiers identify the classes of the serialized objects. Having class identifiers is needed in a serialization library, as the deserialization process is expected to instantiate the objects it encounters in the serialized form, and instantiation requires knowledge about the class of the object. As

[4]Explaining why this is so is out of the scope of this thesis, but thinking about schema evolution support and reading material about memory management should help. See for instance [69] for the non-garbage-collected case and [68] for the garbage-collected case, or `http://www.memorymanagement.org/` for a web site dedicated to memory management. With more restricted approaches, such as the pointer swizzling method explained in Section 2.6, memory addresses can be manipulated.

[5]Boost supports non-tracked objects as an optimization. See Section 2.5.

explained in Section 2.3.4, the actual run-time type of an object may be different than the compile-time type, so the class identity needs to be explicitly serialized as well. If this were not required, the compile-time type encountered by the deserialization routine would suffice, and no further class information would be mandatory[6].

2.3.6 Error handling

Serialization and deserialization are not always successful. The serialization library itself may fail due to missing or corrupted data, broken network or database connections, full disk drives and so forth. Also the code in the serializable class may fail during serialization or deserialization, for instance if the developer has decided not to support certain old data schemata.

In object-oriented programming *exceptions* should be used to report most failures. Exceptions from the serializable objects and the serialization mechanisms should not cause the serialization library to go to an invalid state, and the exceptions should be mediated to the code calling the serialization services.

2.3.7 Serializable Class Interface

The interface provided to the application programmer implementing the serializable classes should be minimal so that the code overhead caused by serialization remains low. This is important because of two reasons. First, it improves the separation of concerns, as discussed in Section 2.3.1. Second, it makes the effort required for serialization support smaller. From the agility point-of-view, this enables the developers to focus on business value producing features.

The serialization library needs to be able to read and write member variables of the objects it serializes. The developer of the serializable domain class needs to provide functions for accessing the member variables. One function is used for reading during serialization and the other for writing during deserialization. As explained in Section 2.3.5, the functions will often be dynamically bound so that the correct function is called in the case of serializing through base class pointers. Usually the functions will be member functions of the serializable class in order to guarantee enough visibility to the member variables. If the serialization functions of the serializable class are not members, there may be some problems with the visibilities of the member variables, since the variables will typically be hidden from the public interface of the class. In this case, the serializable class needs to provide additional access rights to the serialization functions, for instance by using the `friend` keyword in the C++ programming language [54], or package private variables in Java. If the serializable class derives from a base class defined by the serialization library, the access functions will be both dynamically bound and members, which avoids the above complications.

It is also worthwhile to consider combining the reading and writing functions to just one function which works as a reading function when serializing and as a writing function when deserializing. The advantage of this approach is that it optimizes the effort of the domain model developer in the most common case, in which the reading and writing functions are very similar. The drawback is that during later development, the function may become bloated with conditional statements.

[6]Boost supports omitting class information as an optimization. See Section 2.5.

During deserialization, the library needs to be able to create new instances of the serializable classes. For this, the library needs to be able to determine the classes of the objects during serialization, and it needs to have access and the ability to use at least one constructor of each such class. The class type is usually registered to a class factory with a serializable identifier, and the serializable class provides means to access its identifier.

In order to serialize pointers, the library needs to be able to figure out the identity of the objects being serialized. The identity is needed for keeping track of the mapping between the object identity in the memory and in the serialized form. This is trivial to achieve in any object-oriented programming language since pointers to objects are always supported[7], but may require further care in some cases of *smart pointers*. Smart pointers are a common concept in C++. They are used for equipping pointers with additional features such as reference counting.

A further requirement for the interface provided by the serialization library to the serializable classes is that it properly supports the data types used by the serializable classes. In practice this means that at least the built-in types of the language, such as integers, are directly supported. Also standard library types, especially containers should be directly supported by the library. Otherwise the burden of implementing serialization for these types will shift to the domain model developers, which increases the effort required from them. Additionally, the library should support extensions for supporting other types used by the domain model.

2.3.8 Simultaneously Active Mechanisms

Ambler [4] requires that a persistence layer should support several simultaneously active connections to different databases. His motivation is that this enables for instance copying objects from one database to another. The feature can also be useful for evolution purposes, as simultaneous connections can transfer objects from an older schema to a newer schema behind another mechanism [43]. For instance, moving data from a legacy or a third party data source to the production database is quite common.

In a serialization library this feature is also useful. Assuming that the code calling the library triggers serialization and deserialization, there is no reason to bind the life-time of the serialization mechanism to the library, but the callers can provide the mechanisms as parameters. This also enables auto-saving, which is a standard feature in many interactive applications nowadays.

2.3.9 Advanced Mechanism Use

It is not always necessary to deserialize the entire set of serialized objects at once. Deserializing objects on-demand basis is known as lazy deserialization, or lazy materialization [36]. The motivation is to make deserialization initially faster by skipping much of the work, and also to save memory. The trade-off is that error detection is also deferred.

Implementation can be handled in a disciplined manner by using proxies [4]. Proxies are also known as proxy objects [13, 43] and virtual proxies [36]. In C++, proxies can be conveniently implemented

[7] Although they are sometimes called references.

by a form of smart pointers, that is, pointers which behave as normal pointers, but when dereferenced they deserialize the pointee object if it is not already in memory.

Sometimes additional knowledge of the serialization mechanism can be useful. For instance the lazy deserialization is probably always implemented using a database mechanism, since databases support efficient random access to objects through queries. Implementing queries to all mechanisms, which would be required if there is no mechanism-related additional knowledge, is very difficult at best. It may also be possible that the domain model developer wants to serialize certain variables only for certain types of mechanisms.

In particular, if the serialization mechanism is a database, the developers often want to use the features it provides, which sets additional requirements to the library. Such features and requirements include transactions and concurrent access, locking, security and access control, cursors and caching. Further study of these issues is out of scope for this thesis, but for instance Nurmentaus [43] and Ambler [4] provide further information.

2.4 About Third Party Access to Serialized Data

With certain serialization mechanisms, it is easy for third parties to access and modify the data stored in them. Examples of such mechanisms are relational databases and XML files. There are some potential problems due to this.

First, this clearly exposes the internal structure of the domain model to third parties, as the serialized form is typically quite direct a reflection of the class model. This can be problematic, if the authors of the program do not want to reveal implementation details due to, for instance, commercial reasons, or in order to avoid third party software to become dependent on the implementation details of the current program version.

Second, modifications to the serialized form may break the integrity constraints of the domain model. This will happen almost necessarily, since it can be too demanding to implement deserialization in a way which guarantees the same data integrity as the domain model updates themselves.

Third, domain model changes will cause evolution in the data model. Unless special care is taken, this will typically break third party applications which rely on a certain version of data model. This is quite bad for agile development, since the fear of breaking other applications may make refactoring less appealing.

When implementing such open serialization mechanisms, the developers should consider documenting the classes which are stable and allow external modifications. Those classes can be then used to implement import and export interfaces to external third party applications. From the point-of-view of developer effort to serialization, these interfaces will take additional time and also limit the possible refactorings and changes which can be applied to the domain model. Other domain classes are still free to change at will, since they are declared unstable.

2.5 Existing Serialization Implementations

Several serialization libraries exist for most available programming languages. This makes it possible to enjoy the benefits of the library approach without the effort of writing one from scratch. Next we study some of the C++ and Java libraries in the context of the requirements presented in this chapter.

Microsoft Foundation Classes [41] is a C++ framework which includes support for serializing objects. Serialization is achieved by deriving the domain model classes from the common base class of MFC, `CObject`, overriding a member function for reading and writing the class member variables and applying two macros for dynamic creation during deserialization-time. Schema evolution is supported so that each object can have an integer value for the object version. Various serialization mechanisms can be implemented by deriving mechanism classes from the `CArchive` base class. MFC includes support for a binary flat output out-of-the-box. The framework does not have direct support for C++ Standard Template Library (STL) types, such as containers, but instead it provides custom implementations for most usual containers.

Boost Serialization Library [47] is a modern peer-reviewed, open-source and highly flexible C++ library for serializing arbitrary data types. The library runs on several platforms and compilers. The serialization library is plugged into user defined types by implementing a member template function. Dynamic binding is implemented with a clever scheme using the C++ `typeid` facility, that is, the correct serialization functions can be called depending on the run-time type of the object. Object instantiation at deserialization-time is implemented using a generic factory, which by default constructs new objects using their default constructors. The classes are automatically registered to the factory once the deserializer encounters them. Built-in and most standard library types, such as containers, are directly supported. Boost handles versioning in the same way as MFC, by having a version number for each object. The library includes mechanisms for writing to XML, text and binary files, and the users are free to add new ones. Failures in the library are reported using exceptions.

The application developer has the ability to control the behavior of many of the library features, mostly due to extensive use of C++ templates. For example, the serialization function can also be a free-standing non-member function, if non-intrusive serialization is required. This requires that the separate function is able to access enough of the internals of the serialized object. Furthermore, the function can be split into separate loading and saving functions if necessary. The default `typeid` based dynamic binding can be replaced using a custom type information facility. Classes can be registered to the factory manually using macros. The instantiation function can be manually overloaded, for instance to use a non-default constructor. The user may prevent the library from tracking pointers to objects, which will result performance and memory boosts in the case where there are no shared objects or cyclic data structures.

The flexibility and the default assumptions made by the library are not purely beneficial. For instance, making the field identifiers optional causes serializers written just for the binary files be inadequate for the XML serializers. Registering classes to the factory automatically without forcing the user to do it may cause run-time failures in certain cases of serializing derived classes through base class pointers.

s11n Serialization Library [8] is a C++ library whose most distinctive feature is the built-in support for a wide variety of serialization mechanisms. It supports a binary mechanism, six text formats, three of which are XML, and a MySQL database mechanism. The current implementation is still somewhat immature. Most disturbingly, it does not properly support cyclic data structures, which makes the library of limited use.

Java serialization [57] is a built-in feature of the Java programming language. Java classes can be made serializable simply by implementing the `Serializable` interface. This is possible because Java provides run-time reflection which enables third party access to the internal attributes of objects during runtime and object instantiation by the class name, among other things. The serialization code can then automatically create, read and write objects. The included `ObjectOutputStream` serializes objects to a binary format. Schema evolution can be achieved by manually adding a version number to each class, although it requires certain additional trickery. Namely, the serialization code refuses to load changed classes, unless their `serialVersionUID` attributes are equal, so it must be ensured that the value does not change. Also changes in the inheritance relationship are not allowed.

Java classes can also be made serializable by implementing the `Externalizable` interface. In this case, the serialization and deserialization functions need to be implemented by the programmer, that is, they are not automatically generated. Swapping between the `Externalizable` and `Serializable` interfaces will break the schema evolution support, so once the decision has been made, it may be difficult to change it afterwards.

Java Serialization to XML (JSX) [31] and *XStream* [29] are libraries which provide XML serialization for the Java language. They use the Java reflection facilities to extract data from objects and to write it back to them, so the programmer does not need to provide the serialization and deserialization functions. They support schema versioning by various transformations of the XML data.

Hibernate [30] is an open-source object/relational persistence service for Java. It implements a object-relational mapping schemes so that Java objects can be persisted to relational databases. The Java classes do not have to be modified in order to make them persistent. The library makes the necessary changes to their bytecode at run-time. The mappings from the classes to the database tables are specified in an XML file per each class. Hibernate supports many of the advanced features mentioned in Section 2.3.9, such as lazy deserialization.

Sun Java Data Objects (Sun JDO) [58] is a specification for achieving persistence to (relational) databases in Java. Several independent implementations exist, so the application has several vendors to choose from. JDO enables Java objects to become persistent by creating a schema description XML file and implementing the `PersistentCapable` interface. Most implementations have a bytecode enhancer, which removes the need to manually implement the (empty) interface in the Java code. JDO is neutral of the persistence mechanism used, but implementations typically use relational or object databases. Changes in the Java classes will require the developer to write some explicit evolution support code, as JDO implementations tend to support automatic schema evolution, that is, they do convert the database to use the new schema, but then just fill in missing values in objects with default values, such as `0`, `false` or `null`.

2.6 Other Approaches

In addition to the serialization library design described in the previous sections and the ad-hoc approaches (brute-force and data access objects) presented in Section 2.1, there are other possibilities for achieving serialization or persistence.

Prevayler [33] implements a prevalence approach where the whole data model is always in memory, and persistence is achieved by logging the changes made to the domain model with the possibility to have full snapshots every now and then. Logging is implemented by making commands (see Section 3.2.1) persistent. The motivation of Prevayler for this is that main memories are growing to be big enough to hold *most* domain model instances completely in memory, and the Prevayler community argues that prevalence is more natural to object-oriented design than other approaches.

Texas [52] is an implementation of *pointer swizzling* at page fault time [70]. The basic idea is to access-protect pages which contain pointers to persistent objects, and load the objects from the persistent store to the memory when page faults occur. Persistence is achieved by physically copying the contents of the memory pages to the persistent store. This approach is very efficient and can be hidden from the application programmer quite well. As a drawback, versioning and multiple serialization mechanism support become quite difficult to implement.

gSOAP [59] is a C++ web services development toolkit. It employs a precompiler, which can be used to implement XML serialization for data structures. The precompiler does most of the implementation work, so the developer effort is minimal. However, as the toolkit is intended to be used with SOAP RPC, the implementation is not very flexible for other purposes, such as using other mechanisms or supporting versioning.

There are also other methods for dealing with persistence and serialization, but we omit their further investigation in this thesis. Examples of such approaches are object databases, which support the object-oriented programming model better than traditional relational databases, persistent containers, which require persistent data to be inserted into special containers, and aspect-based designs such as the one of Willink [67], which separate serialization code from other parts of the system using aspects.

Chapter 3

Undoing Actions in Interactive Applications

The possibility to undo and redo actions performed has become a prevalent feature in interactive graphical user interface applications [38]. In direct manipulation GUI applications a single mouse click is often enough to invoke complicated operations and the users do not always exactly know what they have done [11]. With undo the users are able to explore the functionality of the application without the fear of getting into trouble. This is an important factor in learning to use new software [21].

According to Abowd and Dix [2], the need for undo has been recognized at least since the early 1980's. Later studies have mostly focused on more advanced models of undo, such as selective undo [11, 73] and multi-user undo [15, 16, 56]. However, it has been observed that undo semantics, such as undo granularity, are not entirely clear to researchers, developers or users, even in the case of simple linear single-user undo [18, 38]. Due to these usability issues along with factors related to the cost and complexity of the more advanced approaches, the single-user linear undo model remains as the predominant recovery facility in contemporary interactive systems [60].

It should be noted that undo is a user intention, not a system function [2]. This means that undo should be viewed as the intentions of the users to recover from errors they have made, not as the undo button in the toolbar of the software. The users may accomplish the error recovery in a forward manner by manually applying such commands in the user interface that reverse the effects of their mistakes. The undo button should be seen as a backward error recovery facility, which is just one way of accomplishing the user intention. This thinking is particularly useful in the context of multi-user undo support.

Next we study how undo, the backward error recovery facility, can be accomplished. We do not study the details of the undo user interface, but instead the underlying programming techniques. We start by introducing different kinds of undo models. Then we discuss how the models, mostly single-user linear models, can be implemented. Finally, we briefly review some existing undo framework implementations.

3.1 Undo Models

Undo facilities vary greatly not only in their user interfaces, but particularly in their features. Several models describing which user actions can be undone and redone exist. Rough classification of the models can be presented according to two criteria:

- Chronological & linear versus selective & non-linear.
- Single-user versus multi-user.

In linear undo, the latest action in the history list must be undone before the ones preceeding it, and respectively the previously undone action must be redone before the succeeding ones. Selective undo allows choosing arbitrary actions in the history list and undo them without undoing the succeeding actions. Multi-user undo is needed when more than one user is allowed to modify the domain model concurrently.

Single-step undo, illustrated in Figure 3.1, is the simplest undo model. It allows undoing only the previous action, that is, the length of the history list is at most one [18, 38]. The model is called *flip undo* if there is a support for redo as well, and then the redo is interpreted as the inverse of undo so that undoing will toggle between the latest state and the state before it. This is shown in Figure 3.2.

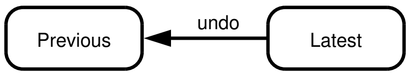

Figure 3.1: Single-Step Undo. The user can only reverse the effects of the latest action.

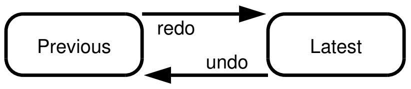

Figure 3.2: Flip Undo. The user can flip between the latest action and its predecessor.

Restricted linear undo allows undoing past operations in the reverse order of the user actions [38]. Usually redo is supported similarly, that is, undone operations can be redone in the order the user originally performed the actions. In Figure 3.3 this would mean that the application can be in any of the states, and the user can move between adjacent states using the undo and redo commands.

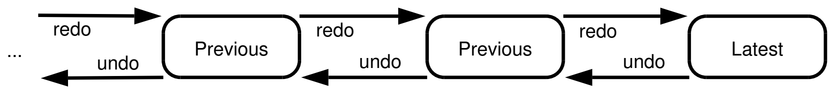

Figure 3.3: Linear Undo. The user can browse the history back and forth in a step-by-step manner.

When the user submits another command, there are two alternatives about what to do with the redo list [11]. The first approach is just to clear it. The second approach is to allow branches in the history, and give the users the possibility to choose the branch to take when they are redoing actions. The first approach is perhaps more common, since the latter approach, illustrated in Figure 3.4, may be more demanding for the users. It makes the user interface more complicated and also it may be more difficult to remember the contents of the states in the history tree compared to a history list.

It is so easy to submit commands in graphical user interfaces that it is likely that the user submits more commands before detecting an earlier mistake [11]. This is emphasized in multi-user appli-

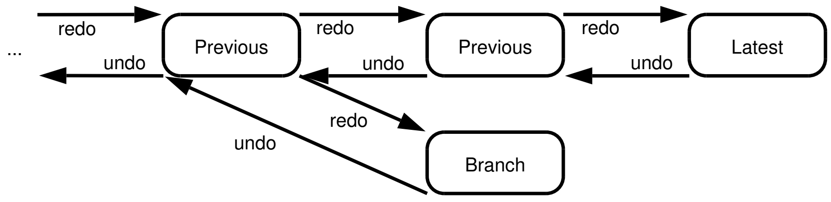

Figure 3.4: Linear Undo with a Branch. Submitting new commands when the redo-list was not empty has caused a branch to occur.

cations where other users may have submitted commands as well. It would be convenient if other commands than the previous one could be undone.

Non-linear undo enables the users to undo arbitrary commands in the history list, sometimes with validity constraints. There are several models for non-linear undo support. *Direct selective undo* [11] is based on the idea of equipping commands with functionality to directly selectively undo and redo arbitrary commands in a way which is only dependent on the current application state and the command[1]. Another non-linear undo, the Undo, Skip & Redo (US&R) model [61], is based on first undoing and then re-executing commands in the history list and skipping the ones that should be undone [66]. According to Zhou and Imamiya [73], other undo models include the script model [28], the Triadic model, the script-defined selective undo model [45] and the event-object recovery model [64].

A non-linear undo model is also applied almost always in collaborative environments where the domain model can be concurrently edited by many people. *Global multi-user undo* allows undoing the latest command submitted by any user [15, 16, 56, 65]. This does not necessarily require a non-linear undo, if user actions are synchronous. *Local multi-user undo* allows the users to undo the last action they have performed locally. This requires non-linear undo since other concurrent users may have submitted commands that should not be undone by the local undo. Abowd and Dix [2] give advice about selecting between global and local undo models in certain situations.

3.2 Undo Implementation Techniques

Like serialization, undo is also often implemented using a library or a framework. However, using an undo framework can still require a considerable amount of effort from the application developer. Also, undo often becomes an architecture fragment similarly to serialization. Washizaki and Fukazawa [66] state that the use of an undo framework costs a great deal in both the development and maintenance stages, and propose an approach where part of the undo code is generated automatically. Pugh and Leung [46] observe that *undo facilities are always appreciated by users but the same is not true of implementors*. Berlage also acknowledges the efforts required for undo, and states [11]:

> Like many details of user-friendly interfaces, implementing an undo facility places an additional burden on the application programmer. Generic features of the undo facil-

[1] See Section 3.2.6 for further details.

> ity that appear in every application should be available as reusable code. Application-specific coding must be kept to a minimum. What the programmer needs is an implementation framework for undo.

A problem with typical undo frameworks is that they mostly focus on the user interface and on providing the policy for determining when to call the undo mechanism and for which user actions it can be called, but leave the details of the undo mechanism implementation to the application developer. Indeed, the mechanism can be quite painful to implement as we demonstrate in the coming sections. However, we will first describe a classification of common inversion mechanisms [11]:

Full checkpoint The full state of the domain model is saved between user actions. This is easy to implement, but clearly quite inefficient. Full checkpoint is rarely used in practice, but for implementations see [42, 50].

Complete rerun The initial state of the domain model is saved, and any state in history can be reached by redoing the commands in the history list. Like full check point, this is also rather inefficient.

Partial checkpoint The most often used strategy in which just the part of the domain model state modified by the user action is stored. Undo is achieved by restoring the state. The programmer must make sure that all relevant modifications to the domain model are captured by the checkpoint, or the undo will not work correctly. This may be somewhat difficult, as we explain later in this chapter.

Inverse function Sometimes commands have inverse functions which do not require explicit state information. For instance, moving an object in a drawing program can sometimes be undone by moving the object back by the same distance. This is not always possible even in this simple example, see for instance [24, p. 283]. Inverse function cannot be used in non-linear undo since the effects of such functions depend on the execution context. Another problem particularly emphasized with inverse functions is error accumulation, known as hysteresis. Small errors in the inversion can accumulate when repeatedly undoing and redoing actions.

Using any approach, it is natural that storing the information needed for undo consumes some memory. Therefore a strategy for pruning undo information is also needed. The most straight-forward choice is to remove very old undo information completely from the memory. We discuss another possibility in the context of our undo solution later in Section 6.2.5.

A patented approach for efficiently achieving the local checkpoint mechanism was introduced by Zundel *et al* [74, 75], and is based on memory protection. Memory pages are initially marked read-only, and a page fault handler is installed. Trying to write to the memory will then cause page faults and it becomes possible for the handler to save the memory state before the client application modifies it. This is somewhat analogous to the pointer swizzling method discussed in Section 2.6.

3.2.1 Command Pattern

Virtually all undo literature uses the *command design pattern* to implement undo. The pattern facilitates the complete rerun, partial checkpoint and inverse function approaches. The idea of the command pattern is to encapsulate the user change request to a command class which knows how to achieve the change in the domain model, and which can be invoked by a third party. This way the third party invoker is fully decoupled from the details of the change, but it can still schedule the execution of the action.

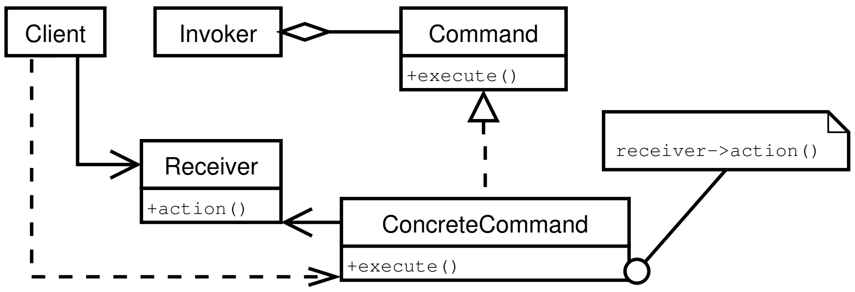

Figure 3.5: Command Structure. The `action()` of `Receiver` is decoupled from the `Invoker` by `Command`.

Figure 3.5 depicts the structure of the command design pattern according to Gamma *et al* [24]. Command can be used for supporting undo and redo by adding two more member functions to the interface, namely `undo()` and `redo()`. They are sometimes also referred to as `unexecute()` and `reexecute()`, respectively. The `redo()` function will usually be the same, or at least very similar to `execute()`, but in certain special cases there may be differences. Some implementations also add `canUndo()` to the interface for supporting non-undoable actions. The use of a command object in the context of a layered GUI application would then be as follows:

1. The user interface layer creates a new `ConcreteCommand` for modifying a `Receiver`. The `Receiver` is an object in the domain model.
2. The UI passes the `ConcreteCommand` object to the `Invoker`[2], which executes the command. The `Invoker` is unaware of the fact that the command is actually a `ConcreteCommand`, that is, it uses just the `Command` interface.
3. Later, when the user wishes to undo the command, `Invoker` will call the `undo()` member function of the latest command object, which in turn will reverse the effects it has had on the domain model in the `execute()` function.

This is convenient, as most of this code can be packaged in a reusable library. What remains to be implemented by the application developer is the undo mechanism, that is, the algorithms which perform the forward and backward operations. These are to be implemented as the `execute()`, `undo()` and `redo()` functions. Implementing `execute()` should be trivial, the function usually just delegates to the domain model. Additionally, the `execute()` function can have further responsibilities,

[2]The `Invoker` will typically be a `CommandProcessor`, see Section 3.2.5.

such as logging. `redo()` should be similarly easy to implement, because it is identical to `execute()` in most cases.

```
class PaintCarCommand : public Command {
    Car* car;
    int newColor;
    int oldColor;
    ...
    virtual void execute() {
        oldColor = car->getColor();
        car->setColor(newColor);
    }

    virtual void undo() {
        car->setColor(oldColor);
    }
};
```

Listing 3.1: The `PaintCarCommand` class. The command will paint a car with a new color in the `execute()` function and undo the painting in the `undo()` function.

Implementing the `undo()` function is sometimes quite straight-forward. For instance, a command for painting a `Car` object can undo the effects it has by repainting the car with the old color. This is shown in Listing 3.1. Details have been omitted for the sake of clarity. The `execute()` function just stores the old color of the `Car` object prior to modifying it. The `undo()` function then just reverts to the old color. However, implementing `undo()` may be trickier, too. The programmer must find out what parts of the domain model the `execute()` function modifies, how the modified state can be captured and how the modifications can be reverted. This may be non-trivial. For instance, the implementation of a `RemoveCarCommand` class would be somewhat more challenging, given that cars can be in garages and have wheels. Also, if the car keeps track of the layers of paint it has or is later changed to do so, the command given in the example listing will not suffice.

3.2.2 Memento Pattern

Implementing the `undo()` function in the command pattern is tedious at best, and in some cases quite difficult as the effects of the `execute()` function are hidden in the domain model. The *memento design pattern* is helpful for reducing these difficulties. The idea is illustrated in Figure 3.6, which describes the memento pattern according to Gamma *et al* [24]. The `Caretaker`, which would be the command object, asks the domain object it is about to modify (`Originator`) to create a snapshot of its internal state prior to the mutating function call. It then stores the snapshot, `Memento`, for later use in `undo()`. When the `undo()` function is called, the command object can just send the memento back to the `Originator`, effectively restoring the state. The `Caretaker` never needs to, and is never supposed to do anything with the contents of the `Memento` object. It just gets the object from the `Originator` and gives it back later.

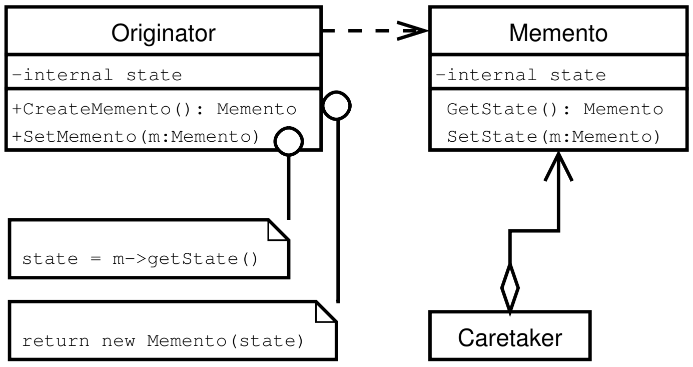

Figure 3.6: Memento Pattern. `Memento` externalizes the internal state of `Originator` without violating encapsulation.

For example, suppose that there is a `Car` class in the domain model, and the user has selected to paint a car using red color. Figure 3.7 shows the calls made by the related parties. The `Invoker` tells the `PaintCarCommand` to execute. The command starts by saving a `CarMemento` of the `Car` object whose color is about to change. Then it paints the car red by calling the member function `setColor(...)` of the `Car` class. Later, when the user wants to undo the painting, the `Invoker` calls the `undo()` function of the `PaintCarCommand`, which in turn sets the stored memento back to the `Car` object, thus reverting it to the old state.

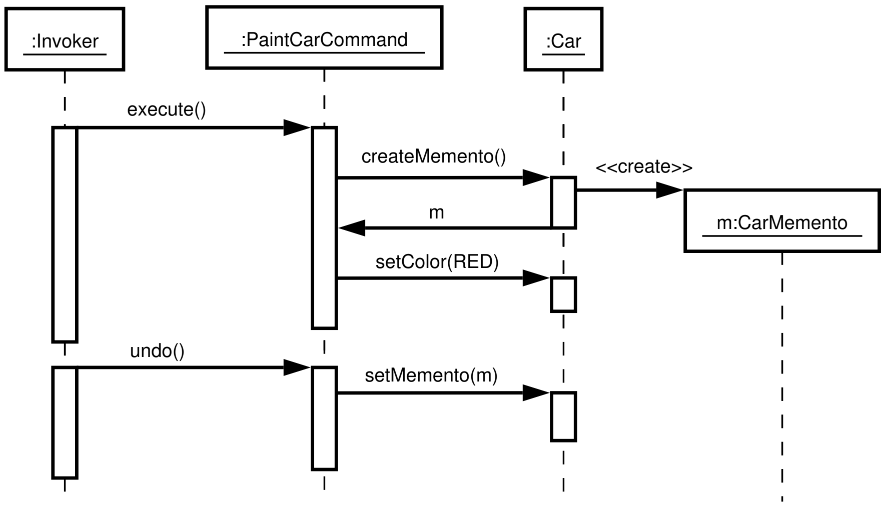

Figure 3.7: Painting a `Car`. The `Car` is first painted red, then the painting is undone.

A requirement with this approach is that the domain model classes must provide the functionality for creating and setting mementos. Implementing these can again be somewhat burdening, as the traditional approach is to have one memento class per each domain class. The implementations of such mementos can also be tedious, as they need to be able to contain the local states of the domain objects.

Völter [62] proposes using Java serialization to ease implementing mementos. He suggests that the internal state of the domain object be stored in an inner class of the domain class. The contents of the inner class can then be conveniently stored and restored using the Java built-in serialization support.

This leaves the programmer with the effort of implementing the memento setter and getter functions, but eliminates the need for manually implementing a new memento class for each domain class.

3.2.3 Undo and Domain Model Encapsulation

Especially without using mementos, there is a risk that using the command design pattern the command classes will end up violating the encapsulation of the domain model classes. Unless the domain model explicitly provides reverse operations for all functionality it exposes, the responsibility of *knowing* the details of how to achieve the reversing effect is moved from the domain classes to the command classes. If the command classes are not part of the implementation of the domain classes, the command objects become dependent on the *internal behavior* of the domain objects, that is, violate encapsulation. Encapsulation violations should usually be avoided.

Edwards *et al* [19] observe this and point out that knowing the side-effects at the time when the set of command classes is written can be difficult. Meshorer [40] recommends making commands inner classes of the domain classes. Berlage [11] states that commands may violate the principle of encapsulation because target objects may have internal state not accessible to the commands. However, he considers the problem to be theoretical on the basis that a good program design should provide means to reverse the effects of most actions from the outside, and for the rest a memento-style approach can be used. Wang and Green [64] assert that the construction of a history list that contains object state at the system level is not possible in an object-oriented system, and suggest that the history information would be stored within the domain objects themselves.

A virtue of the memento pattern is that it allows the originator object to provide a convenient means for implementing the undo mechanism *without violating encapsulation*. This means that the caretaker of the memento never operates on the contents of the object, which leaves the originator fully in control of its internal state. It is usually possible for the originator class to completely prevent third party access to the member functions of the memento by using the programming language facilities. For instance, in C++ this can be achieved by using the `friend` keyword. In Java private inner classes or package private visibilities can be employed.

There is still an issue left with using mementos in the described way. The command needs to know which objects it will be modifying by the domain model function call it intends to make, and store the states of those objects by using mementos or other means. It would be convenient if storing a memento just for the domain object being called would suffice. This, however, is not realistic, since it would require the domain object to store its transitive state instead of the *local state* [53], which would make the mementos unnecessarily large. This would blur the distinction between partial and full checkpoints and make undoable operations unnecessarily inefficient.

This leaves us in a situation where we will be somewhat dependent on the details of the domain model implementation, that is, violating its encapsulation. The basic trade-off using the techniques described in this chapter is between encapsulation and efficiency. In practice most do not see the encapsulation very problematic and go for the better performance.

3.2.4 Composite Command

Sometimes multiple commands need to be executed as one grouped logical command. For instance, if the user has selected multiple `Car` objects which need to be painted, undoing the action should undo painting all of the cars, not just the last one. In other words, there should be some mechanism for controlling the *undo granularity*. The `PaintCarCommand` class can of course be extended to support painting multiple cars at the same time, but just grouping several `PaintCarCommand` objects as a single transaction is a better approach, as it cleanly generalizes to different command types. A method for implementing the feature is to apply the *composite design pattern* [24]. It is illustrated in the context of the command pattern in Figure 3.8.

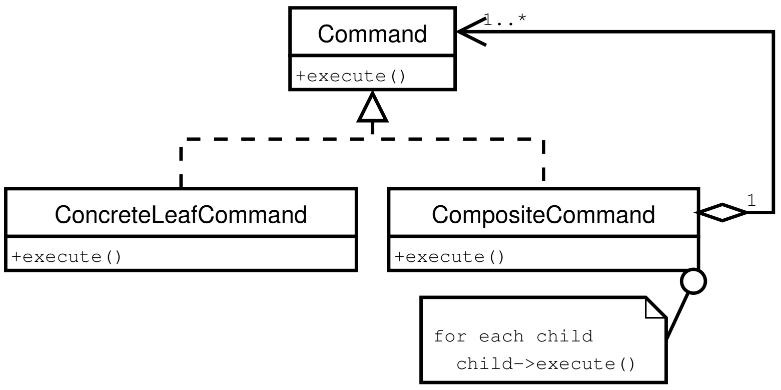

Figure 3.8: Composite Command. The composite pattern allows creating hierarchies of commands which behave as a single command from the point-of-view of the invoker.

The `CompositeCommand` class implements the `Command` interface by calling the `execute()` operation for all of its child commands, and similarly for the `undo()`, `redo()` and other functions in the chosen `Command` interface. When a `CompositeCommand` is executing, undoing or redoing its child commands, it should only succeed if all of its child commands were successful. The composite treats its children through the `Command` interface, so composites may also be children of other composites. This allows a hierarchical structure to be built.

Some implementations place the functions for accessing the children of the composite directly to the `Command` interface. In a strict object-oriented sense this means that all commands need to be able to act as composites. Whether it is justified to add such responsibilities to all derived classes is questionable, but can be acceptable for example in the case of a *merging* scheme. For instance, if two commands representing typing text in a text editor are submitted consequently during a couple of seconds, merging them to just one command could make sense.

3.2.5 Command Processor Pattern

In order to keep track of the commands to be undone or redone next, some scaffolding code is required. The command processor design pattern [14] describes how that code can be organized. Historically, the command processor pattern is the basis of how commands are used in most restricted linear undo facilities [66].

The most important responsibilities of the command processor are the execution of the commands and storing the command objects in undo and redo lists. The former means that the command processor acts as the invoker of the command pattern. The latter is depicted in Figure 3.9. As new commands are submitted to the command processor, it executes them and appends them to the undo list. As commands are undone, it removes them from the undo list and prepends them to the redo list.

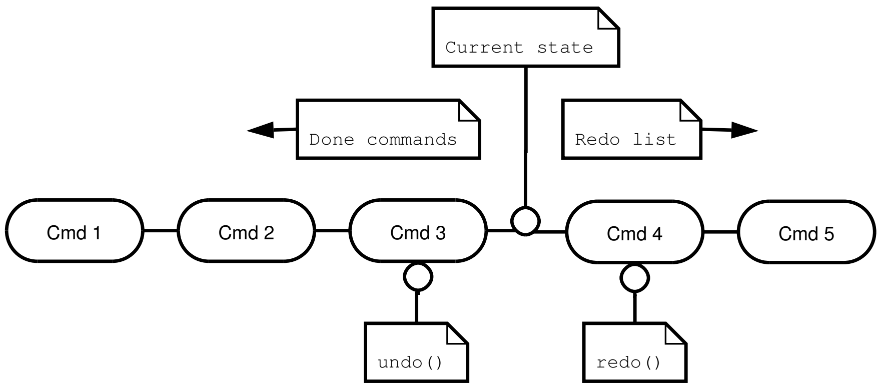

Figure 3.9: Linear List of Commands. Commands on the left side of the current state are historical commands performed by the user. They can be undone. Commands on the right side have been undone, and can be redone.

Additional responsibilities can be added to the command processor class. For example, it can provide logging facilities, serve the names of the next commands to be undone and redone to the user interface and keep track of the *changed since the last save* flag.

3.2.6 Direct Selective Undo

A factor which makes implementing a restricted linear undo facility easier is that commands are always undone in the resulting state of the original execute operation. This is called *stable execution property* [11]. In a non-linear undo, the programmer needs to pay additional attention on checking whether it makes sense to perform undo in the current state. It may also be necessary to adjust the undo command to fit the state, if it has been modified in a way which makes the original undo command faulty. For instance, it makes no sense to undo painting a `Car` object, if it has already been deleted. Likewise, inserting a new `Car` to position three in a row of cars in a `Garage` may require adjusting the insertion position, if some of the cars have been removed from the garage earlier.

Berlage [11] proposes a convenient approach for implementing a non-linear undo. The fundamental idea of the *direct selective undo* method is to equip the command classes with four additional member functions: `isSelectiveUndoPossible()`, `isSelectiveRedoPossible()`, `selectiveUndo()` and `selectiveRedo()`. The `isSelectiveUndoPossible()` implementation needs to check if it makes sense to undo in the current program state. If the function returns true, then the `selective-Undo()` function must return a new command object, which reverses the effects of the original command in the current program state. The new reverse command is then executed like any normal command and appended to the undo list. The new functions for redoing work similarly to the undo functions.

The direct selective undo approach has several advantages compared to traditional non-linear undo facilities. Because the undo is performed directly in the current program state, without dependencies to other commands, it becomes possible to only implement the selective undo functions to selected commands. That is, the developers may choose the most important commands and implement selective undo just for them. It also becomes easier to add support for selective undo later in the development process, and the implementation can then be incremental.

Another benefit with direct selective undo is the direct interpretation of the reverse effects. Traditional script-based non-linear undo models, which perform undo using the semantics of skipping commands in a script may sometimes yield unintended results for the user. For instance, suppose that the user has first painted a blue `Car` red and then repainted the red car with yellow. If the color of the car should be restored to blue, the natural action using backward error recovery would typically be to undo the command which painted the blue car with the red color. However, using script-semantics, the car would remain yellow, as the yellow paint had been applied later in the script. On the other hand, the direct interpretation would have the desired results.

It is evident that implementing the additional functions required for selective undo is not always trivial. In the best case the effort approaches that of implementing the normal `undo()` function in the command class without using mementos. Due to the non-linear nature of the selective undo, it appears that there is no obvious way of coming up with a memento-like implementation simplification.

3.3 Existing Undo Implementations

Dia [1] is a diagram creation program written in the C programming language. It implements undo by having apply and revert functions for all changes. Objects, hand-crafted using C, have an interface for getting and setting their internal states. Each class in the domain model has a sibling class for storing the internal state. Change objects are stored in an undo stack. The undo stack can be navigated using a linked list built into the change class. Changes can be grouped using transaction points. In effect, the implementation uses the standard techniques described in this chapter with different names. Table 3.1 summarizes the terminology differences.

Term Used in This Thesis	*Dia Term*
Command	Change
Do&Redo	Apply
Undo	Revert
Memento	Object State
Command Processor	Undo Stack & Change

Table 3.1: Dia Terminology. Dia uses common techniques for undo, but the terminology differs from ours.

Java Swing [40] provides a framework for implementing undo. It is based on command objects, although the terminology is again different. In Swing, `Command` is called `UndoableEdit` and `Command-Processor` is called `UndoManager`. The `UndoableEdit` interface defines member functions for undoing, redoing and getting names for both operations. It also provides `canUndo()` and `canRedo()`

functions for supporting commands which are not undoable or redoable. Additionally, the interface allows commands to be grouped in a hierarchical fashion by using the `addEdit(...)` function. The framework provides an implementation called `CompoundEdit`, which supports containing an arbitrary number of `UndoableEdits`, all of which will be undone and redone at the time the respective member function of the `CompoundEdit` object is called.

PDF Studio [72] is a pure Java program for creating Portable Document Format (PDF) files. It contains a straight-forward implementation of the command design pattern, with the `canUndo()` function included. The `CommandProcessor` implementation is called `CommandManager`. Commands can be grouped into one similarly with the Java Swing approach, using a composite command named `MacroCommand`.

Völter [62] uses Java serialization to implement mementos as described in Section 3.2.2. Then he uses the mementos to implement rolling back in case a transaction to the domain model fails [63]. The contents of the transaction are specified by adding commands to a transaction object.

SQLite [26] is a light-weight relational database implementation, which can be used for implementing undo. If the application directly passes changes made in the user interface to the database, SQLite can be configured to record all changes into a temporary undo and redo log table by using triggers. This is quite different than the command approach, and according to the author, surprisingly little code is required for supporting undo and redo using SQLite.

Kaava [44] is a prototype for a user interface to computer algebra systems. The domain model of the application is implemented as a tree. Linear undo is implemented by creating a full checkpoint of the tree. However, a partial checkpoint can also be created by sharing unmodified subtrees. Additionally, Kaava implements a non-linear undo which copies histories of subtrees to the current tree. The linear undo sees the non-linear undo as a separate editing action, so actions undone using the non-linear undo can be redone by using the linear undo.

Chapter 4

A Serialization Library

In this chapter we present the design of an object serialization library. The library is implemented using the C++ programming language, which supports the object-oriented programming paradigm among others. It does not include reflection and memory is usually manually managed. The language has a powerful template system, which can be used for various additional features. However, the core design we present does not depend on templates, and can be implemented in other object-oriented programming languages as well.

We start the discussion with an overview of how the library is connected to client applications. Then we move on to describing the details of the implementation within the context of the requirements specified in Chapter 2. Finally, we give example program code for the serialization function the client application developer needs to write. The reasoning behind the design decisions is somewhat explained in this chapter, but further analysis of the consequences and implications of the design are given in the coming chapters.

4.1 Application Interface

The use of our library from a client application is illustrated in Figure 4.1. Interacting with the library consists of two major parts. First, the domain model of the application needs to be connected to the library by deriving the domain classes from the serializable base class `StateObject`. The domain model also needs to implement a factory for creating instances of all domain classes. The factory needs to derive from the `AbstractStateFactory` class.

Second, the user interface layer of the application software needs to be able to call the saving and loading functions of the `StateManager` class in the serialization library. For the saving function to be called, the application needs to provide a set of root objects, whose *transitive* states are then serialized by the library. This is usually quite painless, as typical applications have a single root class, often called `Document` or `Project`, whose transitive closure includes all objects in the domain model.

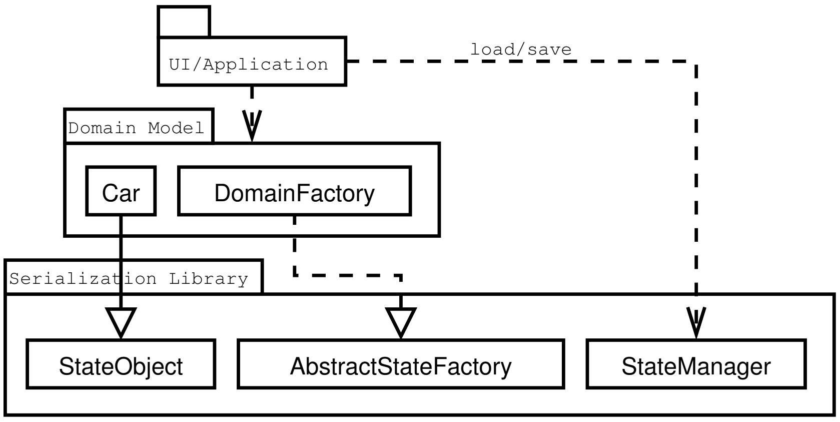

Figure 4.1: Connecting the Library to Client Applications. The application needs to implement certain interfaces provided by the library in order to enable serialization and deserialization.

4.2 StateObject Interface

Serializable domain classes that completely integrate to the library need to derive from the `StateObject` interface. For non-intrusive integration, an alternative option is also provided by means of *external* objects. The application developer can equip any data type with two free-standing template functions, `ExToBin` and `BinToEx`, which serialize and deserialize the data type to binary format. Then the details of the serialization are the sole responsibility of the application developer, that is, the library provides no assistance. However, the library can use the two functions for serializing external objects non-intrusively.

The `StateObject` class contains two member variables: the object identifier and the `StateManager` which exists to manage the object. The `StateManager` instance needs to be provided in the constructor of the `StateObject` class. The `StateManager` will then assign an object identifier to the newly created object. The identifier is implemented as an increasing 32-bit integer number.

For the purpose of serialization, the `StateObject` interface has three member functions: one for getting the object identifier, another for getting the class identifier and yet another for managing the state of the object. The class identifier is further explained in Section 4.7; the next section discusses state management.

4.3 Managing Object States Using Mementos

In order to serialize and deserialize the state of an object the library needs to be able to read and write the state. In our library, this is achieved using the `manageState` member function in the `StateObject` interface. Derived classes need to override the function and provide an implementation that calls the base class implementation and announces all new member variables that need to be serialized or deserialized.

The `manageState` function takes one argument, a reference to an instance of the `Memento` class. The memento interface has functions which serialize or deserialize the arguments given to them, depending on which activity the library is currently performing on the objects. The object can also ask the memento whether it is being serialized or deserialized, in case operation-specific actions are needed. Overloaded (de)serialization functions exist for the most common types the application developer would want to use in the domain model. This includes the built-in types of the language, strings and standard library containers. For schema evolution support, which is further described in Section 4.8, the `Memento` provides a function for getting the current domain model version number.

We present example code which also clarifies the memento interface in Section 4.9.

4.4 Managing Mementos

It would be convenient if the `StateManager` could deal with multiple different states of the domain model. In our library, this is achieved by managing the `Memento` instances by a separate class, the `MementoManager`. This is also good from the view-point of responsibility assignment, as it reduces the number of responsibilities of the `StateManager` class.

When the `StateManager` should serialize domain objects it manages, it creates a new `Memento-Manager`, and ask it to create the mementos for the root objects the client application wishes to serialize. The `MementoManager` will then serialize the objects to internal data structures of `Memento` objects. Two reachability modes are supported. First, the full transitive closure of the root objects is serialized. Second, only the given objects, that is, their local state is serialized. As a result of the serialization process, an instance of the `MementoManager` class is returned to the client application. The client application can then use a serialization mechanism to turn the contents of the `Memento-Manager` into binary flat files, for instance.

Deserialization works similarly to the serialization process. The client application provides an instance of the `MementoManager` class, which it has either obtained from a serialization mechanism, or from an earlier call to the serialization functionality of the `StateManager`. When the `State-Manager` receives a `MementoManager`, it will ask it to set all the mementos, which will cause the respective objects to be instantiated if necessary, and their states restored.

4.5 Serialization Mechanism

The `MementoManager` serializes the objects into a generic format. The main data structure of the format is an object, which contains an object identifier, a class identifier and a varying number of fields. Fields are used for storing member variables of the domain classes. Each field contains the field identifier, the field type and the value of field. All identifiers are stored as 32-bit unsigned little-endian integers. The encoding and the length of the value are dependent on the field type. As mentioned in Section 4.3, the built-in types of the language, strings and the standard library containers are supported. The encodings are implemented straight-forwardly, for instance a string is encoded by storing the length of the string as an integer followed by the 8-bit characters in the string.

The serialization mechanism has access to the extracted object data structures of the `Memento-Manager`. This allows arbitrary serialization mechanisms to be implemented and satisfies the mechanism decoupling condition illustrated earlier in Figure 2.5. A mechanism for persisting the data to a zlib [37] compressed binary flat file is implemented.

4.6 Object Tracking

When a new `StateObject` is created, the `StateManager` assigns an object identifier to it. The identifier is unique within a `StateManager`, but will collide with other instances of the class. The `StateManager` maintains a mapping from the object identifier to the `StateObject` instance. Once a `StateObject` is destructed, it is removed from the mapping. This allows looking up alive objects based on their identifiers. Deriving domain model objects from `StateObject` enables the mapping to be implemented conveniently since the library can intercept the creation and destruction of domain objects using constructors and the destructor.

The mapping of the objects is not emptied when there are no serialization or deserialization activities taking place. This may appear to be somewhat surprising as the mapping naturally consumes memory. The mapping will however prove to be of use when implementing undo as can be seen in Chapter 5, and can be useful for detecting memory leaks as explained in Chapter 7.

Looking up objects is required during the deserialization process. When a class has a member variable pointer to another object, the referenced object needs to be instantiated and assigned to the pointer. However, if the object has already been instantiated, a second instantiation would be erroneous. Thus, when a pointer is encountered, the deserializer first tries to get the referenced object from the `StateManager` by the object identifier. Only when the lookup fails will a new instance be created.

Creating the new instance is handled using the `AbstractStateFactory` which is further described in Section 4.7. The factory requires that the caller knows the class identifier of the object to be created. This is not a problem since the `MementoManager` is already completely in memory, and the class identifier of any object which has a `Memento` in the manager can be retrieved by the object identifier.

4.7 Object Instantiation

The `StateObject` interface has a virtual member function for getting the class identifier for an object. Each class in the domain model of the client application must return a different identifier. The domain model has to implement an instantiation factory, which derives from the `Abstract-StateFactory` interface. This is known as the *abstract factory* design pattern [24]. The interface has one member function, `createInstance(...)`, which takes a class identifier as an argument. The implementation of the factory must then provide a new blank object, whose class is determined by the argument of the function call. This allows the serialization library to create new instances of the client application domain model classes without coupling the library to the client application.

The `StateManager` takes an `AbstractStateFactory` argument in its constructor. The `State-Manager` will use the same factory throughout its existence for creating new instances during deserialization, and as we will see in the next chapter, during undo.

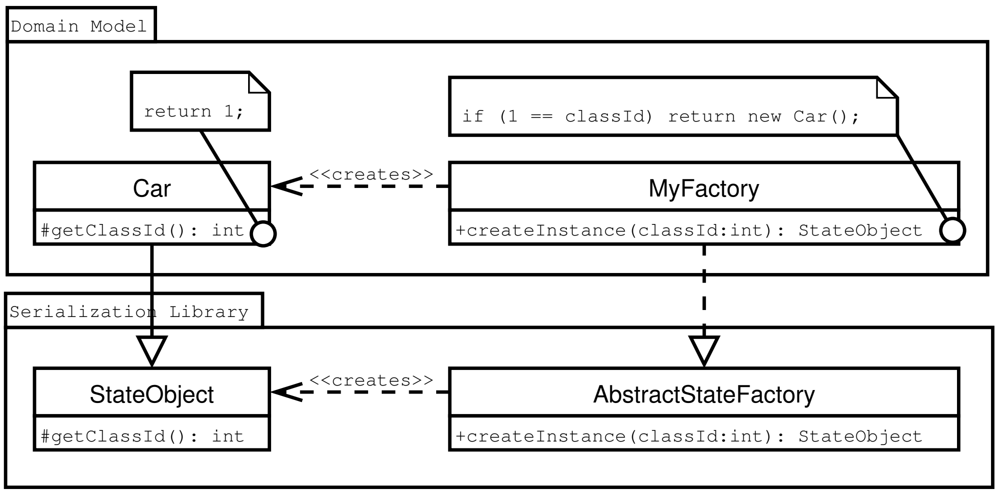

Figure 4.2: Factory Example. The serialization library can create new instances of client application classes by using the `AbstractStateFactory` interface.

For example, suppose that the domain model consists of only one domain class, `Car`. The client application will then derive the `Car` class from `StateObject` and create a new factory which is derived from `AbstractStateFactory`, as depicted in Figure 4.2. The `getClassId()` member function of the `Car` class returns one, so the factory implementation will return a new `Car` instance when the argument to the `createInstance(...)` call is one. When the library encounters the class identifier one in an object to be deserialized, it will use the `AbstractStateFactory` interface to get a new instance of the corresponding class.

4.8 Schema Evolution

Schema evolution is implemented using a global domain model version number. The version number can be queried through `Memento` during deserialization so that appropriate adjustments can be made to the member variables of the class.

Because some adjustments require that other objects have been deserialized already, the library also provides a validation round. The validation round is executed for all domain model objects after everything has been deserialized. This allows further interactions across objects, since their states are more valid than in the middle of deserialization. However, when validating an object, there is no guarantee that the objects it requires for validation have already been validated.

4.9 Serialization Function Example

The case we are using for this serialization example is familiar from the previous parts of this thesis. Let the domain model have two classes, `Car` and `Wheel`. The class definition of the former is outlined in Listing 4.1. Details have been omitted for the sake of clarity. The `Car` class derives from the

StateObject class, and overrides member functions for getting the class identifier and for reading and writing the object state.

```
class Car : public StateObject {
    int color;
    int numberOfWheels;
    std::list<Wheel*> wheels;
    ...
    virtual void manageState(Memento& m);
    virtual void getClassId() { return 1; }
};
```

Listing 4.1: Outline of the Car Class. The manageState function will pass class member variables to the Memento argument.

Just calling a function of the Memento class for each member variable of the class would suffice for the implementation of manageState. However, in Section 2.3.1 we speculated that saving the numberOfWheels might not be desired, so that scenario is illustrated in Listing 4.2.

```
void Car::manageState(Memento& m) {
    StateObject::manageState(m);
    m.field(100, color);
    m.field(101, wheels);
    if (m.isDeserializing())
        numberOfWheels = wheels.size();
}
```

Listing 4.2: (De)Serialization Function of Car. The numberOfWheels derived variable is not serialized so it must be updated manually during deserialization.

The manageState implementation first calls the base class implementation in case it needs to serialize or deserialize something. The code continues by passing the serialized member variables to the Memento, which will either read or write them depending on whether the object is being serialized or deserialized. The field function is overloaded, so the Memento is able to handle various types using code written specifically for them. The numberOfWheels variable is never passed to the Memento, but instead it is manually updated at deserialization-time. The field identifier constants 100 and 101 are arbitrary, but the domain model developers need to ensure that they do not collide with base or derived classes.

Chapter 5

Extensions for Undo Support

The serialization library presented in the previous chapter allows, among other things, external code to access the internal state of domain model objects much like reflection facilities in some programming languages. Furthermore, it already provides an implementation for creating and setting mementos for arbitrary classes. The contents of a memento manager can be deserialized into a domain model already in memory so that existing objects are updated. This can be implemented conveniently because the object tracking map is continuously maintained.

In this chapter we will demonstrate how the library can be used and extended to support undo, and what additional requirements it will place on the client application. The undo model we implement is the restricted linear single-user undo. As explained in Chapter 3, it is difficult to implement more complicated undo models just by using mementos. However, the design we provide does not prevent implementing selective undo, which is shown in the next chapter.

We will start this chapter by showing how the user interface layer is connected to the undo support of the library. Then we present ways of handling the user interface responsibilities imposed by the library. Next we discuss how the domain model classes need to be modified in order for the undo mechanism to work properly. Finally, we present the details of how the state inversion works.

5.1 Application Interface

The client application controls the behavior of the undo system from its user interface layer. When the user submits a new command to be executed, a new transaction needs to be started in the serialization library, so that the set of modifications done by the command can be identified and recorded. Contents of the transactions are recorded by the `UndoLevel` class of the library. Unlike transactions in databases, committing is not needed because the latest `UndoLevel` object will always capture the changes made to the domain model.

The changes made by the user action need to have a name which can be displayed in the undo menu, so it is possible to assign names to undo levels. The name can be freely formatted by the user interface. For instance, an undo level in which a color of a car is changed could be named "Color change of XYZ-123 from red to blue”, where “Color change” would be the type of the action, “XYZ-123” would meaningfully identify the target car to the user and “from red to blue” would describe the content of the change.

Finally, the most obvious function provided by the library is the ability to undo previous actions. Undoing once will revert the changes made within the most recent undo level and remove the undo level from the history list. Redoing would work similarly, but in the opposite way, if it were implemented.

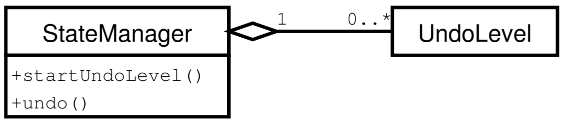

Figure 5.1: Undo Levels. `StateManager` can have an arbitrary number of undo levels. The client application can create new undo levels by calling the `startUndoLevel()` function. Actions can be undone by calling the `undo()` function, which also removes the latest undo level.

As shown in Figure 5.1, the `StateManager` class has an interface for starting and undoing undo levels. Additionally it supports naming and renaming the undo levels. The number of stored undo levels can be limited in order to limit memory use, and all of the undo levels in memory can be cleared at will.

To summarize, adding undo support to the user interface layer requires two main steps. First, before a logical set of changes to the domain model, a new undo level needs to be started and named. The name can be changed later, if necessary. Then the new undo level starts recording the changes made, and the previous level becomes passive. Second, an undo button needs to be implemented so that clicking it will call the undo function of the `StateManager`. It is also a good idea to display the name of the undo level in, for example, the tooltip of the undo button.

5.2 Managing Undo Levels

The question when to start a new undo level in the user interface is conceptually simple, but the implementation requires further attention. The simplest way to handle the requirement is just to locate all places in the UI which modify the domain model, and prepend the code block with a `startUndoLevel()` call.

The disadvantages of this brute-force approach are that it may be difficult to remember to add the function call at all modification sites and that modifying the undo level starting preamble becomes difficult. An example of the latter case is displaying the name of the undo level in the status bar of the program.

A more disciplined way of implementing the undo level starting is by using commands and the command processor pattern. As the command processor invokes a command, it can start a new undo level. This approach factors the common operations related to making a change to a centralized place in the command processor. It also factors the code related to a user interface action to one class.

Additional functionality for combining undo levels may be required in cases where composite command is not enough for grouping multiple commands. An example of such a case is a dialog which makes changes to the domain model not only when the OK button is clicked but also at other times, and the dialog invocation should produce only one undo level.

5.3 Storing Object States for Undo

In addition to changing the user interface layer, the domain model needs to be slightly modified as well in order to get working undo support. The library needs to be able to track changes made to the domain model during an undo level, so that the changes can be reversed when the undo level is reverted. Possible operations on domain model objects can be categorized to *create*, *read*, *update* and *delete* (CRUD [4]). The reverse operation for creation is deletion and vice versa. Recreating a deleted object needs to put the object to the state in which it was before being deleted. Reading does not change its target, so no reversal is necessary. Updating is reversed by restoring the object to the state before the update.

Our library keeps track of the domain model changes by three notification member functions in the `StateObject` class: `notifyCreated()`, `notifyChange()` and `notifyDestruction()`. The domain model classes of the client application need to call the `notifyChange()` function *before* making any changes to their member variables. In the beginning of the destructors, `notifyDestruction()` needs to be called. The `notifyCreated()` function is automatically called by the constructors of the `StateObject` class.

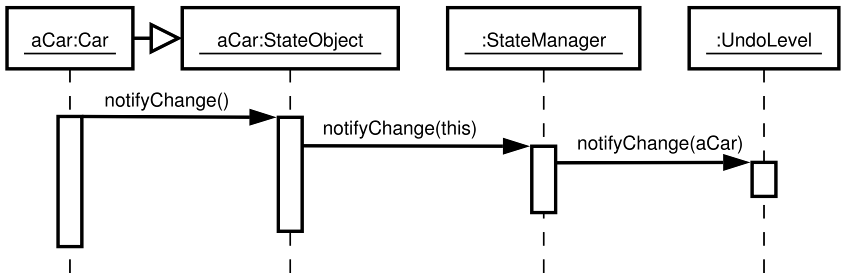

Figure 5.2: Notification Delegation. When a `Car` object is about to be modified, it calls the `notifyChange()` member function of its base class `StateObject`. The `StateObject` function delegates the notification to `StateManager`, which in turn delegates the call further to the currently active `UndoLevel`.

The notify functions mark the object identifiers and the types of the notifications made. The notify functions in the `StateObject` class delegate to the `StateManager`, which delegates to the currently active `UndoLevel`. This is illustrated in Figure 5.2 in the case of a `notifyChange()` call. The chain of delegations is similar in the case of the other notifications. The currently active `UndoLevel` object is responsible for storing the markings. The `notifyChange()` and `notifyDestruction()` notifications additionally store the states of the objects prior to their modifications.

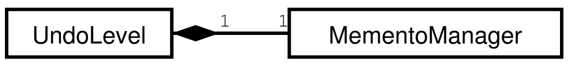

Figure 5.3: `UndoLevel` Uses `MementoManager`. The `UndoLevel` class implements state storing and restoring using the `MementoManager` class of the serialization library.

The states are stored using the serialization facilities of the library. Each undo level has an instance of `MementoManager`, which supports creating and setting mementos. This is depicted in Figure 5.3. For the purposes of undo, `MementoManager` does not create mementos by reachability, that is, it only stores the local states of the modified objects. This is enough since if other objects in the transitive state of the notified object are changed, they will call the notification functions for themselves.

In other words, there will be `Memento` objects only for those objects which have called either the `notifyChange()` or the `notifyDestruction()` function. As an optimization, the library does not create mementos for objects that have been created on the current undo level, since the reverse operation for the creation is deletion, which will remove the object anyway.

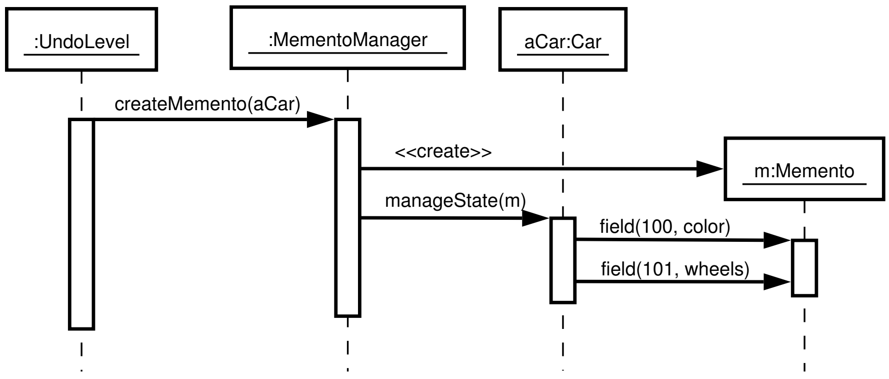

Figure 5.4: Storing the State of a `Car` to a `Memento`. When `notifyChange()` or `notify-Destruction()` is called, the state of the object is stored to a `Memento` object.

For instance, when the `Car` class from Section 4.9 calls the `notifyChange()` function, the call will be delegated to the active `UndoLevel` like in Figure 5.2. As shown in Figure 5.4, the active undo level will then create a memento for the car using its `MementoManager`. The `MementoManager` stores the state of the car in the same way as in normal serialization. The only exception is that it will not create mementos for the wheels of the car, unless they too call the notification functions.

Only the first instance of a `notifyChange()` or `notifyDestruction()` will cause the object state to be stored. However, further calls are typical since most mutating member functions will start by a `notifyChange()` call, and all destructors in an inheritance relationship will call the `notify-Destruction()` function. To deal with these issues, state saves beyond the first one are skipped. This is adequate, since the states are being stored for to the purpose of reverting back to the state *before* the undo level.

In order to guarantee that the full state of an object can be stored before destruction, `notify-Destruction()` must be called from the destructor of the most derived class. Otherwise the object will already be partially destructed, which at best means that all member variables cannot be read properly. A good practice is to call the `notifyDestruction` function from the destructor of *all* classes. As explained previously, this will not cause problems because the subsequent calls made in the based classes are ignored.

5.4 Reverting to an Earlier State

When the client application calls the `undo()` function of `StateManager`, the changes marked to the most recent undo level need to be reverted. Undoing more than one undo level is accomplished by calling the undo function several times. The following procedure is used for implementing the domain model state reversal:

1. *Delete created objects*, unless they have already been deleted in the same undo level.
   - (a) Reset object state to the factory default in order to prevent the class destructor from failing. Invoking a class destructor in an arbitrary state may lead to crashes, or may put the domain model to an invalid state. The library requires that the state after factory creation is adequate for the destructor to execute correctly, without side-effects.
   - (b) Apply the C++ `delete` operator, which calls the destructor of the class and frees the memory allocated to the object.
2. *Create deleted objects*, unless they were originally created within the same undo level. Creation is achieved using the `AbstractStateFactory` interface and the class identifier of the object.
3. *Restore states of modified objects* by setting their mementos using `MementoManager`. Also the states of the objects created in Step 2 are set.

The ordering between Steps 1 and 2 is not important, but the presented ordering usually has lower peak memory use because created objects are deleted before deleted objects are recreated.

In Step 1 it is possible that some of the objects that were created during the undo level were also deleted within the same undo level. This case is detected by the procedure and the deletion is skipped. Similarly for Step 2, if an object which was deleted on the undo level was also created in the same level, there is no need to recreate it since the goal of the procedure is to go back to the state before the undo level.

After the undo level has been undone, it is removed from the history list of `StateManager`, and the previous undo level becomes active, that is, it will start recording further changes until another undo level is started.

Chapter 6

Library Evaluation

In the previous two chapters we presented a serialization library which supports undo. The design is based on a generic serializable memento. The goals for our work were to provide serialization and undo facilities whose use requires only moderate effort from the application developer. In the scope of the thesis detailed studies of serialization mechanisms and multi-user and non-linear undo models were ruled out.

This chapter is organized as follows. We start by comparing the designs of our serialization and undo facilities to the background presented in Chapters 2 and 3, respectively, and give improvement ideas. We will also include analysis about performance related issues and think about how the C++ language has affected our decisions. Then we inspect the effort requirements introduced by the library.

6.1 Comparison to Serialization Requirements

Our serialization library can provably be used for persistence, that is, storing and restoring the domain model between program runs. The implemented serialization mechanism serializes the data to a flat compressed binary file. The library can also be employed for extracting mementos from the objects for the purposes of an undo support. However, other possible uses have neither been designed nor implemented. The most obvious subject for further studies would be implementing remote procedure calls.

In Section 2.3 we described several requirements a serialization library needs to address. In the next sections we discuss how our design meets them. The comparison is outlined in a similar order as in Section 2.3 except for most effort related observations, which are separately treated in Section 6.3.

6.1.1 Coupling

Coupling between the serialization library and the domain model is quite strong, as domain model classes need to derive from a base class defined in the serialization library and override its member functions. This avoids many complications, as explained in Section 2.3, but increases coupling and forces all domain model classes to become *polymorphic*. Polymorphic classes have at least one dynamically bound member function. For implementing dynamic binding, additional data needs to be added to the class layout. The overhead of the additional data may be excessive in very small classes, such as ones representing color or date values. There is also a very strong bias towards dynamically allocating the serializable objects from the heap. This may be problematic in situations where stack-allocated objects would be more suitable.

The problems caused by inheritance can be avoided by non-intrusive serialization. It is supported by the library, but its use causes losing most of the advantages of the library as explained in Section 4.2. Improving the support for non-intrusive serialization could probably be done using similar techniques as those in the Boost serialization library[1], but it would require further research for both the serialization and undo facilities. Consequences for the effort of the application developer and to the error-proneness would require special attention.

However, even if the serialization library is tightly coupled to the domain model, the serialization mechanism is well separated, so a major problem caused by coupling is avoided. Also the domain model encapsulation is theoretically preserved, as the classes are free to check their integrity constraints during deserialization.

It is possible to inspect the content of the internal state of domain model objects by using the serialization library, which breaks encapsulation. However, this problem is inherent in all generic purpose serializers, and in languages with reflection. In practice this is not a problem, because the purpose of encapsulation is to facilitate information hiding, not enforce it.

6.1.2 Versioning

Versioning is supported to the extent of schema evolution, that is, old schema versions can be read by later program versions, but in the general case earlier program versions cannot read later schema versions. The implementation consists of two parts: a global schema version number and a validation pass. The version could be made local so that each class would have its own schema version number. This would enable independent development for the classes. The current approach was adequate because the library is used only in in-house products (see Chapter 7).

The validation pass is enough for handling most versioning needs. New class member variables can be default-initialized. The default-initialization can access other objects in the domain model so that it is guaranteed that they have been deserialized. However, cases which require that other objects have been validated as well require manual trickery.

Also since the versioning code is supposed to be placed inside the domain model classes, it is possible that legacy versioning code becomes a burden by bloating the domain model. A way around this is to drop the support for very old schema versions in the actively developed code base. As it is probably required that all previous versions are supported by the software, older versions of the code base can be used as external conversion programs. The external conversion could then be triggered when very old schemata are encountered.

[1]See Section 2.5.

6.1.3 Proper Handling of Pointers

Shared objects and cyclic data structures are properly handled, since the library checks that each object is only processed once. Serialization through base class pointers is handled properly because the serialization function is dynamically bound. The library supports serializing by reachability, that is, it can also serialize the objects which are pointed to from member variables of the objects explicitly passed to the library. The reachability-based serialization is not mandatory though, since the undo support requires that only the local state of the explicitly passed objects is stored.

The way the library checks that each object is only processed once uses an object tracking map, which is not cleared after deserialization is finished. This is described in Section 4.6. Not clearing the map causes a memory overhead, since each object will occupy an entry in the map during its lifetime. It will also cause a performance penalty, since the entry needs to be added to the map on construction and removed on destruction.

6.1.4 Identifiers

The library forces the user to provide integer identifiers for all classes and their member variables. Each object is automatically assigned an object identifier. These three identifiers should enable supporting a wide variety of serialization mechanisms, since all data in the domain is extracted in an identifiable way. For instance, without identifiers for member variables, additional effort would be required to group the variables into the same columns of database tables or into XML tags with the same name.

A drawback of our implementation is that because the identifiers are integers, additional mapping is needed to make them meaningful human- readable class or variable identifiers in mechanisms such as XML or databases. Also it is questionable to force the developers to assign integer identifers to classes and their member variables. Supporting string identifiers and initializing them with macros to the names of the member variables by default would improve the situation. In this case, two additional things should be considered. First, the developer should be able to change the default identifier string for such purposes as schema evolution. Second, strings may reduce performance when compared to integers, unless special attention is paid.

Having member variable identifiers assists in dropping variables during schema evolution, as no manual skipping code during deserialization is necessary. However, even though this eliminates subtle errors, there is still room for improvement. It can be argued that the scheme as-is is too *implicit*, so for debugging purposes a `droppedField(identifier, inVersion)` function could be added to the `Memento` interface. The function could do a check that only schema versions below the second argument contain the field. Another aspect which could be considered too implicit is the way transient member variables are handled. They are just not listed in the `manageState()` function. Ideally the library could check that all member variables are explicitly listed to either be serialized or transient, but unfortunately we have not come up with a clean way of implementing this in C++.

The object instantiation support based on the `AbstractStateFactory` and the class identifiers works, but has a disadvantage of promoting an implementation which bundles all instantiation code into the same class. This may result in very large files with dependencies to all of the domain model classes, which in turn causes unnecessary recompilation of the factory. The situation could be improved by implementing a generic factory, which allows the domain model classes to register instantiation functions based on their class identifiers. The generic factory would then call the correct instantiation function when a class identifier is presented to it. With this approach, it should be enforced that all classes which serialize themselves also provide an instantiation function to be used at deserialization-time. We feel that the Boost-style automatic registration is not worth doing because of the increased error-proneness (see Section 2.5).

6.1.5 Error Handling

Obvious failures, such as running out of disk space, corrupted file contents and running out of memory encountered during serialization and deserialization abort the operation and are reported to the client application by throwing exceptions. Some major failures, such as detecting dangling pointers are reported by crashing the program. Certain situations which could be considered faulty, such as trying to read a non-existing member variable from a memento or not reading all data in a memento are currently ignored. Ideally the client application should be able to specify the library behavior in these cases.

6.1.6 Serializable Class Interface

The memento interface presented to the serializable domain model classes is quite straight-forward and comprehensive. It supports the built-in types of the language and the most relevant standard library types. Pointers are supported but C++ references are not, because C++ references are not reseatable. However, two useful features are not currently supported, namely containers of containers and containers of plain objects. Containers of pointers to objects are supported. The reason for omitting the former is that the compiler used for originally implementing the library had problems with its support for C++ templates. A workaround for using containers of containers is to wrap the inner containers to a class derived from `StateObject` and use a container of pointers to such objects. Another workaround is to manually transform the data structure to a supported one. For instance, a set of set of integers can be transformed into a multimap from integer to integer, where the first integer describes the set into which the second integer belongs.

The latter omission was intentionally made due to its error-proneness. A container of plain objects is represented in C++ as code such as `std::list<Class>`, where as a container of pointers to objects is represented as `std::list<Class*>`. In the former case, the container takes care of memory management. The problematic case occurs when the application has pointers to the elements in the container of objects. This will require that the objects that contain the containers of objects are deserialized *before* the objects that contain the pointers to the elements in the containers. Otherwise the container has not yet instantiated elements, which will cause them to be instantiated as freestanding objects. Later, when the object containing the container is deserialized, the container will reinstantiate the objects. Handling the situation is somewhat difficult, but could be implemented by fixing the objects in another deserialization pass. Multiple passes would also be required for correctly handling pointers into non-container member variables.

To avoid such complicated scenarios, the support for containers of objects was omitted completely. The workaround is to use containers of pointers to objects. This is also the only possible action in many other programming languages, such as Java. In C++ this has the downside of forcing dynamic allocation of the container contents. Dynamic allocation is not as cheap in typical C++ implementations as it is in modern garbage-collected heaps.

A special case of serialization takes place when domain model classes contain handles to external resources, such as file or socket descriptors and thread or window identifiers. The library does not support serializing these variables directly, so the client application developers need to write their own serialization functionality in these cases. Typically external resources can be described by serializable data types and then reclaimed during deserialization. For instance, a file can be serialized by serializing the filename string and the file position integer, and deserialized by opening the file and seeking to the correct position. Special attention needs to be paid to error handling in this case.

6.1.7 Simultaneously Active Mechanisms

Serialization mechanisms are applied to instances of the `MementoManager` class. When serializing, the object will contain the state of the domain model. As serializing will only read the contents of the object, several mechanisms can be used consecutively or even concurrently.

Furthermore, as the contents of the `MementoManager` are not subject to change, the domain model can be updated concurrently to applying serialization mechanisms. This allows for instance improving the responsiveness of saving files. While the user waits, the contents of the domain model would be extracted to a `MementoManager`. Then a new thread is started for writing the data to disk, and the user interface is free to receive user commands at the same time. However, in this case crashes due to the UI thread while the background thread is working may cause data loss.

6.1.8 Serialization Mechanisms

Currently we have only implemented a serialization mechanism for writing the contents of a `MementoManager` to a compressed binary flat file. A comprehensive library would also include an XML mechanism. Furthermore, it would be interesting to design and implement a mechanism for persistence to relational databases, as they are widely used. A potentially useful approach would be to use a generic object-relational mapping scheme [5] at least in the first version. Later, more advanced mapping schemes could be investigated, if needed.

A database mechanism would open up a wide range of further studies. For instance, lazy deserialization, multi-user access and other database-specific features mentioned in Section 2.3.9 and studied in [43] could be investigated and implemented.

A particularly urgent topic is related to the performance and scalability of domain models using our library. Currently we do not use lazy deserialization, which forces the whole domain model to be in memory at the same time. While this may be adequate in many cases, particularly when the program is not supposed to run strictly on top of a database, it would be interesting to design proxy schemes using smart pointers in ways which would hide the laziness from the application programmers.

An aspect we ignored due to the limitations on mechanism studies was standards compliance. For instance, an XML serializer could use the Object Data Management Group (ODMG) endorsed format specified by Bierman [12] and binary mechanisms could be added for importing files in the formats used by other available libraries, such as MFC and Boost.

6.2 Comparison to Undo Techniques

We use the partial checkpoint implementation strategy for undo. The domain objects themselves notify when they are about to change, and the library stores their states prior to the modifications. The states are stored using the memento pattern, and the implementation is directly reused from the serialization library. A benefit of this approach is that the application developer effort is very low compared to for instance using commands and mementos in the traditional way. The memento class we use is generic, that is, it can be used for any domain model class. It is also serializable, so serialization and undo are achieved with partially overlapping effort. The effort required for using our library is further discussed in Section 6.3.

6.2.1 Directions for Improvements

The undo model we have implemented is a restricted single-user linear undo without branches. The most obvious limitation is the lack of redo. However, redo could be implemented on top of the current implementation without further requirements from the application developers. The basic idea would be to reverse the changes made during undoing. Also branches in the history list could be allowed in the current design, but redo would have to be implemented first.

A doable direction towards non-linear undo would be to implement the direct selective undo of Berlage [11], which is described in Section 3.2.6. Berlage too states that his approach can be added afterwards, and that it can even be added to just individual selected commands without caring about other commands. However, such an extension would have certain drawbacks. The required effort would increase quite much because implementing the new member functions to all command classes is far from trivial. Another disadvantage is that the direct selective undo would require using commands, which is not required by the current library.

For supporting multi-user undo, a starting point could be to use a locking scheme, in which only one user at a time can edit the domain model [2]. Using an implicit lock which automatically locks the domain model during changes, combined with immediate updates of remote changes would make implementing global multi-user undo almost transparent for the application developers. Local undo models would require a non-linear undo, for instance the direct selective undo of Berlage.

For improving the implementation, there are a couple of things to consider. Ideally the `StateManager` would be split into two. First, the undo-related responsibilities would be moved to another class, for example `UndoManager`. The second part would consist of the remaining parts of the current `StateManager`. Another obvious improvement would be to provide a command and command processor implementation with the library.

6.2.2 Undo Levels

The current undo support can be driven by commands and a command processor, as explained in Section 5.2, but it is in no way dictated by the library. As far as the library is concerned, it is enough that the client application calls the `startUndoLevel()` function before modifying the domain model.

However, should the client application developers decide to go with commands, they will quickly observe that part of the work has already been implemented for them. The `StateManager` takes care of maintaining a list of undo levels, which is a responsibility of a command processor if undo levels are viewed as sets of commands. Also the client application developer need not store partial checkpoints inside the command objects, since the domain model is already equipped to do it for itself.

6.2.3 Relaxing Exception Safety Guarantees

An additional feature of capturing the changes made to the domain model to undo levels is that it can be used for implementing transaction semantics. Many actions result in several consecutive modifications to the domain model. If a modification fails, and for instance an exception is thrown, the changes made by prior modifications within the same undo level can be rolled back by using the undo facility. This is similar to rolling back instead of committing in databases. This provides a shortcut to the application developer: the *exception safety* guarantees can be relaxed. Basic exception safety guarantee maintains the class invariants when an exception is thrown and does not leak resources, such as memory [55]. Strong exception safety guarantee includes the basic guarantee and additionally leaves the class to the state in which it was before the whole operation. Using our undo facility, providing the strong guarantee in the scope of the whole domain models becomes possible.

6.2.4 Coupling

The user interface code can enjoy the benefit of not having to track changes it makes to the domain model, since the domain model will do it by itself by using notify calls to our library. This not only avoids unnecessary coupling, but also reduces the encapsulation problems discussed in Section 3.2.3.

However, this adds an unrelated responsibility to the domain model and increases its coupling to the serialization library. Furthermore, the notify calls are somewhat error-prone, as we argue in Section 6.3. We also present possible workarounds for the problem in the same section.

An interesting observation is that the `manageState()` function can be used for reading and writing the member variables of objects much like reflection facilities in languages such as Java. Separating this reflection aspect from the serialization functionality would increase the modularity and cohesion of the design.

If reflection were present, either as built into the language or as manually implemented by a separate facility, it could probably be exploited in the design. For serialization purposes the current `manage-State()` function also deals with other responsibilities than plain reflection, such as versioning. With reflection, it could perhaps be possible to provide a default implementation without versioning support for the initial versions of classes. For undo however, it might be possible to directly use the reflection facilities to implement our approach for undo. In fact, Völter [62] has implemented something similar to this as explained in Section 3.2.2, but his approach inconveniently requires creating separate state classes and manually implementing memento getter and setter functions.

6.2.5 Performance

Our undo solution has an impact on both the computational performance of the client application and on its memory use. The performance penalty is caused by the notify calls and particularly the serialization which is done within the notifies for storing the object states before changes. Also checking whether the state of an object has already been stored to the currently active undo level takes some time, and may be problematic if the domain model function which makes the modification is otherwise trivial and is called frequently. However, the performance penalties are proportional to the change set size, which makes the approach feasible even in large instances of domain models.

Another aspect of the performance is the time it takes to undo the changes made on an undo level. The time is again proportional to the number of changes made within the undo level. As an absolute value, the time will be a fraction of the time it would take to deserialize a domain model whose size equals the change set size. This is because the deserialization is done from the main memory and as the undo level is stored as a `MementoManager`, the serialization mechanism need not be invoked. However, there is room for improvement in the case of undoing multiple undo levels at a time. The current implementation undoes the levels one by one, while it would suffice to first calculate the combined effects of all of the undo levels to be undone, and then just revert the net result from the domain model.

While it may appear that storing the full state of each modified domain model object is expensive in terms of memory use compared to just storing the high-level command that reverses the individual changes made, the memory use of our undo facility has some good properties. First, it is possible and even mandatory to actually delete the objects that are removed, because the `notifyDestruction()` call will create a memento of the object before its destruction. In traditional command-based undo implementations it easily becomes tempting not to delete, but actually store the deleted objects in the command so that undeleting them later would be possible.

Second, our solution allows arbitrary optimizations to be made on the undo levels once they are no longer active. That is, undo levels which are not very close to the current state of the history list can be subjected to serialization mechanisms. Serializing an undo level and compressing the output will reduce the memory use by an order of magnitude, and is very easy to implement. Furthermore, it is possible, although non-trivial, to create optimized compression strategies based on the fact that the differences between adjacent undo levels are not likely to be very big. An easy way to further reduce memory use is to swap very old undo levels to the hard disk. This would allow practically unlimited undo history to be implemented.

While we were worried about performance implications of the intermediate representation in Sections 2.3.2 and 2.3.3, the `MementoManager` turns out to be useful for improving the total system responsiveness during the optimization of the memory use of undo levels. The intermediate representation causes some additional data copying and avoiding it may yield marginally better performance, but the intermediate form also allows the serialization mechanism to run *in a separate thread* of the application. This means that the user interface thread may continue responding to user requests while the undo levels are compressed and swapped to disk in a background thread.

The data loss problem mentioned in Section 6.1.7 is not relevant when swapping undo levels to disk, because if the application crashes, losing the undo levels is hardly a problem. However, the swapping strategy requires further studies. It is non-trivial to determine when and what undo levels to compress or swap to disk.

6.3 Effort Analysis

The goal for this thesis was to present a design which requires only moderate developer effort to implement serialization and undo, and which minimizes the distraction caused by them in everyday work. To evaluate how the goal was met, we start by summing up the steps required to have serialization support in a client application. First, all domain model classes need to be modified as follows:

1. Derive the class from `StateObject`.
2. Override `getClassId()` and make it return an integer which is unique within the domain model.
3. Extend the domain model factory to return a new instance of the class according to the class identifier.
4. Override `manageState()`, add a call to the base classes `manageState()` and add one line per member variable for serializing and deserializing purposes.
5. Call `notifyDestruction()` in the beginning of the destructor.
6. Call `notifyChange()` in each mutating member function.

Additionally, if more than one instance of the domain model can be in the memory at the same time, for instance if the program uses a multi-document interface (MDI), the `StateObject` constructor needs to know its `StateManager` instance. This is required, because otherwise it is difficult to know to which undo level the changes made to domain objects should be put, as each `StateManager` has its own active undo level. Versioning and very unusual data structures may cause additional effort requirements.

If a new undo level is wanted for each logical user interface action, starting undo levels inevitably requires manual intervention from the application programmer. The reason is that the boundaries between transactions cannot be automatically determined. Relaxing undo level starting could be achieved by automatically starting new undo levels for instance after the user has made changes for three seconds. In this case the undo levels would not contain logical user interface actions, but a set of changes made during a three second editing session. The approach results in nameless undo levels with varying content, but would perhaps suffice in some cases.

On the other hand, there are several ways how manual calls to `notifyChange()` and `notify-Destruction()` could be avoided. Using memory protection similarly to pointer swizzling or the undo of Zundel[2] and calling the notify functions for all objects in the accessed memory pages would be one solution. This approach would be quite similar to the patented method of Zundel though. *Method call interception* (MCI) [35] or *aspect-oriented programming* (AOP) [20] can be used for automatically prepending member function calls with the respective notify calls. This is a viable way to avoid the manual work once these techniques become mainstream. Nowadays in most programming languages, including C++, their use requires a preprocessor, which makes the approach less appealing.

The second class of modifications applies to the user interface. Saving and loading buttons and menu items need to call the respective functionality in the library, if persistence is required. Similarly, the undo button needs to call related library functions. Less trivially, before making changes, new undo levels need to be started by calling `startUndoLevel()`.

Most modifications required for implementing serialization and undo in client applications can be considered quite mechanical, that is, there is not much need for careful thinking. The biggest challenge is remembering to perform all the required steps. For serialization, this is the common case in available libraries. However, in the case of undo our methods have clear advantages, which can be emphasized by comparing our solution to traditional uses of the command and memento patterns for undo.

Traditionally, in order to get a command class to support undo, the application developer needs to override and implement three member functions: `execute()`, `undo()` and `redo()`. Implementing the `undo()` function is the most difficult part as explained in Chapter 3, even when the memento design pattern is used. Using our library, the `undo()` and `redo()` functions need not be implemented at all. In fact, it is not mandatory to use the command pattern at all.

Implementing mementos in the traditional way requires creating a new memento class per each domain model class. The memento class needs to able to hold the whole state of the domain model class for which it is created. Using the memento classes from command classes is not trivial either. First, the command class making changes to the domain model needs to know which domain model objects it will be modifying. Second, it needs to explicitly store mementos for them, and just for them. With our library, a function call to `startUndoLevel()` suffices.

The reduced undo effort is possible because restricting the undo support to linear single-user undo allows the library to implement undo using the semi-automatically captured physical modifications to the domain model state, which means that the undo mechanism is implemented by the means provided by the serialization library. This frees the application developers from having to implement the undo mechanisms for each command themselves.

[2]See Sections 2.6 and 3.2 for descriptions of these methods.

Chapter 7

Case Studies

We have applied the presented design for serialization and undo to two commercial project management applications between the years 2001 and 2005 at Dynamic System Solutions Ltd[1]. In this chapter, we share our experiences with the used implementations of the design. We start with brief overviews of the case applications. The purpose of the overviews is to show to which kind of real world applications the library was deployed. Next, we discuss how the design was implemented and deployed in the cases of the applications. Finally, we discuss positive experiences as well as situations where the library or the way it was used would require further work.

7.1 Case Applications

The two project management applications to which we have deployed the library are specialized for construction business. The first application, DynaProject, is designed for scheduling and production management of building construction. The second application, DynaRoad2, is a schedule and mass haul optimization system for highway and railroad construction. The applications run in the Microsoft Windows operating system, and are in both commercial and educational use in Finland as well as internationally. For instance, DynaProject is used in the Kamppi Center project, which is one of the largest single city center projects in Europe. The largest reference project for DynaRoad2 is the Kerava-Lahti railroad connection, which is also the largest railroad project in the history of Finland.

The applications include most of the features in contemporary project management programs with a standalone GUI. Additionally, due to their industry-specific nature, they include various unique extensions and specializations on top of the basic project management functionality.

The user interfaces and domain models of the applications have been divided into separate layers. Additionally both layers depend on various technical libraries, which constitute the technical services layer. The sizes of the applications are listed in Table 7.1 as thousands of non-blank non-comment source lines of code (KLOC), excluding the source code of the used third-party components. While the source lines of code is commonly acknowledged to be a bad metric for software size, it should

[1]`http://www.dss.fi/`

give a rough feeling about the orders of magnitude of the applications.

Application	*Tech. Serv.*	*Domain Model*	*UI*	*Total KLOC*
DynaProject	34	71	122	227
DynaRoad2	26	59	77	162

Table 7.1: Case Application Sizes. The sizes are given as a thousands of non-blank non-comment source lines of code.

While the interpretation of the figures in Table 7.1 depends on numerous factors, it can safely be stated that our applications are semi-large in size. This statement is further backed up by the extent and number of features implemented in the programs compared to other available project management systems.

7.2 Library Deployment

We started DynaProject with a different approach for persistence, and later changed the implementation to use the design presented in this thesis. To DynaRoad2, we reimplemented the library with various improvements mainly focused on improving reusability and reducing architectural coupling.

Undo was added to both applications much later than serialization. In both cases, the operation was quite painless. We implemented the extensions required for the library and made the necessary modifications to the applications. Particularly because undo is considered to be architecturally sensitive [22], it is interesting to observe that a delayed deployment is still doable with moderate effort. The late addition is also a sign of agile principles since the features were added only after their priorities became high enough.

DynaProject originally started undo levels in a brute-force ad-hoc manner by just calling `startUndoLevel()` all around the user interface code before making modifications to the domain model. This was somewhat problematic as it made modifying the code related to domain model changes difficult and caused code duplication. Recently, we have started to migrate towards using commands.

DynaRoad2 on the other hand, being written later, started with a centralized place for making changes. All of the changes made by the user interface were routed through a controller class. While this avoided code duplications and made undo level controlling easier, it had the drawback of causing the controller class to become very large. Currently we are using commands in DynaRoad2, too.

7.3 Main Impression

We have been able to use the design presented in this thesis successfully for both persistence and undo. The target applications are not simple test programs, so it appears that conclusions drawn from experiences we have had with them can be considered realistic.

For most of the time, the library has been out of the sight and the minds of the developers. This is good, because when the library is not noticed, it is working from the effort point-of-view. Furthermore, we have not had to postpone the implementation of undo or serialization because of schedule constraints. This would imply that when the library is used, it is not seen as too demanding.

We have noticed additional benefits of the library in addition to achieving persistence and undo. First, the object tracking done by `StateManager` is convenient for tracing memory leaks in the domain model. When the domain model is deleted, it should delete all the objects in it. However, if it misses something, the living objects will still exist in the object tracking map. The contents of the map can then be printed out when the `StateManager` is destructed which allows the developers to see the memory leaks.

Another additional use involves dialogs with a cancel button. Normally, the changes made in a dialog would be applied to the domain model when the OK button is clicked and the dialog is closed. However, if the contents of the dialog depend on each other, this may be unintuitive for the user. For instance, if a duration changing dialog allows editing both the end date and the duration in work days, changing either one of them in the dialog should update the other. Making the calculations in the dialog would duplicate domain model code. With our library, we can start an undo level when the dialog is opened and apply the changes made by the users immediately when they do them. Then the OK button needs to do nothing and the cancel button can use the undo functionality to revert the modifications done.

7.4 Problems

Although the library has been useful in many cases, we have observed several situations where its behavior could be improved. In this section, we outline some of the problematic situations with proposed solutions.

7.4.1 Versioning

The first versions of the serialization library date back to the year 2001, and the oldest schemata still supported are from the early 2002. Since then, we have obviously done much development on the applications. While we have been quite successful in supporting old file versions, there have been some challenges as well.

First, during the years our domain model classes have accumulated quite much code for handling old schema versions. The domain model classes that have been in use for a long time have much code devoted to reading old versions of the classes. Also there is a significant number of classes not used anymore in the latest version of the domain model. Those classes cannot be removed because the data in them is loaded and converted to the current domain model classes. The most obvious way of fixing this domain model bloat would be to create a separate conversion application which would use the current domain model and then strip the legacy schema support from the main application (see Section 6.1.2).

Second, updating the schema version of an object sometimes requires investigating some other object in the domain model. While the validation pass has taken care of most of these cases, we have encountered situations where the other object needs to be validated before investigating it. We have worked around this by manually adding flags which track the validation status of the class, and by manually calling the validation functionality in the investigated classes. However, this is clearly not the cleanest solution. The schema evolution should allow the client application to communicate the required order of validation.

Finally, the code for reading old schema versions requires testing just like any other code. A good asset for testing would be to have a collection of old files, which could then be deserialized using the latest version of the application. This would require systematically saving project files which use a rich set of the program features. Each release version of the application should add files to the set of these test files.

For implementing the testing, the set of test files could be automatically opened and the automatic unit test suite ran on them. At the very least it should be checked that the application does not crash when the files are opened. Additionally, a human tester should check that the files are converted correctly to the new schema.

In practice, collecting the test set, implementing automatic testing and getting human testers to focus on schema evolution has been quite difficult for us. Many of the bugs in the schema evolution code have been discovered quite late. A reason for this is probably that verifying technical details such as schema evolution are not considered to be as important as validating new program logic. Fixing this in the code would mainly consist of the automatic testing system to be built, but the other related activities require continuous human effort.

7.4.2 Support for Missing Data Types

There have been cases where the memento interface provided to the client application has not been adequate. Two of these cases, the lack of support for containers of objects and containers of containers were explained in Section 6.1.6. In addition, we have had to serialize certain third party types and certain in-house types to a binary format, and then apply the binary serialization functions. Improving the non-intrusive serialization support would alleviate the application developer effort in these cases.

7.4.3 Bloated Factories

The factories we use to instantiate new domain model objects during deserialization and undo are currently implemented as big switch statements on the class identifiers. This couples the factory to all classes in the domain model and any change in their interfaces causes a recompilation of the factory. In practice this has not been that big a problem, but the generic factory approach introduced in Section 6.1.4 would make the code cleaner.

7.4.4 32-Bit Identifiers

For class identifiers and member variable identifiers, a 32-bit representation is more than enough. However, for object identifiers the situation may be different. The problem is not that there would be more than 2^{32} objects simultaneously in memory, or even in the same domain model instance. Instead, as the object identifiers are increasing integers and the identifiers of deleted objects are not reused, it is enough that 2^{32} objects are created during the whole lifetime of a domain model instance including the time it is in the serialized form.

In DynaRoad2, the serialization library redistributes the identifiers at load-time so that they start from one and there are no gaps in the identifier-space. In practice, this fixes all realistic problems. In DynaProject there is no such redistribution scheme, but we have still not yet witnessed any problems. If problems arise, they are likely to cause data loss.

7.4.5 Forgetting Notifies

A problem we have with undo is that remembering to place the notify calls to mutating member functions and destructors is sometimes hard. Omitting the calls will lead to subtle undo bugs. For instance, suppose that the application creates a new object, makes an existing object point to the new object and does not call any notify functions. When these actions are undone, the new object will be deleted by the undo system because it can automatically detect the creation of new objects. However, the object that points to the deleted object is not reverted to its earlier state and the pointer will be invalid. Trying to use the pointer will probably cause the application to crash. Similar problems may take place if the wrong notify call is used, that is, `notifyChange()` instead of `notifyDestruction()` or vice versa.

Aspect-oriented programming and other techniques mentioned in Section 6.3 would be suitable for eliminating the notify calls altogether. While waiting for AOP to become mainstream, we could implement some debugging facilities to detect missing notify calls. For instance, making a full checkpoint of the domain model before an undo level would enable comparing the differences to the situation after the undo level. The differences could then be compared to the ones reported by the notifies and anomalies could be logged.

7.4.6 Undo Memory Use

DynaProject features a simulation model for risk analysis. DynaRoad2 uses global optimization for finding good mass economy solutions. These are examples of such operations on the domain model that result in large change sets. Also more trivial changes such as changing the begin time of a task in a project management software can cause many changes, because the timings of succeeding tasks are typically dependent on preceeding tasks.

Having several big undo levels or a huge number of smaller undo levels in memory is clearly intolerable due to the memory requirements. We have handled to situation by constraining the number of undo levels in the history list. This limitation could be eliminated if the straight-forward undo level compression and disk swapping ideas of Section 6.2.5 were implemented.

7.4.7 Deletion Procedure

A step in the undo procedure in Section 5.4 is deleting the objects which were created in the undo level. In both DynaProject and DynaRoad2 this is implemented by calling the C++ `delete` operator.

Additionally, in DynaRoad2, the state of the object is reset to the factory default prior to the deletion. In DynaProject this is not done.

In DynaRoad2, we have not witnessed problems because of the deletion procedure, that is, all domain model classes survive the `delete` operator when the object is in the factory default state. On the other hand, in DynaProject there have been several failures because the state resetting step is missing. This is because the object may be in an arbitrary state, including one in which it is not possible to directly delete the object and maintain domain model integrity. For instance, if the removal procedure for some objects requires them to be deleted in a certain order, the undo system will most likely cause failures and even crashes. To fix this, proper state resetting prior to deletion should be implemented.

7.4.8 Additional Effort due to Indirection

Starting a new undo level before making changes to the domain model is something that does not need to be done if there is no undo in the application. Sometimes we have forgotten to start the undo level, which has resulted in missing entries in the history list. Also when the same functionality has been available in several places of the user interface, we have observed that the code which invokes the modification is often duplicated. Finally, when we have added new features to the modification code, such as logging, exception handling and status bar updates, it has been problematic to maintain all relevant existing code.

To address these problems encountered in DynaProject, we have tried two approaches, both of which are based on an additional level of indirection. First, in DynaRoad2 a *bloated controller* [36] *facade* [24] was implemented. The functions of the controller were called by the user interface dialogs and views. The functions then delegated to the domain model for the requested changes. While this centralized the change-making code to one place, it quickly became apparent that the controller class would become very large.

To avoid bloating the controller, we went for the command design pattern (see Section 3.2.1) as it is quite common in many applications. However, compared to an application without undo, the existing bloated controller in DynaRoad2 and the existing brute-force code which just starts undo levels before making changes, commands turned out to be burdening. Compared to other the approaches, the application programmer now needed to create a new class for each domain model change. Also, the amount of existing code which did not use commands was quite impressive.

We are currently making good progress with switching to commands in DynaRoad2. In DynaProject the process will take somewhat longer a time, since there is no existing indirection such as the bloated controller, but instead the change-making code is scattered throughout the user interface code.

7.4.9 Undo Granularity

Another issue we have had to deal with during the transition towards commands is the way undo granularity is handled, particularly in the case of dialogs. We have used a composite command (see Section 3.2.4) for grouping several other commands together and executing all of them within the same undo level.

This turned out to be inadequate because our dialogs sometimes apply changes to the domain model immediately when a widget has been modified. Using commands, this caused several undo levels to appear, which is unwanted behavior. The solution we used is an undo level merging scheme, which allows several command invocations to be combined into the same undo level. This also addressed the problem where a part of the dialog was using commands, while the old code in the same dialog was not. In such a case, invoking the commands caused unwanted undo levels to appear, too.

Chapter 8

Conclusion

Agile development models aim towards producing partially functional versions of the software being developed in early phases of the project. The software is gradually modified according to customer feedback until the resulting product is satisfactory. While this is useful for customers and end-users, it poses new challenges to the application developers. They need to be able to design and evolve the application architecture so that it continues to meet the requirements of software.

Typical interactive graphical user interface applications need to support saving and loading their project files and settings. Serialization is a key component in the implementation. Serialization is also often a major part in architectural decisions. It affects a very large subset of the code base of a typical GUI application. Serialization is also useful for other purposes than saving and loading, such as implementing remote procedure calls.

Another essential concept in modern GUI applications is usability. An architecturally significant part of usability is providing the users the ability to undo their actions. The undo facility allows users to recover from their mistakes and to explore the application at hand without being afraid of breaking something. The latter is an important factor in learning to use new software.

In this thesis, we presented a serialization library based on a generic serializable memento. The library allows application objects to create mementos of their states, and later restore themselves to the states stored within the mementos. Our mementos are serializable, which means that their contents can be transformed to other forms and back again. We have implemented a serialization mechanism which transforms the mementos to a compressed binary flat file, which enables efficient project saving and loading to be supported. It would also be possible to support serialization to XML files and relational databases.

We gave a set of extensions for the serialization library, which enables it to be used for undo. The undo implementation is based on creating mementos of the objects prior to their modification, and when undo is invoked, reverting the objects to their old states according to the memento contents.

Our serialization library is quite on par with features compared to other available implementations. It provides an easy-to-implement interface for client applications, and separates it from the details of serialization quite well. Also it does not set such requirements on the serialization mechanism which would dictate using a relational database. The library also provides an explicit support for schema evolution, that is, the ability to read old versions of the serialized data.

Our undo extensions enable single-user restricted linear undo to be implemented quite conveniently. Most modern undo implementations are based on the command design pattern, which requires the application developer to manually implement the undo mechanism, that is, the code which reverses the effects of the previous action. With our approach, the reversing procedure is semi-automatic.

A major goal was to reduce the efforts of application developers when they deal with undo and serialization, so that they can better focus on providing business value to the customers. To use our library, the developer needs to follow six mechanical steps for each domain model class in order to support serialization and undo in most cases. Additionally, they need to extend the user interface so that it can trigger saving, loading and undo, and set the transaction boundaries of user actions by starting undo levels before making changes.

We observed that our library can be implemented and deployed to existing applications a relatively late stage of a project. Also our serialization library supports conveniently reading old schema versions. Furthermore, our serialization and especially our undo require less effort from the developers than many other approaches, which enables developers to focus more on domain issues. It is safe to conclude that the library can be used in agile development projects.

We implemented the design in two commercial project management applications targeted to the construction industry. In both cases, we were successful and also found new uses enabled by the library. We also discovered several areas with room for improvement, and gave directions for further studies.

There are many directions for future development of the library, and for the design ideas it addresses. They are presented in Chapters 6 and 7. Major improvements are related to exploiting the features provided by database management systems more thoroughly. The challenge is to retain relatively high invisibility for the application developers so that they do not need to worry about the details of the database management system. Multi-user scenarios are particularly interesting for both the serialization and undo portions of the library. The memory use of the library could also be refined by supporting lazy proxies in serialization and threaded undo level processing in undo. Also remote procedure calls should be studied further.

Bibliography

[1] A. Larsson *et al.* Dia, a diagram creation program. `http://www.gnome.org/projects/dia/`. Last checked April 9, 2005.

[2] G. D. Abowd and A. J. Dix. Giving undo attention. *Interact. Comput.*, 4(3):317–342, 1992. ISSN 0953-5438.

[3] Agile Alliance. Manifesto for agile software development. `http://www.agilemanifesto.org/`. Last checked April 9, 2005.

[4] S. W. Ambler. The design of a robust persistence layer for relational databases. Nov. 2000. Available at `http://www.ambysoft.com/persistenceLayer.pdf`. Last checked April 9, 2005.

[5] S. W. Ambler. Mapping objects to relational databases: O/R mapping in detail. 2005. Available at `http://www.agiledata.org/essays/mappingObjects.html`. Last checked April 9, 2005.

[6] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, 1983.

[7] M. P. Atkinson and R. Morrison. Orthogonally persistent object systems. *VLDB Journal*, 4 (3):319–401, 1995.

[8] S. Beal. s11n serialization library. `http://s11n.net/`. Last checked April 9, 2005.

[9] K. Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10):70–77, Oct. 1999.

[10] E. V. Berard. Abstraction, encapsulation, and information hiding. *Essays on Object-Oriented Software Engineering*, 1, 1993.

[11] T. Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Transactions on Computer-Human Interaction*, 1(3):269–294, Sept. 1994.

[12] G. M. Bierman. Using XML as an object interchange format, 2000. Available at `http://www.odmg.org/`. Last checked April 9, 2005.

[13] K. Brown and B. G. Whitenack. Crossing chasms: a pattern language for object-RDBMS integration: the static patterns. *Pattern languages of program design*, 2:227–238, 1996. Available at `http://www.ksc.com/article5.htm`. Last checked April 9, 2005.

[14] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons Ltd, 1996.

[15] D. Chen and C. Sun. Undoing any operation in collaborative graphics editing systems. In *GROUP '01: Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*, pages 197–206. ACM Press, 2001. ISBN 1-58113-294-8.

[16] R. Choudhary and P. Dewan. A general multi-user undo/redo model. In *Proceedings of European Conference on Computer Supported Work*, pages 231–246, 1995.

[17] S. M. Clamen. Data persistence in programming languages – A survey. Technical report, Pittsburgh, PA, 1991.

[18] A. Dix, R. Mancini, and S. Levialdi. The cube - extending systems for undo. In *Proceedings on the 4th Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSVIS)*, pages 119–134. Springer-Verlag, 1997.

[19] W. K. Edwards, T. Igarashi, A. LaMarca, and E. D. Mynatt. A temporal model for multi-level undo and redo. In *UIST*, pages 31–40, 2000.

[20] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001. ISSN 0001-0782.

[21] X. Faulkner. *Usability Engineering*. Palgrave, 2000. ISBN 0–333–77321–7.

[22] E. Folmer and J. Bosch. Architecturally sensitive usability patterns. In *Proceedings of VikingPLoP 2003*, Sept. 2003.

[23] E. Franconi, F. Grandi, and F. Mandreoli. A semantic approach for schema evolution and versioning in object-oriented databases. *Lecture Notes in Computer Science*, 1861: 1048–1062, 2000.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995. ISBN 0201633612.

[25] H. Hakonen, V. Leppänen, and T. Salakoski. Object integrity while allowing aliasing. In *The 16th IFIP World Computer Congress International Conference on Software: Theory and Practice, ICS'2000*, pages 91–96, 2000.

[26] D. R. Hipp. Appropriate uses for SQLite. `http://www.sqlite.org/whentouse.html`. Last checked April 9, 2005.

[27] *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. IEEE Standard 1471-2000, 2000.

[28] J. E. Archer, Jr., R. Conway, and F. B. Schneider. User recovery and reversal in interactive systems. *ACM Trans. Program. Lang. Syst.*, 6(1):1–19, 1984. ISSN 0164-0925.

[29] J. Walnes *et al.* XStream. `http://xstream.codehaus.org/`. Last checked April 9, 2005.

[30] JBoss Inc. Hibernate. `http://www.hibernate.org/`. Last checked April 9, 2005.

[31] JSX Enterprises Inc. Java serialization to XML (JSX). `http://www.jsx.org/`. Last checked April 9, 2005, .

[32] JSX Enterprises Inc. Jurassic phoenix. `http://www.jsx.org/jurassicphoenix/`. Last checked April 9, 2005, .

[33] K. Wuestefeld *et al.* Prevayler. `http://www.prevayler.org/`. Last checked April 9, 2005.

[34] K. Koskimies and T. Mikkonen. *Ohjelmistoarkkitehtuurit*. Talentum, 2005. ISBN 952-14-0862-6.

[35] R. Lämmel. Semantics of Method Call Interception. In *Workshop Aspekt-Orientierung der GI-Fachgrupppe 2.1.9 Objektorientiere Software-Entwicklung, 3.- 4. Mai 2001, Universität Paderborn*, 2001. Technical Report tr-ri-01-223 Universität-Gesamthochschule Paderborn.

[36] C. Larman. *Applying UML and Patterns, 2nd ed.* Prentice Hall, 2002. ISBN 0130479500.

[37] J. loup Gailly and M. Adler. The zlib compression library. Available at `http://www.gzip.org/zlib/`. Last checked April 9, 2005.

[38] R. Mancini, A. Dix, and S. Levialdi. Reflections on undo. Technical Report RR9611, 1996.

[39] K. Marquardt. Neglected architecture. In *Proceedings of VikingPLoP 2003*, 2003.

[40] T. Meshorer. Add an undo/redo function to your Java apps with Swing. Available at `http://www.javaworld.com/javaworld/jw-06-1998/jw-06-undoredo.html`. Last checked April 9, 2005, June 1998.

[41] Microsoft Inc. Microsoft foundation class library.

[42] T. Morris. Implementing undo/redo - the docvars method. `http://www.codeproject.com/docview/undoredo_demo.asp`. Last checked April 9, 2005.

[43] S. Nurmentaus. Integrating object persistence to relational databases. Master's thesis, Helsinki University of Technology, May 2004. Also available at `http://sampo.nurmentaus.net/dippa.pdf`. Last checked April 9, 2005.

[44] H. Pasanen. Highly interactive computer algebra. Master's thesis, Helsinki University of Technology, 1992.

[45] A. Prakash and M. J. Knister. A framework for undoing actions in collaborative systems. *ACM Trans. Comput.-Hum. Interact.*, 1(4):295–330, 1994. ISSN 1073-0516.

[46] J. R. Pugh and C. Leung. Application frameworks: experience with MacApp. *SIGCSE Bull.*, 20(1):142–147, 1988. ISSN 0097-8418.

[47] R. Ramey. Boost serialization library. `http://www.boost.org/`. Last checked April 9, 2005.

[48] J. F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.

[49] M. Rontu. Visual queries for a student information system. Master's thesis, Helsinki University of Technology, Apr. 2004. Also available at `http://www.cs.hut.fi/Research/SVG/SEEQ/downloads/mrontu-thesis.pdf`. Last checked April 9, 2005.

[50] K. Rule. Simple and easy undo/redo. `http://www.codeproject.com/docview/undo.asp`. Last checked April 9, 2005.

[51] S. R. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill, 2002. ISBN 0-07-112263-X.

[52] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: good, fast, cheap persistence for C++. In *OOPSLA '92: Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*, pages 145–147. ACM Press, 1992. ISBN 0-89791-610-7.

[53] M. Skoglund. Sharing objects by read-only references. In *AMAST '02: Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, pages 457–472. Springer-Verlag, 2002. ISBN 3-540-44144-1.

[54] B. Stroustrup. *The C++ Programming Language, 3rd ed.* Addison Wesley, 1997. ISBN 0-201-88954-4.

[55] B. Stroustrup. Exception safety: concepts and techniques. *Advances in exception handling techniques*, pages 60–76, 2001.

[56] C. Sun. Undo any operation at any time in group editors. In *Proceedings of 2000 ACM Conference on Computer-Supported Cooperative Work*, pages 191–200, Dec. 2000.

[57] Sun Microsystems Inc. Java object serialization specification. Available at `http://java.sun.com/j2se/1.4.2/docs/guide/serialization/`. Last checked April 9, 2005.

[58] Sun Microsystems Inc. Java data objects specification, version 1.0.1, 2003.

[59] R. van Engelen. gSOAP: C/C++ web services and clients. `http://www.cs.fsu.edu/~engelen/soap.html`. Last checked April 9, 2005.

[60] I. Vargas, J. A. Borges, and M. A. Pérez-Quiñones. A usability study of an object-based undo facility. In *Proceedings of HCI International 2003*, 2003.

[61] J. S. Vitter. Us&r: A new framework for redoing (extended abstract). In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 168–176. ACM Press, 1984. ISBN 0-89791-131-8.

[62] M. Völter. Mementos basierend auf Java's Serialisierungsmechanismus. *JavaMagazin*, Oct. 2000.

[63] M. Völter. Transaktionen basierend auf Commands und Mementos. *JavaMagazin*, Nov. 2000.

[64] 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. ACM Press, 1991. ISBN 0-89791-451-1.

[65] X. Wang, J. Bu, and C. Chen. Achieving undo in bitmap-based collaborative graphics editing systems. In *CSCW '02: Proceedings of the 2002 ACM conference on Computer supported cooperative work*, pages 68–76. ACM Press, 2002. ISBN 1-58113-560-2.

[66] H. Washizaki and Y. Fukazawa. Dynamic hierarchical undo facility in a fine-grained component environment. In *Proceedings of the Fortieth International Confernece on Tools Pacific: Objects for internet, mobile and embedded applications*, volume 10, pages 191–199, 2002.

[67] E. D. Willink. *Meta-Compilation for C++*. PhD thesis, University of Surrey, June 2001.

[68] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.

[69] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), 1995.

[70] P. R. Wilson and S. V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *1992 Int. Workshop on Object Orientation and Operating Systems*, pages 364–377, Dourdan (France), 1992. IEEE Comp. Society Press.

[71] World Wide Web Consortium. Extensible markup language (XML). `http://www.w3.org/XML/`. Last checked April 9, 2005.

[72] M. Zhang and K. Wang. Implementing undo/redo in PDF Studio using object-oriented design pattern. In *TOOLS '00: Proceedings of the 36th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-Asia'00)*, page 58. IEEE Computer Society, 2000. ISBN 0-7695-0875-8.

[73] C. Zhou and A. Imamiya. Object-based nonlinear undo model. In *COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference*, pages 50–55. IEEE Computer Society, 1997. ISBN 0-8186-8105-5.

[74] R. E. Zundel, D. Mullin, and J. Synge. Method and apparatus for automatic undo support, 1999. United States Patent 6,543,006. See also `http://www.thecodeproject.com/cpp/transactions.asp` for another description of the technique. Last checked April 9, 2005.

[75] R. E. Zundel, D. Mullin, J. Synge, and S. Borduin. Method and apparatus for state-reversion, 1999. United States Patent 6,618,851.